Advanced Computer Architecture (191.019)

Lecture Notes

Repository

Contents

1 Block A

- **Chip Production**
  - Sand → Silicon Ingot → Wafer → Lithography

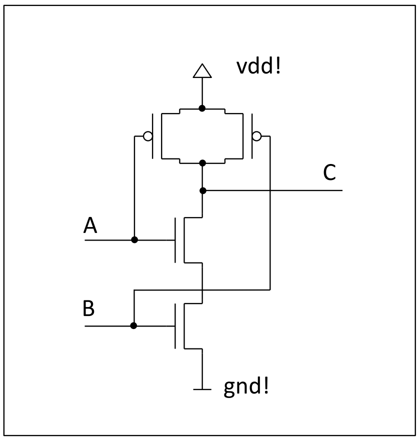


Figure 1.1: Standard Cell (`NAND`)

- Logic with `HIGH` and `LOW`

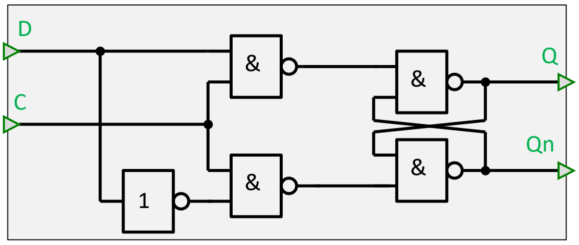


Figure 1.2: D-Latch

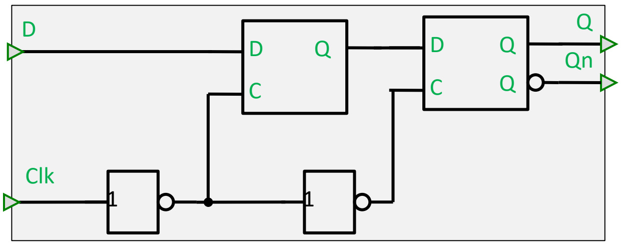
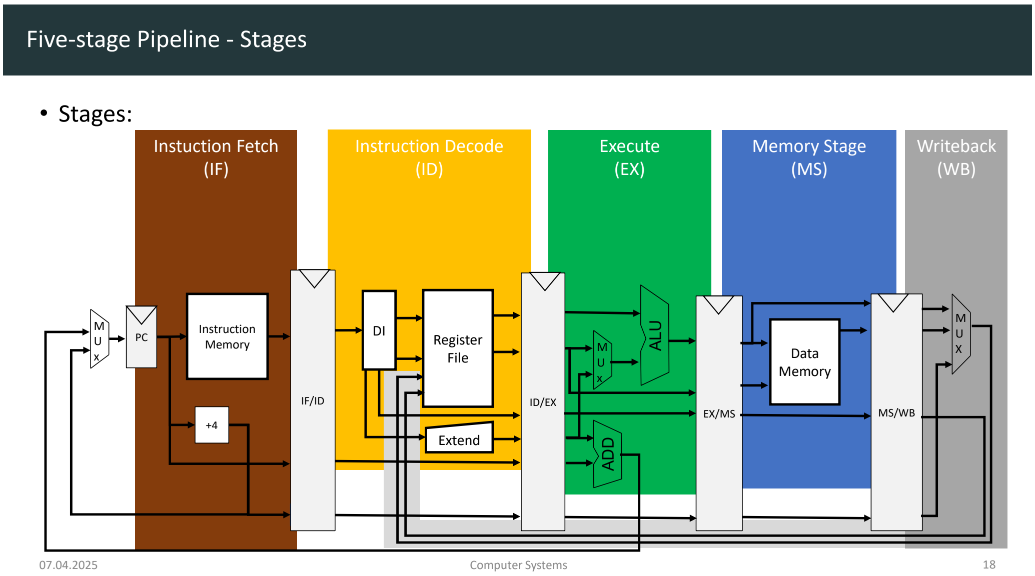


Figure 1.3: Edge-driven D Flip Flop



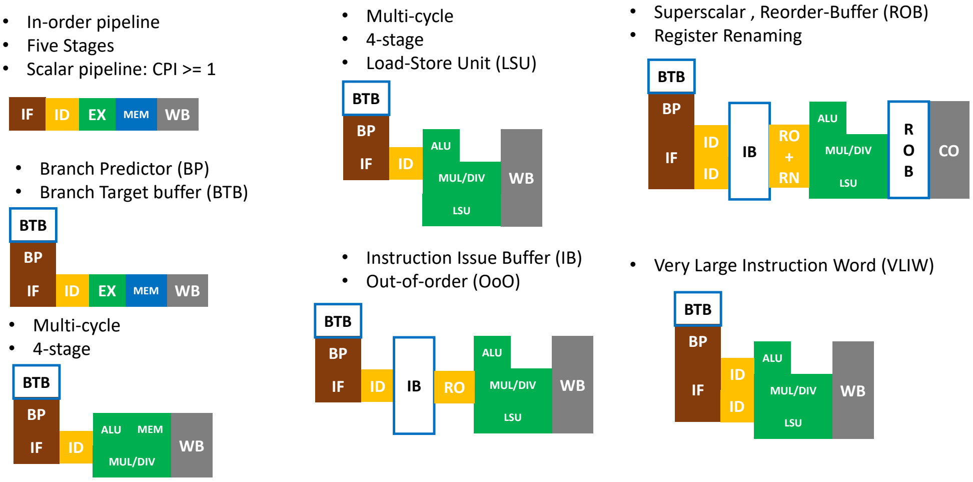
Five-stage Pipeline - Stages

- Stages:

07.04.2025 Computer Systems 18

Figure 1.4: Five-stage Pipeline

- Moore's Law
  - the number of transistors on chips doubles every two years
  - ≈ double the computational power on the same area
- Semiconductor Challenges
  - transistor density is coming close the the size of atoms
  - quantum effects start interfering
  - *N2* dimension
    - $2 \text{ nm} = 2 \cdot 10^{-9} \text{ m}$
  - *A14* has 14 ångström $(= 1,4 \text{ nm})$
    - atomic radius of silver $= 1,72$ ångström
  - **Power**: *"Dennard Scaling"*, hard to bring power to and heat from the chips
  - **Memory** bandwidth cannot keep up with processor performance



- In-order pipeline
- Five Stages
- Scalar pipeline: CPI >= 1

- Branch Predictor (BP)
- Branch Target buffer (BTB)

- Multi-cycle
- 4-stage

- Multi-cycle
- 4-stage
- Load-Store Unit (LSU)

- Instruction Issue Buffer (IB)
- Out-of-order (OoO)

- Superscalar , Reorder-Buffer (ROB)
- Register Renaming

- Very Large Instruction Word (VLIW)

Figure 1.5: Processor Pipelines

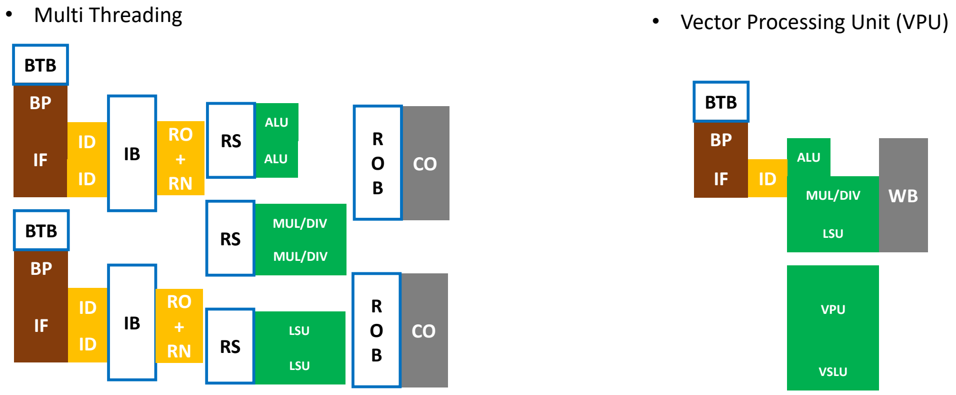


Figure 1.6: Multi-Threading and VPUs

2 Block B

2.1 RISC-V ISA and Compiler Basics

- Instruction Interface
  - *program* is a list of instructions
  - a *instruction* is a 16/32/64 bit value
    - single command
  - the *program counter* (`pc`) points to current position in the program
- Data Interface
  - Load Instruction
    - puts value on the address bus
    - receives a value on the `rdata` bus
  - Store Instruction
    - puts address on the address bus
    - puts value on the `wdata` bus
- Instruction Memory
  - used with `pc` to get the instructions to execute
- Data Memory
  - address is supplied
  - returns/gets the data value
- Register File Memory
  - some values are not stored in a memory but in a bank of registers
  - usually one or more register is read and usually one is written within one instruction
  - register address is small (4-5 bit), typically 16 or 32 values are stored
  - realized with a small SRAM with two read ports and one write port

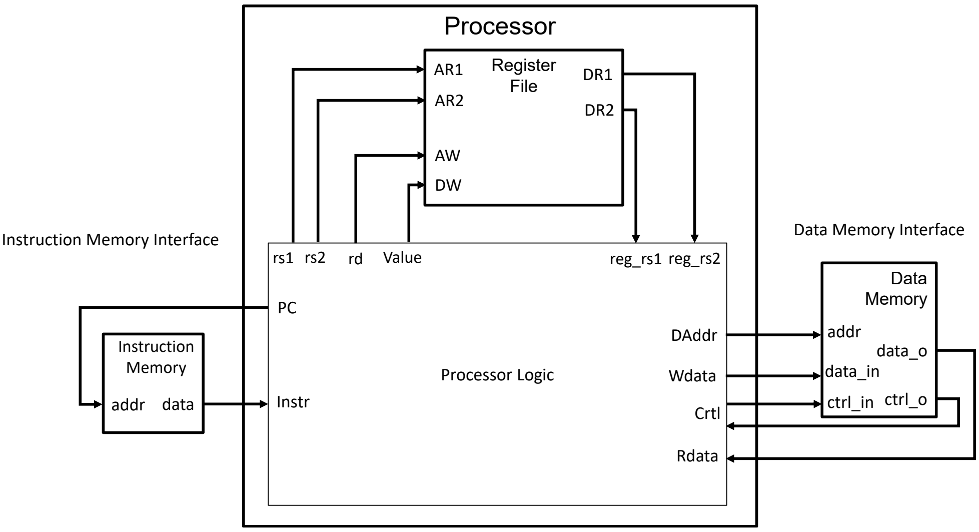


Figure 2.1: Register File inside a Processor

2.1.1 RISC-V Instruction Set Architecture

- Instruction Set Architecture (ISA) defines:
  - processor state organizations (registers)
  - what instructions a processor executes
    - how it is encoded in machine code

    - how assembly looks like
  - some behavior of the processor (exceptions, …)
- a cross-compiler can generate the assembly code for a different processor (as it's running on)
- *Microarchitecture*
  - processor model describes ISA, number of pipeline stages, …
- Why RISC-V?
  - open ISA
  - invented by UC Berkley
  - more and more SoCs are becoming available
  - has 32 registers

Register	ABI Name	Description	Saver
`x0`	Zero	hard-wired zero	-
`x1`	ra	return address	caller
`x2`	sp	stack pointer	callee
`x3`	gp	global pointer	-
`x4`	tp	thread pointer	-
`x5-7`	t0-2	temporaries	caller
`x8`	s0, fp	safed register, frame pointer	callee
`x9`	s1	saved register	callee
`x10-11`	a0-1	function arguments	caller
`x12-17`	a2-7	function arguments	caller
`x18-27`	s2-11	saved registers	callee
`x28-31`	t3-6	temporaries	caller

Table 2.1: RISC-V Register Names

- Application Binary Interface (ABI)
  - specified rules for register usage
    - passing arguments and results for function calls
  - *callee-saved*: called function has to restore modified values in these registers
  - *caller-saved*: called function can modify these and has not to restore them
  - assigns aliases for registers x0-x31 (see Table 2.1)

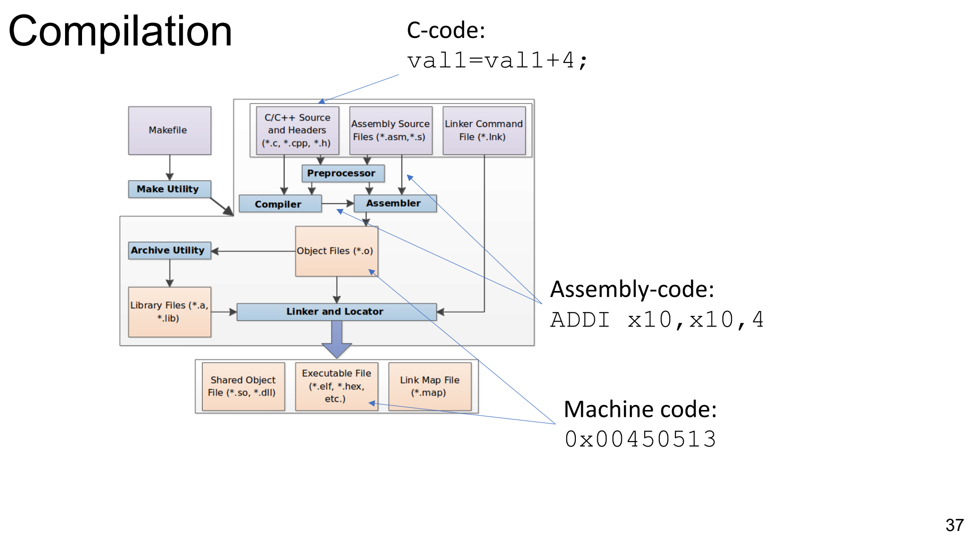
Instruction Format	Primary use	rd	rs1	rs2	Immediate
R-type	register-register, ALU instructions	destination	first source	second source	
I-type	ALU immediate, Load	destination	first source base register		value displacement
S-type	Store, Compare and Branch		base register first source	data source to store second source	displacement offset
U-type	Jump and Link, Jump and Link Register	register destination for return `pc`	target address for jump and link register		target address for jump and link

Table 2.2: RISC-V Instruction Formats

- 32-bit Instructions

  - Integer Register-Register Instructions (R-type)
    - Runs an arithmetic or logical operation on registers
    - Both operands are values in registers
  - Integer Register-Intermediate Instructions (I-type)
    - Second operand is a immediate (constant) value
    - Immediate is encoded in the machine code
    - there is no `SUBI`, use addition with negative immediate
  - Control Transfer Instructions
    - Unconditional jumps
    - Conditional Branches
  - Load Store Instructions
    - Move data between memory and registers
    - Load-store Architecture: Operations on registers only
  - Examples
    - `ADD a1, a2, a3`
      - regs[a1] = regs[a2] + regs[a3]
    - `SUB a1, a2, a3`
      - regs[a1] = regs[a2] - regs[a3]
    - Move is a pseudo instruction
      - actually implemented as a `ADDI a1, a2, 0`
- Control Transfer Instructions - Jumps
  - Unconditional Jump (pc relative)
    - `J 8` ⇒ `pc = pc + (8 << 1)`
    - `J` is a pseudo instruction: `JAL zero, 8`
  - Unconditional Jump and Link (pc relative)
    - `JAL ra, 8` ⇒ `regs[ra] = pc + 4; pc = pc + (8 << 1)`
    - this is used for function calls, where we want to return to the main control flow later
  - Unconditional Jump and Link Register (register with offset)
    - `JALR rd, rs1, imm` ⇒ `pc = (regs[rs1] + imm) & ~1; regs[rd] = pc+4`
    - `RET` is a pseudo instruction: `JAR zero, ra, 0`
- Control Transfer Instructions – Branches
  - `BEQ a1, a2, loop_start` ⇒ `if (reg[a1] == reg[a2]) pc = loop_start else nothing`
  - further branch instructions
    - not equal: `BNE`
    - less than: `BLT`
    - less than (unsigned): `BLTU`
    - greater or equal than: `BGE`
    - greater or equal than (unsigned): `BGEU`
- Load Store Instructions
  - Load Word (4 byte): `LW a1, 80(a2)`
    - ⇒ `reg[a1] = mem[80 + reg[a2]]`
  - Store Word (4 byte): `SW a1, 80(a2)`
    - ⇒ `mem[80 + reg[a2]] = reg[a1]`
  - also options with halfword (`lh`/`sh`) and byte (`lb`/`sb`) (**with** sign extension)
  - to avoid sign extension of halfword and byte loads: `lhu`, `lbu`
- Integer Multiplication Instructions (**M**-Extension)
  - multiplying two 32 bit values can result in a 64 bit value
  - (signed) Only the lower 32 bit: `MUL rdl, rs1, rs2`
  - (signed) Only the higher 32 bit: `MULH rdh, rs1, rs2`
  - unsigned-unsigned version: `MULU` and `MULHU`
  - unsigned-signed version: `MLSU` and `MULHSU`

- Integer Divison Instructions (**M**-Extension)
  - (signed) Divison: `DIV rdl, rs1, rs2`
  - (signed) Remainder: `REM rdh, rs1, rs2`
  - unsigned-unsigned version: `DIVU` and `REMU`
  - unsigned-signed version: `DIVSU` and `REMSU`



Compilation

C-code:
`val1=val1+4;`

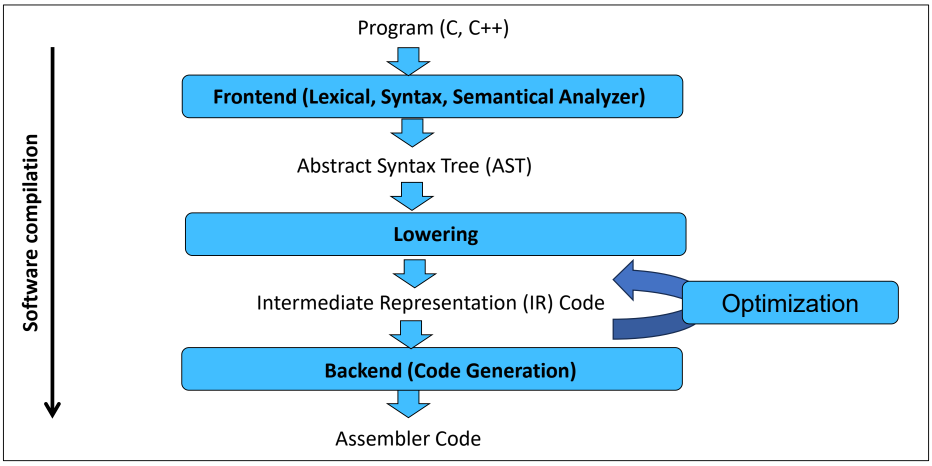
Assembly-code:
`ADDI x10,x10,4`

Machine code:
`0x00450513`

37

Makefile
Make Utility
C/C++ Source and Headers (*.c, *.cpp, *.h)
Assembly Source Files (*.asm,*.s)
Linker Command File (*.lnk)
Preprocessor
Compiler
Assembler
Archive Utility
Object Files (*.o)
Library Files (*.a, *.lib)
Linker and Locator
Shared Object File (*.so, *.dll)
Executable File (*.elf, *.hex, etc.)
Link Map File (*.map)

Figure 2.2: Compilation with different stages of code representation

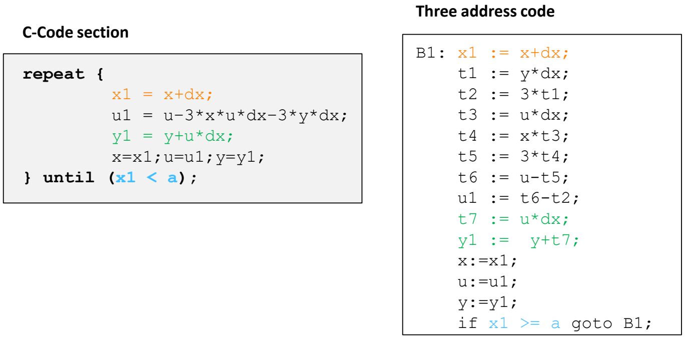


Program (C, C++)
Frontend (Lexical, Syntax, Semantical Analyzer)
Abstract Syntax Tree (AST)
Lowering
Intermediate Representation (IR) Code
Optimization
Backend (Code Generation)
Assembler Code
Software compilation

Figure 2.3: Compiler Frontend and Backend

- Lexical Analysis (*Scanning*)
  - reads stream of characters and groups them in meaningful sequences (*lexemes*)
- Syntax Analysis (*Parsing*)
  - reads token stream and outputs the syntax tree
- Semantical Analysis
  - reads abstract syntax tree and checks against semantics of programming language
  - adds type casts
- Intermediate Representation (IR)
  - *Three Address Code*
    - maximal 3 addresses per operation
    - examples:
      - `x := y op z` with op $\in \{+, -, *, /, \hat{}, \&, ...\}$
      - `x := y`

- `goto Bx`
- `if x relop y goto Bx`
  - with relop $\in \{=, \leq, \geq, <, >, \neq, ...\}$



C-Code section

```
repeat {
        x1 = x+dx;
        u1 = u-3*x*u*dx-3*y*dx;
        y1 = y+u*dx;
        x=x1;u=u1;y=y1;
} until (x1 < a);
```

Three address code

```
B1: x1  := x+dx;
    t1  := y*dx;
    t2  := 3*t1;
    t3  := u*dx;
    t4  := x*t3;
    t5  := 3*t4;
    t6  := u-t5;
    u1  := t6-t2;
    t7  := u*dx;
    y1  :=  y+t7;
    x:=x1;
    u:=u1;
    y:=y1;
    if x1 >= a goto B1;
```

Figure 2.4: Example Code

- Static Single Assignment (SSA)
  - all assignments are to variables with distinct names
  - Φ-Operator chooses the assigned value for recombination of two values

Normal Code	SSA Code

`p := a+b`
`q := p-c`
`p:= q*d`
	`p$1 := a+b`
	`q := p$1-c`
	`p$2 := q*d`

Table 2.3: Normal Code vs. SSA

```
    if (a>b) goto B1
    p$1 := a-b
    goto B2
B1: p$2 := a+b
B2: p$3 := Phi(p$1, p$2)
```

Listing 2.1: Phi Operator

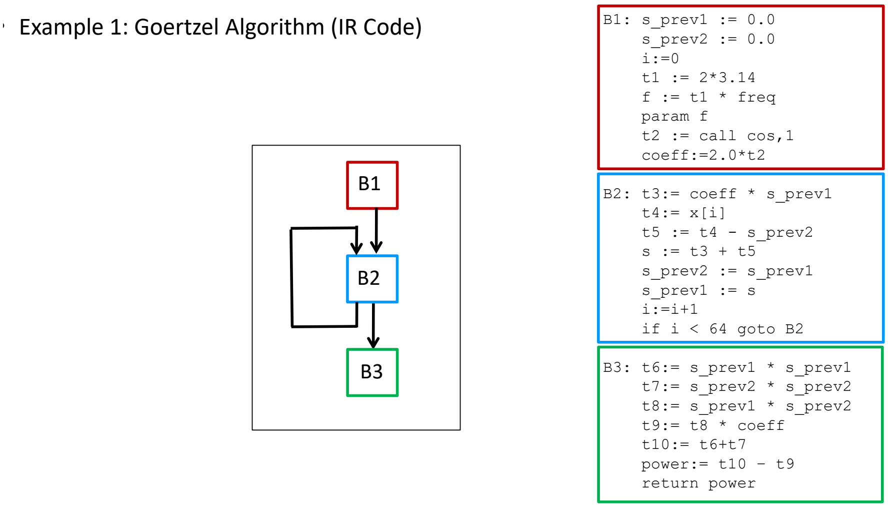
- LLVM Intermediate Representation
  - CLANG is the frontend
  - LLVM has many targets
  - IR
    - in SSA
    - evolves with LLVM, but minor changes
    - variables are marked with `%`
    - has datatypes

2.2 Static Code Analysis

2.2.1 Control and Data Flow Analysis

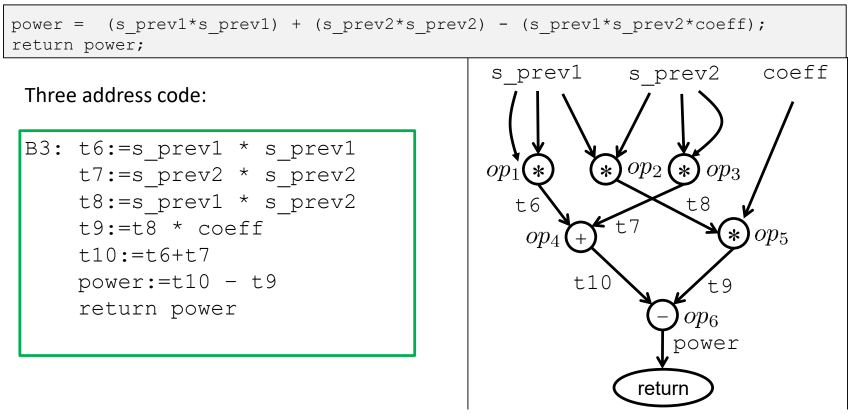
- Basic Block
  - maximal sequence of instructions with
    1. no jump target labels (except at the first line)
       - cannot jump into a basic block
    2. no jump (execpt last "return" instruction)
       - cannot jump out of a basic block
  - single-entry, single-exit, straight-line code segment
- Control Flow Graph (CFG)
  - $G_c(V, E)$
  - nodes $V = \{B_i : i = 1, ..., n_B\}$
    - are basic blocks of the algoritm
  - edges $E = \{(B_i, B_j) : i, j = 1, ..., n_B\}$
    - branches, cycles

- Data Flow Graph (DFG) is a directed acyclic graph
  - describe data dependencies in a basic block
  - Nodes: operations in block
  - Edges: Data dependencies between operations
- paths in DFG describe concurrent operations, that may be executed in parallel



```
B1: s_prev1 := 0.0
    s_prev2 := 0.0
    i:=0
    t1 := 2*3.14
    f := t1 * freq
    param f
    t2 := call cos,1
    coeff:=2.0*t2
B2: t3:= coeff * s_prev1
    t4:= x[i]
    t5 := t4 - s_prev2
    s := t3 + t5
    s_prev2 := s_prev1
    s_prev1 := s
    i:=i+1
    if i < 64 goto B2
B3: t6:= s_prev1 * s_prev1
    t7:= s_prev2 * s_prev2
    t8:= s_prev1 * s_prev2
    t9:= t8 * coeff
    t10:= t6+t7
    power:= t10 - t9
    return power
```

Figure 2.5: Example of Basic Blocks



```
power =  (s_prev1*s_prev1) + (s_prev2*s_prev2) - (s_prev1*s_prev2*coeff);
return power;
```

Three address code:

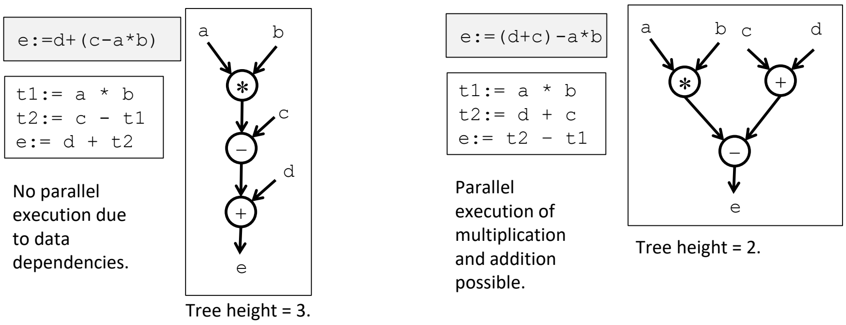
```
B3: t6:=s_prev1 * s_prev1
    t7:=s_prev2 * s_prev2
    t8:=s_prev1 * s_prev2
    t9:=t8 * coeff
    t10:=t6+t7
    power:=t10 - t9
    return power
```

Figure 2.6: Inside a Basic Block with Three Address Code

2.2.2 Code Optimization

- Techniques, mostly executed only on one single basic block
  - common subexpression elimination
    - two instructions execute same operation on the same operands (can be easily seen in SSA)
    - one operation can be replaced by a copy statement
  - dead code elimination
    - can be identified with CFG (leads to nowhere)
  - arithmetic identities
    - remove stuff like $a + 0$, $b \cdot 1$, $c/1$, $\frac{a}{b} \cdot b$
  - strength reduction
    - replace operation with equivalent operations that is cheaper to execute in hardware
    - $x \cdot 2 \Rightarrow x \ll 1$
  - constant folding (propagation)
    - calculate constant expressions at compile time: $2 \cdot 5 + 6 = 16$
  - tree height reduction
    - increase possible concurrency by avoiding data dependencies

    - better for multi-issue processors, multi-threading



```
e:=d+(c-a*b)
```

```
t1:= a * b
t2:= c - t1
e:= d + t2
```

No parallel execution due to data dependencies.

```
e:=(d+c)-a*b
```

```
t1:= a * b
t2:= d + c
e:= t2 - t1
```

Parallel execution of multiplication and addition possible.

Figure 2.7: Tree Height Reduction

- Global Code Optimization Techniques
  - considering more than one basic block
  - global common subexpression elimination
  - global dead code elimination
  - code motion
    - move statements out of the loop, if their value is independent from loop iteration
  - induction variable reduction
    - induction variables change by constant value in each iteration of loop
    - apply strength reduction and common subexpression elimination on induction variables
  - loop unrolling
    - loop classification
      - *do-all loops*: no data dependencies between loop iterations
      - *do-across loops*: exists possible data dependencies between loop iterations
    - execute serveral loop iterations in one single iteration of optimized loop
    - *unroll factor*: number of non-optimized loop iterations executed in one iteration of optimized loop

2.2.3 Live Variable Analysis

- Dataflow Analysis
  - determines dataflow values at each point in the program
  - values **before** IR statement: s_i : $\mathrm{IN}[s_i]$
  - values **after** IR statement: s_i : $\mathrm{OUT}[s_i]$
  - each IR statement applies a *transfer function* on the dataflow values:
    - forward flow analysis: $\mathrm{OUT}[s_i] = f_{s,i}(\mathrm{IN}[s_i])$
    - backward flow analysis: $\mathrm{IN}[s_i] = f_{s,i}(\mathrm{OUT}[s_i])$
    - transfer function of a basic block is the composition of the transfer functions of all statements
- Controlflow Constraints
  - within one basic block, the dataflow values after an IR statement are the same as before the next statement
  - between basic blocks:
    - *forward flow problem*: values at the entry to the basic block are the union of the values at the end of all predecessors
    - *backward flow problem*: values at the end of the basic block are the union of the values at the entry of all its successors
- Variable Liveliness Analysis
  - variables at entry to basic block B_x: $\mathrm{IN}[B_x]$
  - variables at end of basic block B_x: $\mathrm{OUT}[B_x]$
  - set of defined variables: $\mathrm{DEF}[B_x]$
  - set of used variables: $\mathrm{USE}[B_x]$

- Conditions for Live Variable Analysis
  1. is a backward flow analysis
  2. transfer function: $\mathrm{IN}[B_x] = \mathrm{USE}[B_x] \cup (\mathrm{OUT}[B_x] - \mathrm{DEF}[B_x])$
  3. control flow constraints: $\mathrm{OUT}[B_x] = \bigcup_{S_x} \mathrm{IN}[S_x]$
     - S_x are successor basic blocks of B_x
  4. boundary condition: $\mathrm{IN}[\mathrm{END}] = \{\}$

- Why is variable lifetime important?
  - compiler does not need to assign registers to all variables during their full lifetime
    - can be *spilled* (moved in the stack)
  - number of needed registers depends on numver of variables which are live at a certain point in the program
  - registers can be reused in case that the lifetime of variables does not overlap

3 Block C

3.1 Scalar Pipeline and Branch Prediction

3.1.1 In-order Scalar Processor Pipeline

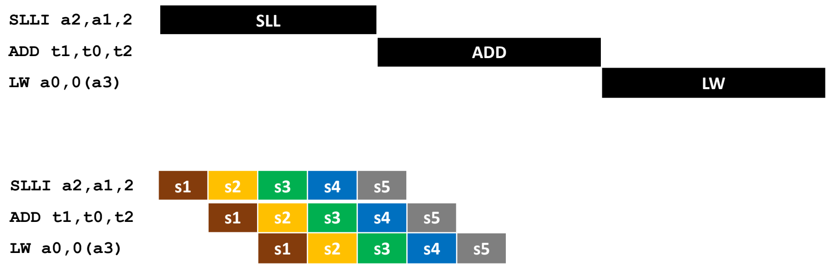
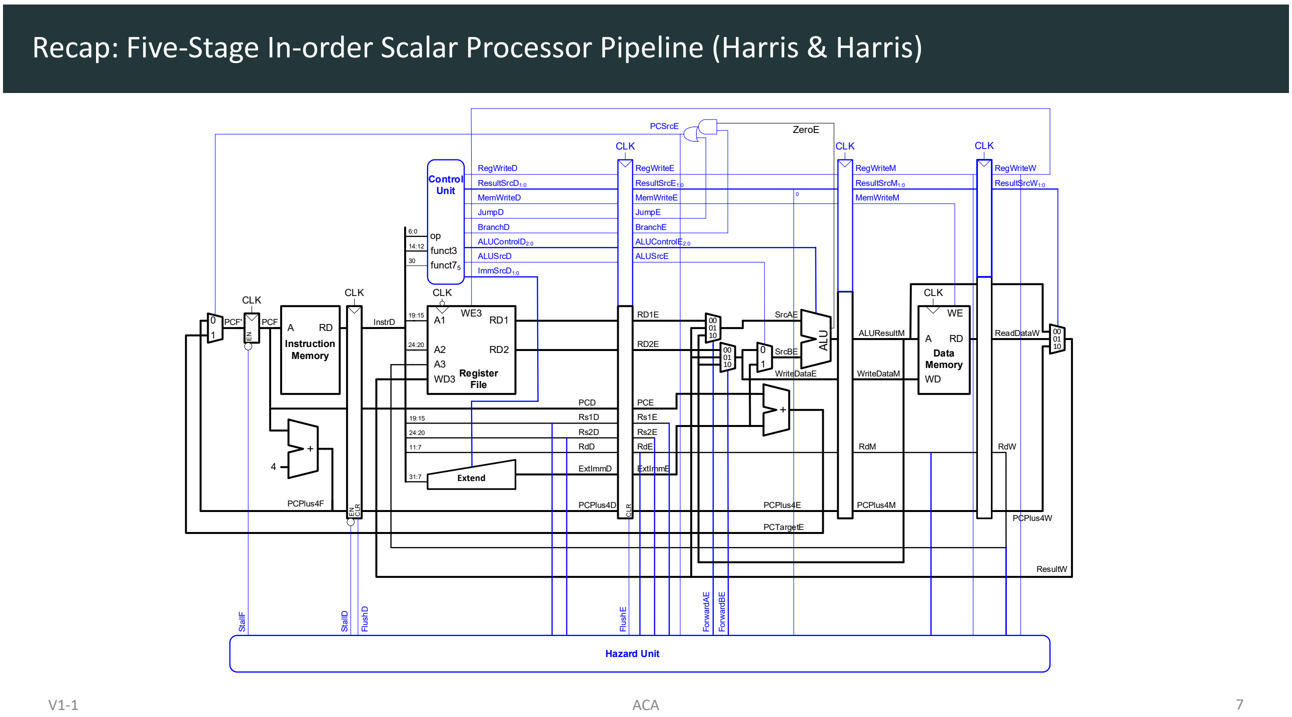
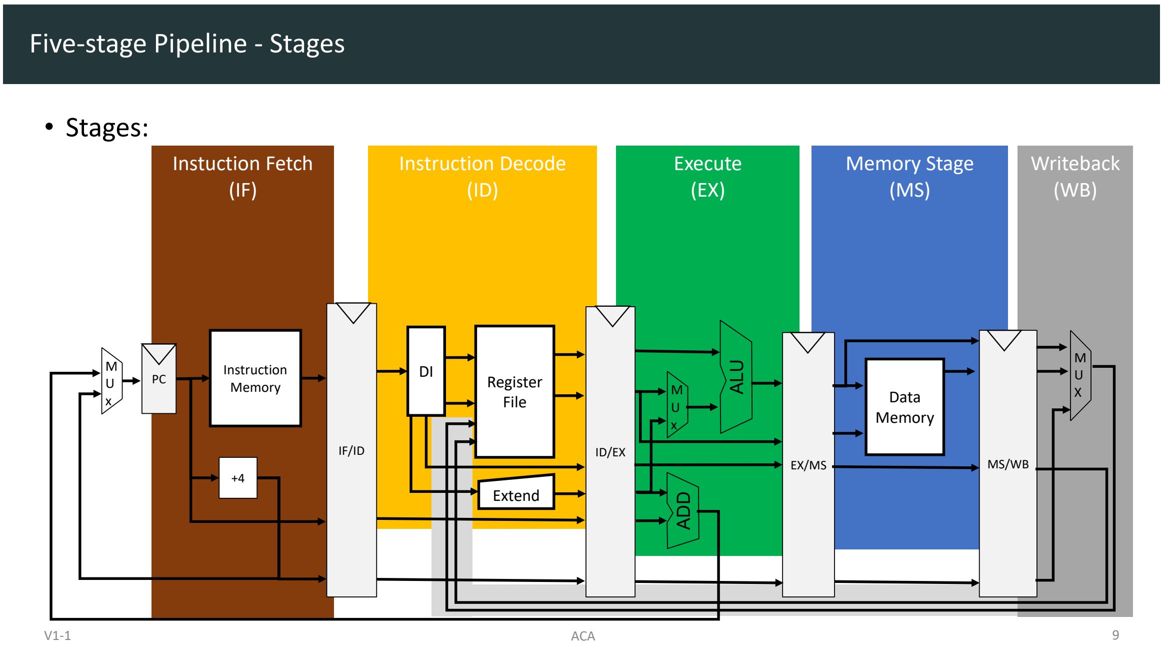


Figure 3.1: Instructions in a pipelined fashion



Recap: Five-Stage In-order Scalar Processor Pipeline (Harris & Harris)

Figure 3.2: Five State In-order Scalar Processor Pipeline



Five-stage Pipeline - Stages

- Stages:

Figure 3.3: Pipeline split up into Stages

Stages and Subcomputations

- **Instruction Fetch** (*IF*)
  - fetch next instruction, next Program Counter (PC)
- **Instruction Decode** (*ID*)
  - **DI**: decode instruction
  - **RF**: read operands
  - **Extend**: sign extend immediate, sign extend and shift or LUI, AUIPC, BX
- **Execute** (*EX*)
  - **Arithemtic Logic Unit (ALU)**: compare result, compute address, comparison (branch taken/not taken), compute JR branch target address
  - **ADD**: compute branch target address
- **Memory Stage** (*MS*)
  - read/write data memory
- **Write Back** (*WB*)
  - write result

- data hazards can be effectively mitigated using a forward path
  - named *"forward path"* but the signal buses go *back* in the pipeline
- RET is a pseudo-instruction
  - `RET` → `JR ra` → `JALR x0, ra, 0`
- the *Harris* Pipeline does not support to load a register value into PC
  - another bus needed for implementing the `JR` instruction

3.1.2 Data Hazards

- due to forward path: possible data hazard after loading with penalty of 1 clock cycle
- Read after Write (RAW)
  - one instructions reads operand that is written as result of previous instructions
- compiler can often move instructions to avoid RAW data hazards after loads
  - program order must not change

3.1.3 Control Hazards

- control hazards arise from instructions that change the PC
  - when the flow of instructions addresses is **not sequential**
    - unconditional branches (`jal`, `jalr`)
    - conditional branches (`beq`, `bne`, …)
    - exceptions
- possible approaches
  - **stall**: impacts Cycles per Instructions (CPI)
  - **move decision point** as early in the pipeline as possible (extra hardware)
  - **predict** and hope for the best 🙏
  - **delay decision** (requires compiler support)
- control hazards occur **less frequently** than data hazards
  - but they cannot be solved as effectively as data hazards with forwarding
- branches determine flow of control
  - fetching next instruction depends on branch outcome
  - PC is either $(PC + 4)$ or $(PC + \text{imm} \ll 1)$
- **Stall on Branch**
  - *conservative* approach: wait until branch outcome determined before fetching next instruction
- **Reducing Branch Delay**
  - move branch decision to ID stage
  - reduce cost of the taken branch
  - target address adder in ID
  - branch penalty: only one clock cycle

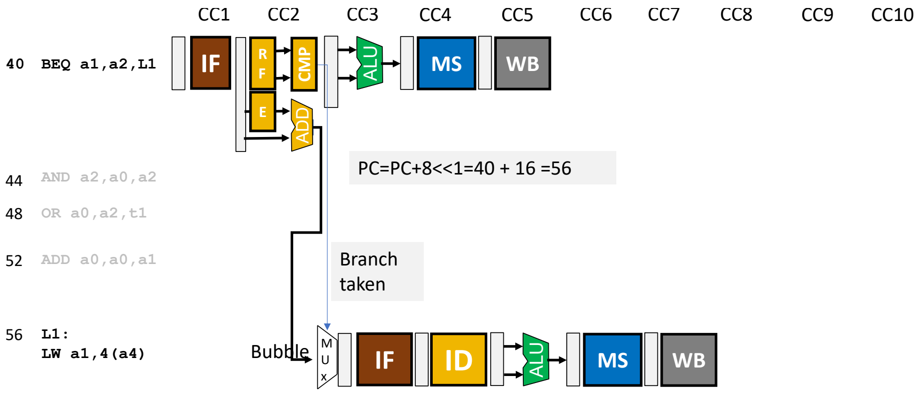


Figure 3.4: Move Branch Decision to ID Stage

3.1.4 Static Branch Prediction

- longer pipelines cannot determine branch outcome early
  - this means branch penalty becomes unacceptable
- Simple Static Branch Prediction Schemes
  - **always not taken**
  - **always taken**
- *Always Not Taken*
  - *Correct Prediction*
    - penalty: 0 Clock Cycles (CC)
  - *Incorrect Prediction*
    - penalty: 2 CC
    - flush instructions from pipeline
- *Always Taken*
  - *Correct Prediction*

    - branch target address is computed in the EX stage
    - Branch Target Buffer (BTB) stores the *Branch Target Address (BTA)* for a certain branch (see Figure 3.5)
    - content addressable memory
    - lookup via PC
    - *BTB has entry*
      - BTA via BTB
      - penalty: 0 CC
  - *Incorrect Prediction*
    - penalty: 2 CC (just as with *Always not taken*)
    - flush instructions from pipeline
- Statistics
  - typical statistics: 60% to 70% branches are taken
  - *always not taken*: 62% mispredictions
  - *always taken*: 38% mispredictions
  - **Backward Taken, Forward Not Taken (BTFNT)**
    - *forward branches not taken*: ≈ 10% mispredictions
    - *backward branches taken*: ≈ 20% mispredictions

$$\mathrm{CPI} = 1 + b \cdot p \cdot m$$

$$b\ldots \text{ relative number of branch instructions}$$

$$p\ldots \text{ cycle penalty for mispredictions}$$

$$m\ldots \text{ misprediction rate}$$

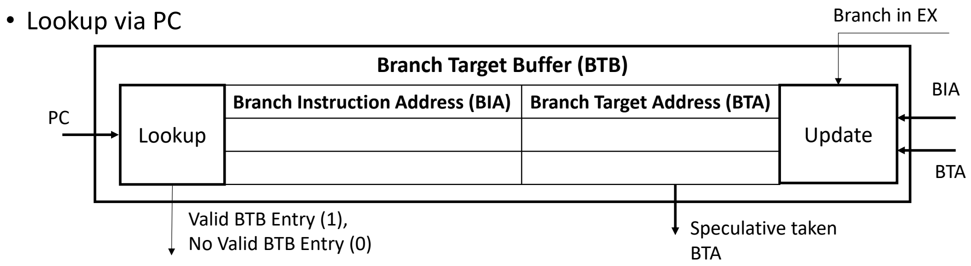


Figure 3.5: Branch Target Buffer (BTB)

3.1.5 Dynamic Branch Prediction

i Note

In longer pipelines, branch penalty is more significant

- *Branch Prediction Buffer* (aka. *Branch History Table (BHT)*)
  - stores last outcome (taken, not taken)
  - check table, expect the same outcome
  - start fetching from fall-through (*not taken*) or target (*taken*)
  - in case of misprediction, flush pipeline and flip prediction

💡 Tip

For example in a loop the taken branch is way more common than not taken (just last iteration)

- **Single-Bit/1-Bit/Last-Time Predictor**
  - indicates which direction the branch went last time it executed
  - PNT: Predict NT (bit=0): fetch the instruction from PC+4

  - PT: Predict T (bit=1): get target address from the BTB
  - see Figure 3.6

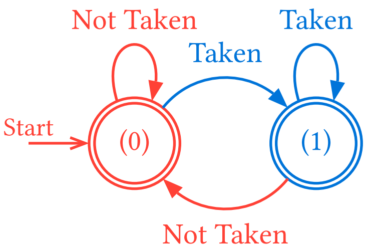


Figure 3.6: Single-Bit Predictor

- **Global Predictor**
  - one single Branch History Entry for all branches
- **Local Predictor**
  - one entry for each BTB entry
- **2-Bit Predictor**
  - prediction does not change on a single misprediction
  - PNST: Strongly Not Taken (00), PWNT: Weakly Not Taken (01)
  - PWT: Weakly Taken (10), PST: Strongly Taken (11)
  - see Figure 3.7
  - a prediction must be wrong twice (consecutively) before the prediction is changed

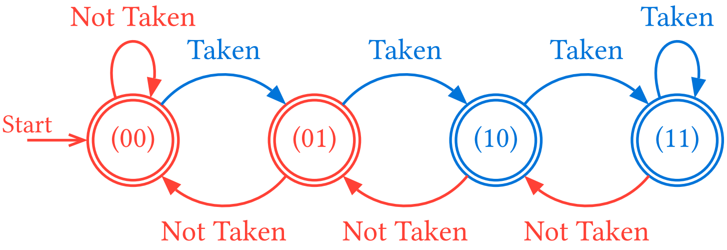


Figure 3.7: 2-Bit Predictor

3.1.6 A Look at a Real Processor - CVA6

This is not relevant for the exam

3.1.7 A Look at a Real Processor - ESP32-C3

This is not relevant for the exam

3.1.8 A Look at a Real Processor - Trap Handling

This is not relevant for the exam

3.2 From Scalar to Superscalar

3.2.1 Multi-Cycle Operations

Pipelined Functional Unit (FU)

- complex computations require deep circuit logic
- critical paths limits the design's frequency
- same is as processor design: break FU into stages and integrate registers (pipeline)
- *latency* is the number of pipeline stages
- *initialization interval*: delay between start of two computations

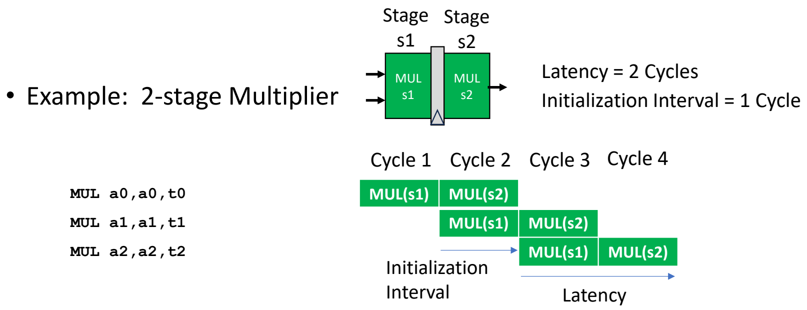


Figure 3.8: Example: 2-stage Multiplier

Serial FU

- often complex operations such as divisions can be computed by iterative algorithms
- the number of iterations often depends on the input values
  - can be implemented on a serial FU

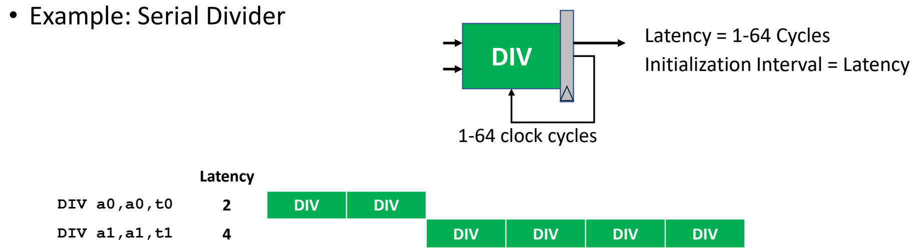


Figure 3.9: Example: Serial Divider

Multi-cycle Functional Units are integrated into the **EX Stage**

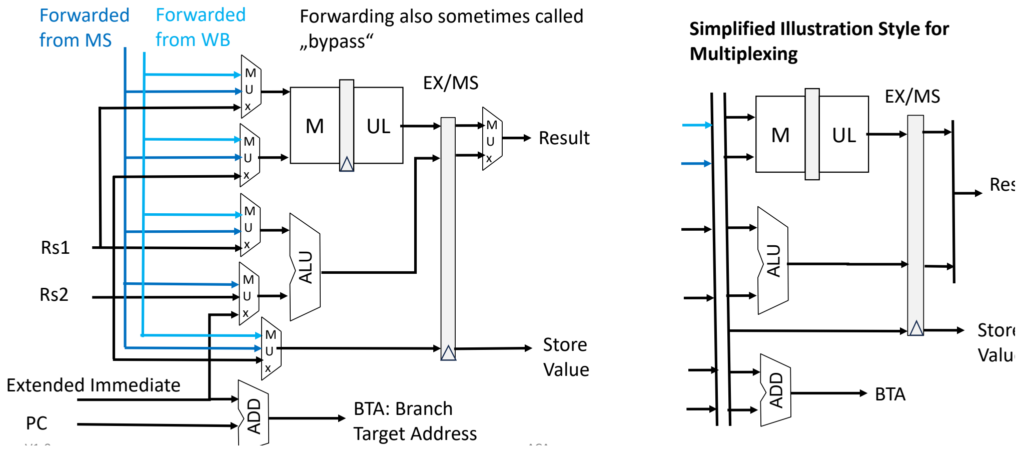


Figure 3.10: Example: for Multiplier in EX Stage

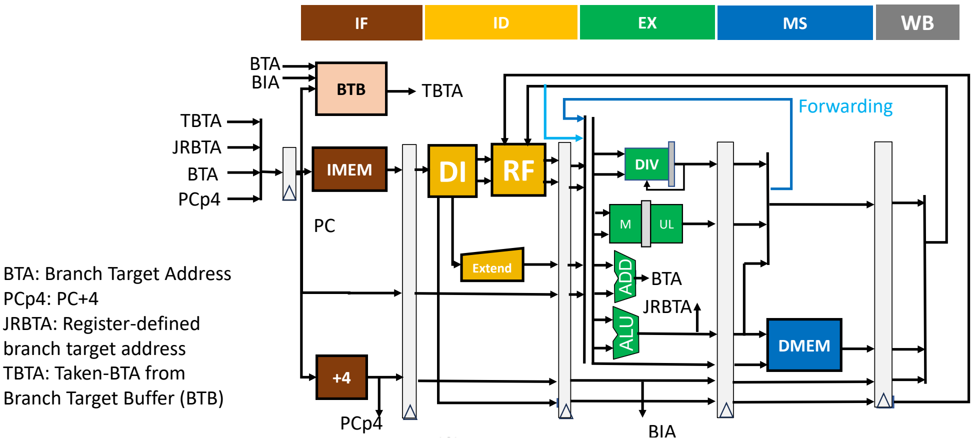

IF
ID
EX
MS
WB
BTA
BIA
BTB
TBTA
TBTA
JRBTA
BTA
PCp4
IMEM
PC
DI
RF
Forwarding
DIV
M
UL
Extend
ADD
BTA
ALU
JRBTA
DMEM
+4
PCp4
BIA
BTA: Branch Target Address
PCp4: PC+4
JRBTA: Register-defined branch target address
TBTA: Taken-BTA from Branch Target Buffer (BTB)

Figure 3.11: Example: Multiplier in Full Pipeline

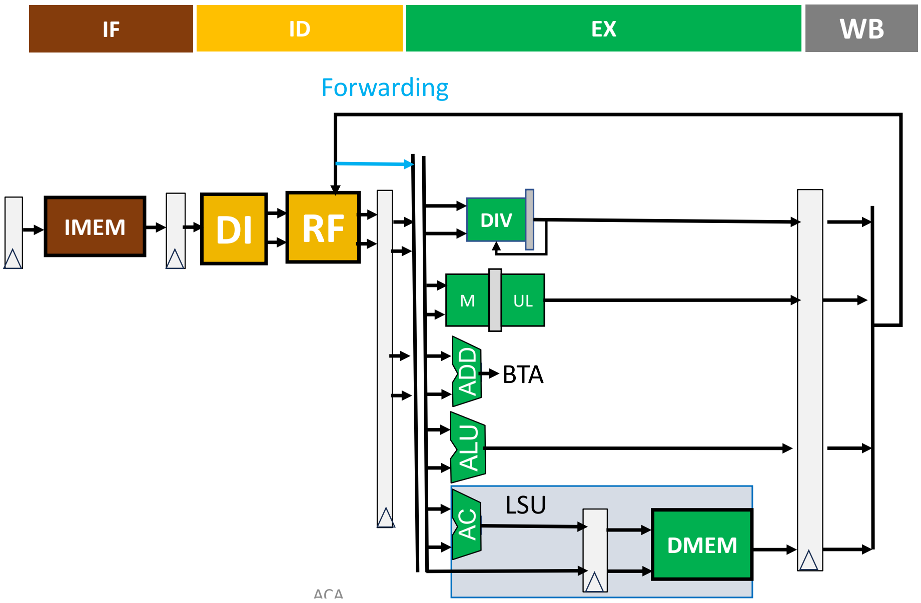
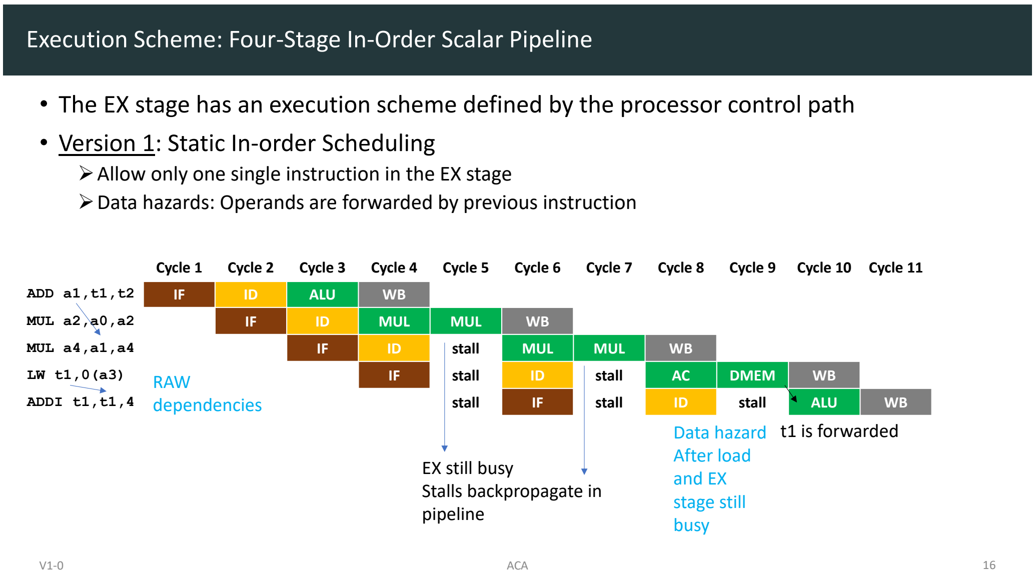


Figure 3.12: Example: with second Address Adder (AC) -> simple Load/Store Unit



Execution Scheme: Four-Stage In-Order Scalar Pipeline

- The EX stage has an execution scheme defined by the processor control path
- Version 1: Static In-order Scheduling
  - ➢ Allow only one single instruction in the EX stage
  - ➢ Data hazards: Operands are forwarded by previous instruction

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10	Cycle 11
`ADD a1,t1,t2`	IF	ID	ALU	WB							
`MUL a2,a0,a2`		IF	ID	MUL	MUL	WB					
`MUL a4,a1,a4`			IF	ID	stall	MUL	MUL	WB			
`LW t1,0(a3)`				IF	stall	ID	stall	AC	DMEM	WB	
`ADDI t1,t1,4`					stall	IF	stall	ID	stall	ALU	WB

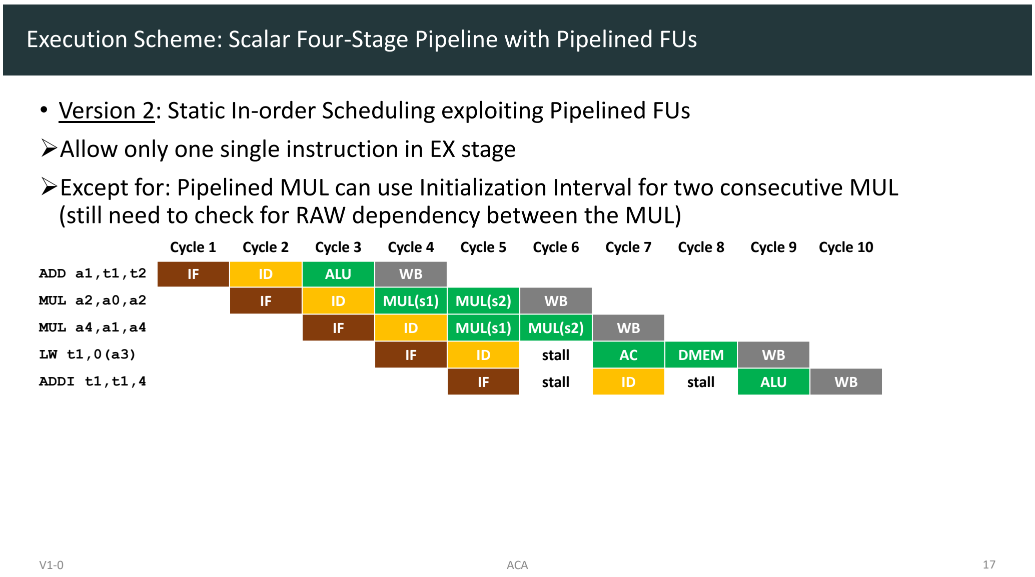
RAW dependencies

EX still busy
Stalls backpropagate in pipeline

Data hazard After load and EX stage still busy

t1 is forwarded

V1-0 ACA 16



Execution Scheme: Scalar Four-Stage Pipeline with Pipelined FUs

- Version 2: Static In-order Scheduling exploiting Pipelined FUs

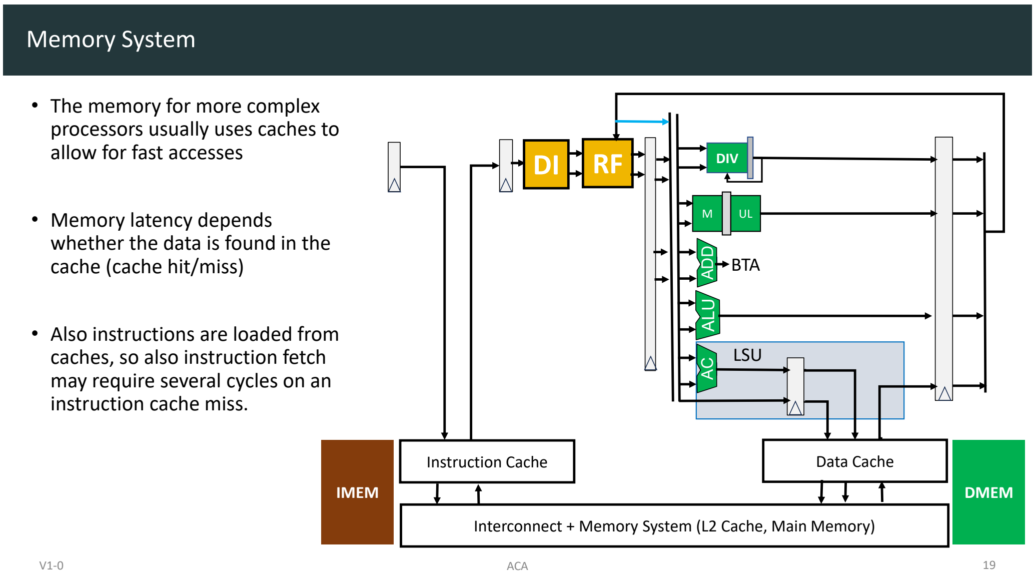
➢Allow only one single instruction in EX stage

➢Except for: Pipelined MUL can use Initialization Interval for two consecutive MUL (still need to check for RAW dependency between the MUL)

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
`ADD a1,t1,t2`	IF	ID	ALU	WB						
`MUL a2,a0,a2`		IF	ID	MUL(s1)	MUL(s2)	WB				
`MUL a4,a1,a4`			IF	ID	MUL(s1)	MUL(s2)	WB			
`LW t1,0(a3)`				IF	ID	stall	AC	DMEM	WB	
`ADDI t1,t1,4`					IF	stall	ID	stall	ALU	WB

V1-0 ACA 17

3.2.2 Load/Store Optimizations



Memory System

- The memory for more complex processors usually uses caches to allow for fast accesses
- Memory latency depends whether the data is found in the cache (cache hit/miss)
- Also instructions are loaded from caches, so also instruction fetch may require several cycles on an instruction cache miss.

V1-0 ACA 19

Instruction Cache Misses

- causes several cycles of delay for instruction fetch (IF)
  - depending on speed to catch fresh instruction block from memory
- instructions are usually reloaded to cache in blocks (*cache line size*)
  - so that usually there are several cahce hits after a cache miss
- advanced caches pre-fetch the next block before the cache miss happens
  - to hide cache refill latencies

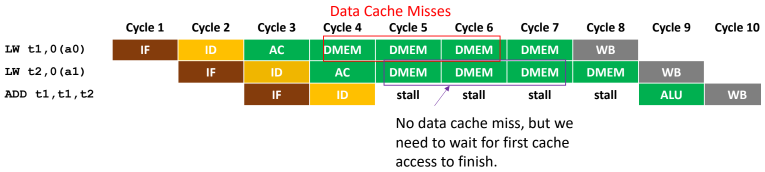
Load Cache Miss

- data cache misses lead to extra cycles for loads as the data needs to get fetched from another memory
  - (e.g. Level 2 Cache, Main Memory, ...)
- due to this the pipeline has to be stalled

Non-blocking Loads

- load access are far longer times *in flight* due to cache misses
- most interconnects/caches allow overlap mulitple memory accesses

Example: cache observes both addresses for load accesses and may need to reload cache lines for both accesses when both miss



Data Cache Misses

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
LW t1,0(a0)	IF	ID	AC	DMEM	DMEM	DMEM	DMEM	WB		
LW t2,0(a1)		IF	ID	AC	DMEM	DMEM	DMEM	DMEM	WB	
ADD t1,t1,t2			IF	ID	stall	stall	stall	stall	ALU	WB

No data cache miss, but we need to wait for first cache access to finish.

Figure 3.16: Example with non-blocking loads

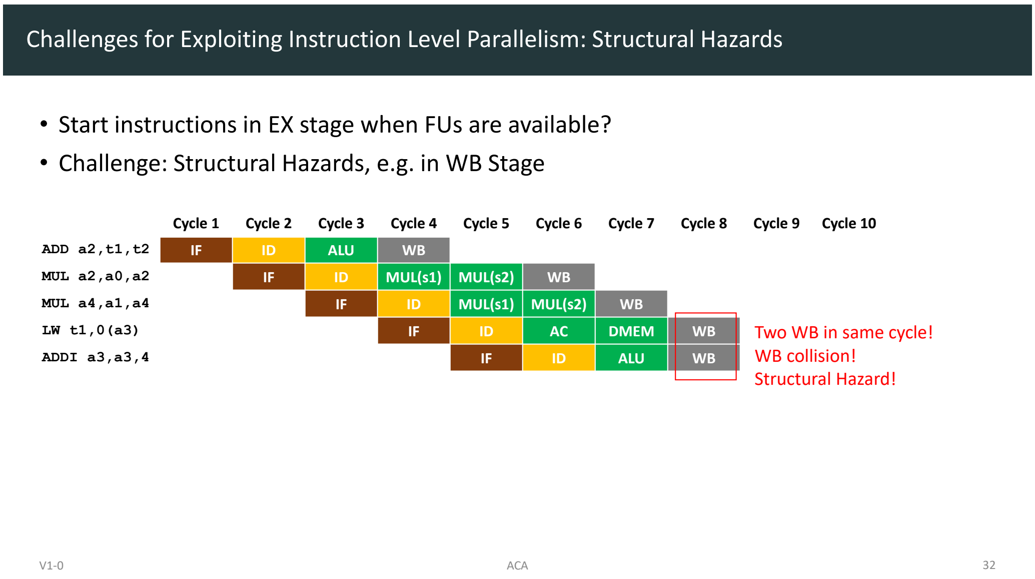
Store Cache Miss

- depending on *Store Policy*
  - additional latencies for store possible when a dirty cache line needs to be replaced (first needs to be written back, before new line can be loaded)
- Write Through Data Cache
  - long store latency because the data is written not only to cache but also main memory

Buffers

- a buffer can store several values
- *FIFO* (first in, first out)
  - buffer values can only be read in the same order they are written to
- *Reorder Buffer*
  - can look up and read any value in the buffer
- **Store Buffer**
  - not needed to wait until a store write is complete
  - *Store Unit (SU)* with Store Buffer
    - put store address and data to store buffer (aka *"Posted Stores"*)
    - store buffer performs memory store access independently from pipeline
    - only stall pipeline for stores when store buffer is full
  - *Load Unit (LU)*
    - need to first look whether address is in store buffer then in cache
    - or need to wait until Store Buffer is empty

3.2.3 Challenges for Exploiting Instruction Level Parallelism



Challenges for Exploiting Instruction Level Parallelism: Structural Hazards

- Start instructions in EX stage when FUs are available?
- Challenge: Structural Hazards, e.g. in WB Stage

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
ADD a2,t1,t2	IF	ID	ALU	WB						
MUL a2,a0,a2		IF	ID	MUL(s1)	MUL(s2)	WB				
MUL a4,a1,a4			IF	ID	MUL(s1)	MUL(s2)	WB			
LW t1,0(a3)				IF	ID	AC	DMEM	WB		
ADDI a3,a3,4					IF	ID	ALU	WB		

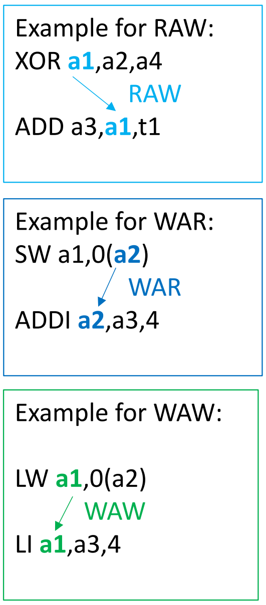
Two WB in same cycle!
WB collision!
Structural Hazard!

V1-0 ACA 32

- instructions can overtake each other due to different FU latencies
  - requires consideration of instruction dependencies during pipelined execution to preserve program order

3.2.4 Instruction Dependencies

- Read after Write (RAW)
  - result of one instruction is needed as input for another instruction
- Write after Read (WAR)
  - a value is used (read) and then updated (write)
    - the write is not allowed to overtake the read
- Write after Write (WAW)
  - a value is written and then written again
  - the second write may not overtake the first update
  - often created when registers are reused

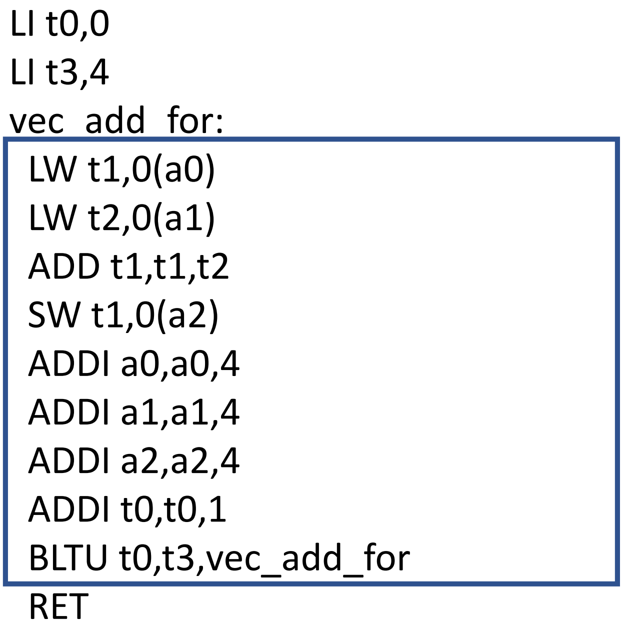


Example for RAW:
XOR a1,a2,a4
ADD a3,a1,t1

Example for WAR:
SW a1,0(a2)
ADDI a2,a3,4

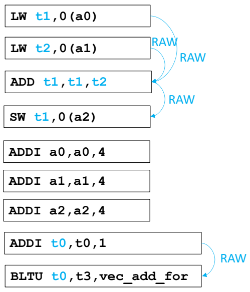
Example for WAW:
LW a1,0(a2)
LI a1,a3,4

Figure 3.18: Examples RAW, WAR, WAW



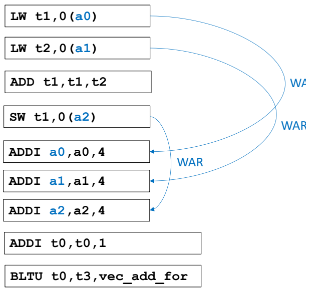
```
LI t0,0
LI t3,4
vec_add_for:
 LW t1,0(a0)
 LW t2,0(a1)
 ADD t1,t1,t2
 SW t1,0(a2)
 ADDI a0,a0,4
 ADDI a1,a1,4
 ADDI a2,a2,4
 ADDI t0,t0,1
 BLTU t0,t3,vec_add_for
 RET
```

Figure 3.19: Mark all RAW, WAR, WAW



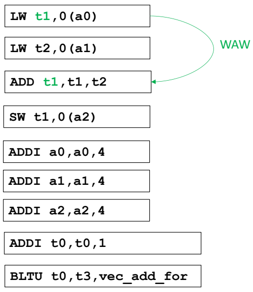
```
LW t1,0(a0)
LW t2,0(a1)
ADD t1,t1,t2
SW t1,0(a2)
ADDI a0,a0,4
ADDI a1,a1,4
ADDI a2,a2,4
ADDI t0,t0,1
BLTU t0,t3,vec_add_for
```

Figure 3.20: Marked RAW



```
LW t1,0(a0)
LW t2,0(a1)
ADD t1,t1,t2
SW t1,0(a2)
ADDI a0,a0,4
ADDI a1,a1,4
ADDI a2,a2,4
ADDI t0,t0,1
BLTU t0,t3,vec_add_for
```

Figure 3.21: Marked WAR



```
LW t1,0(a0)
LW t2,0(a1)
ADD t1,t1,t2
SW t1,0(a2)
ADDI a0,a0,4
ADDI a1,a1,4
ADDI a2,a2,4
ADDI t0,t0,1
BLTU t0,t3,vec_add_for
```

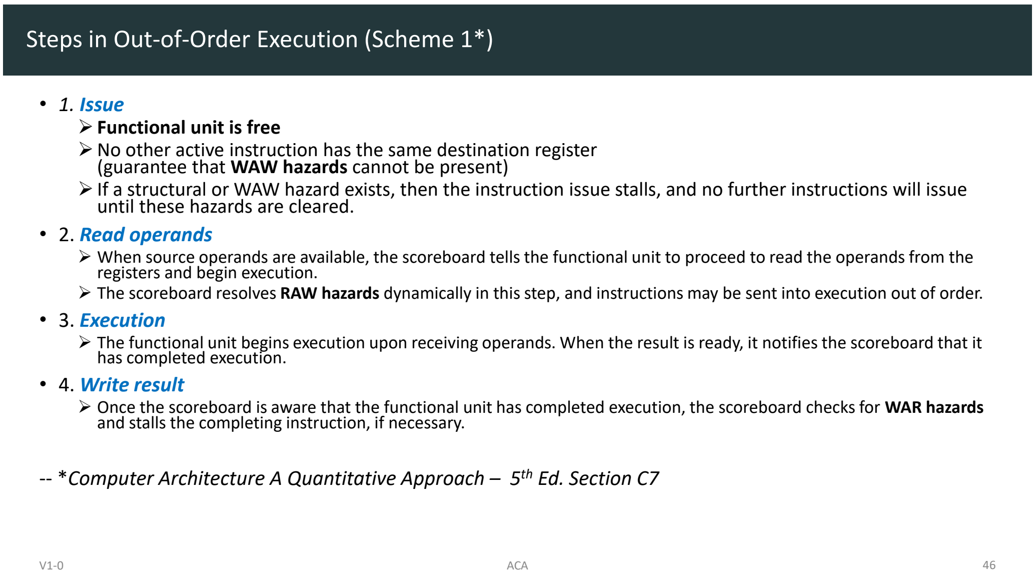
Figure 3.22: Marked WAW

Challenges with Interleaving Instruction Executing in EX Stage

1. consider RAW, WAR and WAW
2. **structural hazards** must be avoided (e.g. FU is already busy)
3. some instructions can cause **exceptions** (e.g. memory fault)

3.2.5 Out-of-Order (OoO, O3) Pipeline

- first implementation of *Scoreboard* in 1964
- to use out-of-order execution, the ID pipeline stage has to be split into two stages
  1. *Issue*: decode instructions, check for structural hazards
  2. *Read Operands*: wait until no data hazard
- in a *dynamically scheduled pipeline* all instructions pass through the issue stage in order (**in-order issue**) but they can be stalled or bypass each other in the second stage (**read operands**) and thus enter EX stage out-of-order



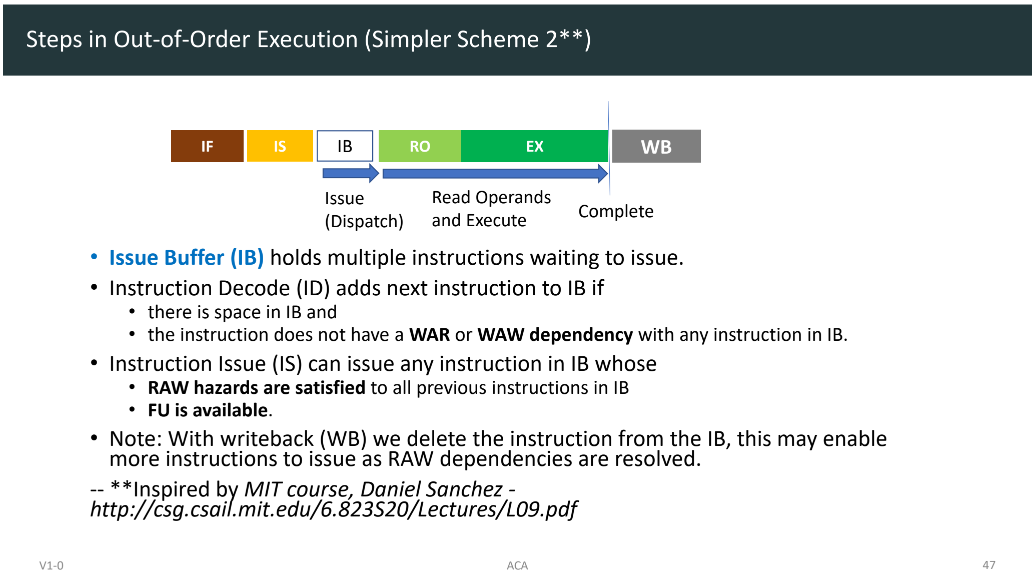
Steps in Out-of-Order Execution (Scheme 1*)

- *1. Issue*
  - **Functional unit is free**
  - No other active instruction has the same destination register (guarantee that **WAW hazards** cannot be present)
  - If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared.
- 2. *Read operands*
  - When source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution.
  - The scoreboard resolves **RAW hazards** dynamically in this step, and instructions may be sent into execution out of order.
- 3. *Execution*
  - The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution.
- 4. *Write result*
  - Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for **WAR hazards** and stalls the completing instruction, if necessary.

-- **Computer Architecture A Quantitative Approach – 5th Ed. Section C7*

V1-0 ACA 46

Figure 3.23: OoO Execution Scheme 1



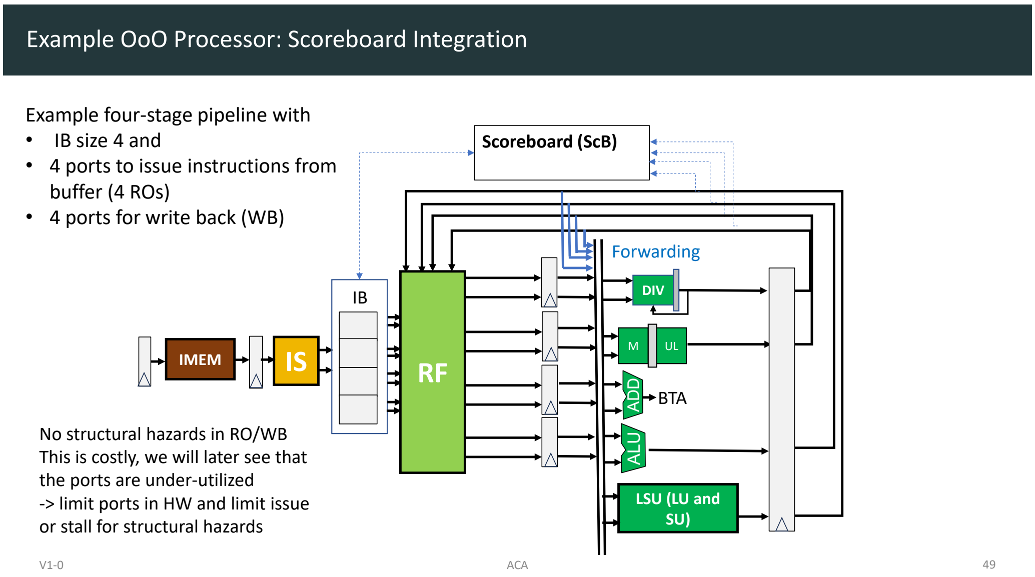
Steps in Out-of-Order Execution (Simpler Scheme 2**)

- **Issue Buffer (IB)** holds multiple instructions waiting to issue.
- Instruction Decode (ID) adds next instruction to IB if
  - there is space in IB and
  - the instruction does not have a **WAR** or **WAW dependency** with any instruction in IB.
- Instruction Issue (IS) can issue any instruction in IB whose
  - **RAW hazards are satisfied** to all previous instructions in IB
  - **FU is available**.
- Note: With writeback (WB) we delete the instruction from the IB, this may enable more instructions to issue as RAW dependencies are resolved.

-- **Inspired by *MIT course, Daniel Sanchez - http://csg.csail.mit.edu/6.823S20/Lectures/L09.pdf*

V1-0 ACA 47

Figure 3.24: OoO Execution Scheme 2



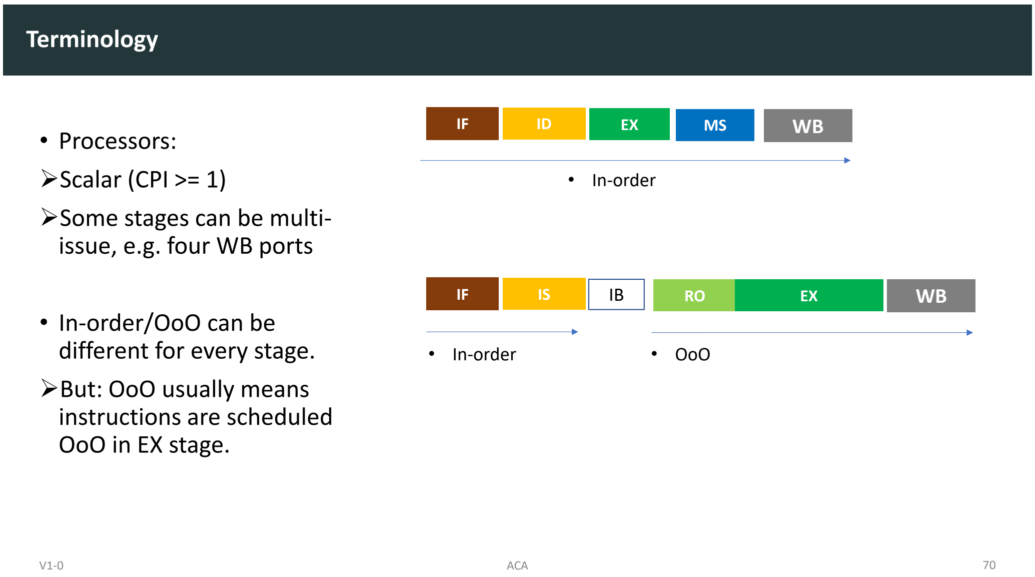
Example OoO Processor: Scoreboard Integration

Example four-stage pipeline with
- IB size 4 and
- 4 ports to issue instructions from buffer (4 ROs)
- 4 ports for write back (WB)

No structural hazards in RO/WB
This is costly, we will later see that the ports are under-utilized
-> limit ports in HW and limit issue or stall for structural hazards

V1-0 ACA 49

Figure 3.25: OoO Execution Scoreboard Integration



Terminology

- Processors:
  - ➢Scalar (CPI >= 1)
  - ➢Some stages can be multi-issue, e.g. four WB ports
- In-order/OoO can be different for every stage.
  - ➢But: OoO usually means instructions are scheduled OoO in EX stage.

V1-0 ACA 70

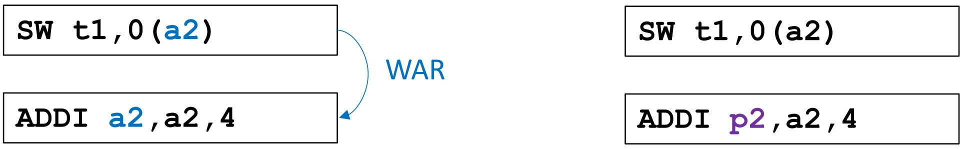
Figure 3.26: Terminology

3.2.6 Register Renaming

- WAW and WAR limit further reordering
  - no real dependencies (→ artificially added because of limitation of registers)
- register limited by ISA
- compiler optimizations limited

Approach: Register Renaming

- rename to microarchitecture register names
  - more microarchitecture registers than logical ISA registers
  - entirely eliminates WAR and WAW hazards



```
SW t1,0(a2)
ADDI a2,a2,4
```

```
SW t1,0(a2)
ADDI p2,a2,4
```

Figure 3.27: Use microarchitecture names

Example: Register Renaming removes WAW, RAW stalls

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
`LW x12,8(x9)`	IF	IS	RO	LU	LU	WB														
`LW x13,0(x7)`		IF	IS	RO	LU	LU	WB													
`DIV x17,x13,x12`			IF	IS	IB	RO	DIV	DIV	DIV	DIV	WB									
`ADDI x18,x12,28`				IF	IS	RO	ALU	WB												
`MUL x19,x12,x18`					IF	IS	RO	MUL	MUL	WB										
`MUL x20,x17,x14`						IF	IS	IB	IB	RO	MUL	MUL	WB							
`ADD x10,x20,x13`							IF	IS	IB	IB	IB	RO	ALU	WB						
`SW x10,0(x11)`								IF	stall	IS	IB	IB	RO	SU	SB					
`LW p1,4(x8)`										IF	IS	RO	LU	LU	WB					
`ADDI X13,p1,4`											IF	IS	IB	RO	ALU	WB				

10 instructions
4 cycles ramp-up (5-stage pipeline)
Total 16 cycles -4 cycles = 12 cycles

CPI = 1,2

We <u>do not</u> have to stall IF and IS on WAW and WAR, but RAW still makes instruction wait in IB for operands.
In this example the LW stores to x10 and we use an extra physical register p1 to replace x10.
Removes WAW dependency to the store.

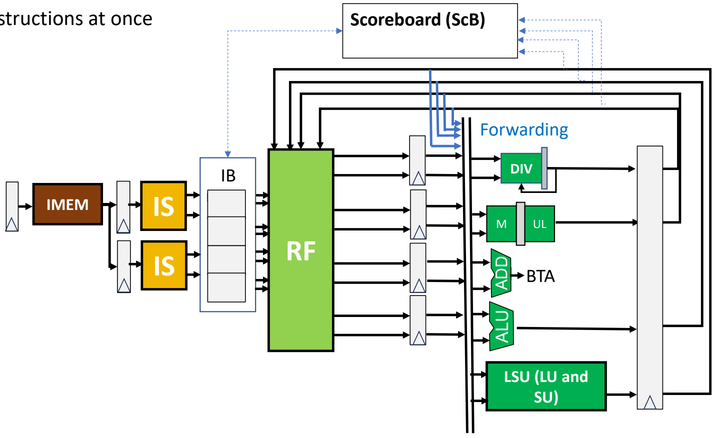
V1-0 ACA 74

Figure 3.28: Example of Register Renaming

3.2.7 Simple Superscalar Processor

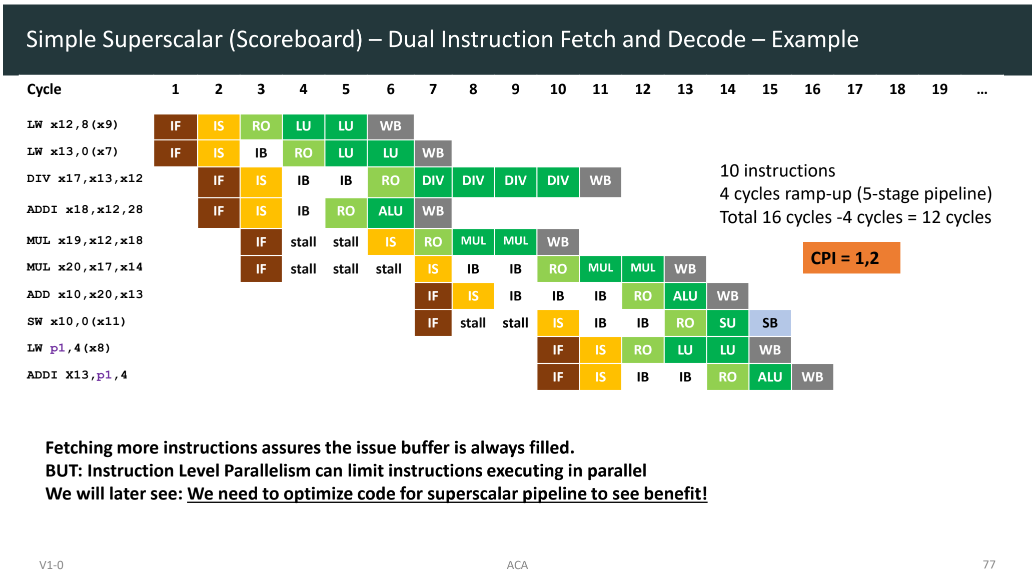
Simple Superscalar (Scoreboard) – Dual Instruction Fetch and Decode

Instruction fetch can
fetch two instructions at once
Ideal IPC = 2



V1-0 ACA 76

Figure 3.29: Dual Instruction Fetch and Decode



Simple Superscalar (Scoreboard) – Dual Instruction Fetch and Decode – Example

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	RO	LU	LU	WB														
`LW x13,0(x7)`	IF	IS	IB	RO	LU	LU	WB													
`DIV x17,x13,x12`		IF	IS	IB	IB	RO	DIV	DIV	DIV	DIV	WB									
`ADDI x18,x12,28`		IF	IS	IB	RO	ALU	WB													
`MUL x19,x12,x18`			IF	stall	stall	IS	RO	MUL	MUL	WB										
`MUL x20,x17,x14`			IF	stall	stall	stall	IS	IB	IB	RO	MUL	MUL	WB							
`ADD x10,x20,x13`							IF	IS	IB	IB	IB	RO	ALU	WB						
`SW x10,0(x11)`							IF	stall	stall	IS	IB	IB	RO	SU	SB					
`LW p1,4(x8)`										IF	IS	RO	LU	LU	WB					
`ADDI X13,p1,4`										IF	IS	IB	IB	RO	ALU	WB				

10 instructions
4 cycles ramp-up (5-stage pipeline)
Total 16 cycles -4 cycles = 12 cycles

CPI = 1,2

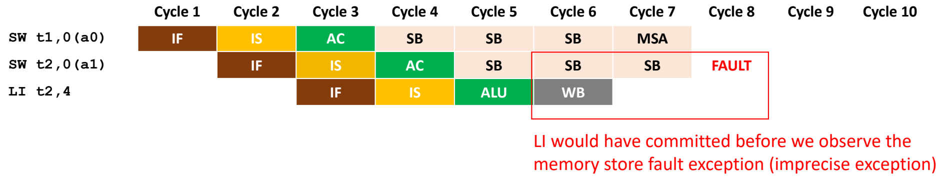
Fetching more instructions assures the issue buffer is always filled.
BUT: Instruction Level Parallelism can limit instructions executing in parallel
We will later see: We need to optimize code for superscalar pipeline to see benefit!

V1-0 ACA 77

Figure 3.30: Dual Instruction Fetch and Decode - Example

3.2.8 Reorder Buffer (ROB)

- some instructions can cause exceptions
  - memory fault
  - before entering exception handling all previous instructions should have committed
  - no instruction after the one that caused the exception should have committed



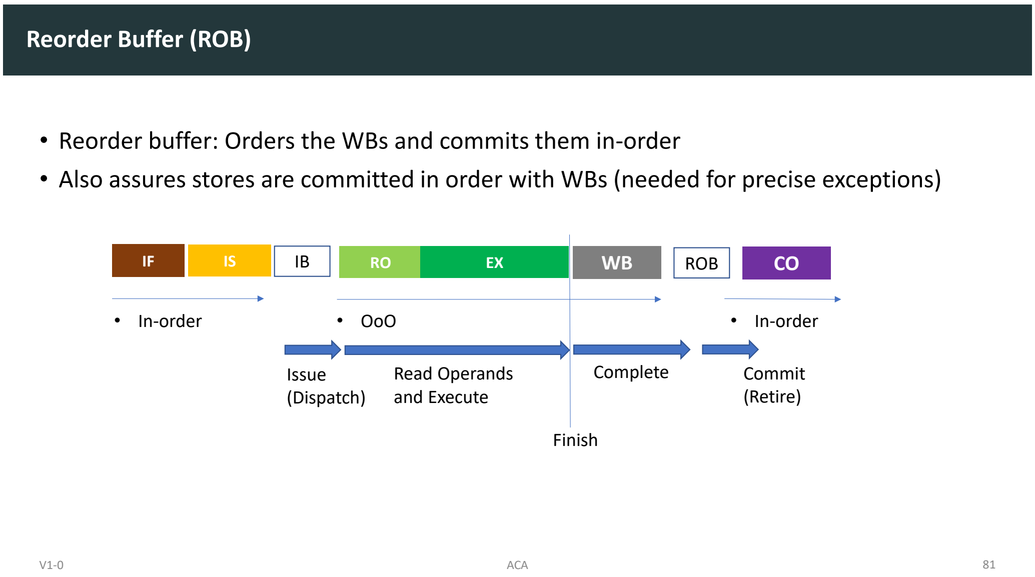
	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
`SW t1,0(a0)`	IF	IS	AC	SB	SB	SB	MSA			
`SW t2,0(a1)`		IF	IS	AC	SB	SB	SB	FAULT		
`LI t2,4`			IF	IS	ALU	WB				

LI would have committed before we observe the memory store fault exception (imprecise exception)

Figure 3.31: Pipelines and Exceptions

Implementing Precise Exceptions in OoO Pipelines

- all correct before should have committed, no of the following has committed
- scoreboard approach did not support precise exceptions
- different approach:
  - Reorder Buffer sorts all WB commits and makes sure store buffer only sends committed stores to memory

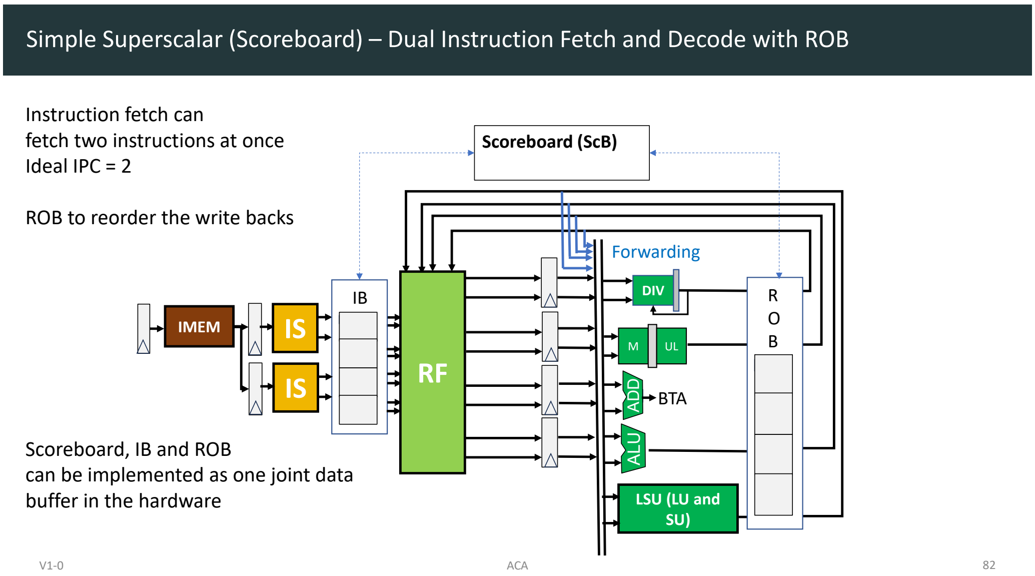


Reorder Buffer (ROB)

- Reorder buffer: Orders the WBs and commits them in-order
- Also assures stores are committed in order with WBs (needed for precise exceptions)

V1-0 ACA 81

Figure 3.32: Reorder Buffer (ROB)



Simple Superscalar (Scoreboard) – Dual Instruction Fetch and Decode with ROB

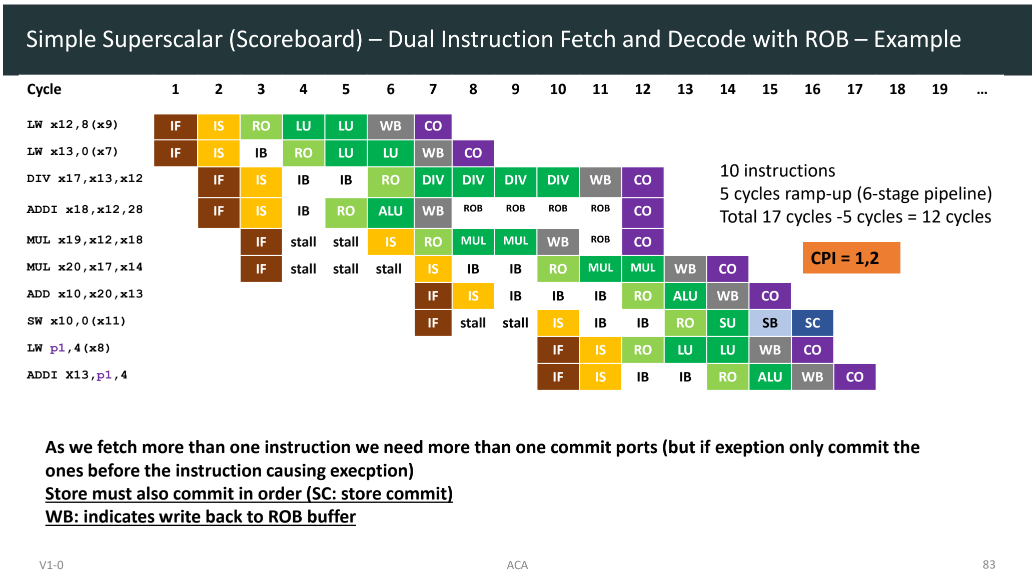
Instruction fetch can
fetch two instructions at once
Ideal IPC = 2

ROB to reorder the write backs

Scoreboard, IB and ROB
can be implemented as one joint data
buffer in the hardware

V1-0 ACA 82

Figure 3.33: Dual Instruction Fetch and Decode with ROB



Simple Superscalar (Scoreboard) – Dual Instruction Fetch and Decode with ROB – Example

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
LW x12,8(x9)	IF	IS	RO	LU	LU	WB	CO													
LW x13,0(x7)	IF	IS	IB	RO	LU	LU	WB	CO												
DIV x17,x13,x12		IF	IS	IB	IB	RO	DIV	DIV	DIV	DIV	WB	CO								
ADDI x18,x12,28		IF	IS	IB	RO	ALU	WB	ROB	ROB	ROB	ROB	CO								
MUL x19,x12,x18			IF	stall	stall	IS	RO	MUL	MUL	WB	ROB	CO								
MUL x20,x17,x14			IF	stall	stall	stall	IS	IB	IB	RO	MUL	MUL	WB	CO						
ADD x10,x20,x13							IF	IS	IB	IB	IB	RO	ALU	WB	CO					
SW x10,0(x11)							IF	stall	stall	IS	IB	IB	RO	SU	SB	SC				
LW p1,4(x8)										IF	IS	RO	LU	LU	WB	CO				
ADDI X13,p1,4										IF	IS	IB	IB	RO	ALU	WB	CO			

10 instructions
5 cycles ramp-up (6-stage pipeline)
Total 17 cycles -5 cycles = 12 cycles

CPI = 1,2

As we fetch more than one instruction we need more than one commit ports (but if exeption only commit the ones before the instruction causing execption)
Store must also commit in order (SC: store commit)
WB: indicates write back to ROB buffer

V1-0 ACA 83

Figure 3.34: Dual Instruction Fetch and Decode with ROB - Example

3.2.9 A Look at a Real Processor - CVA6

This is not relevant for the exam

3.3 Multi-Issue Multi-Threading

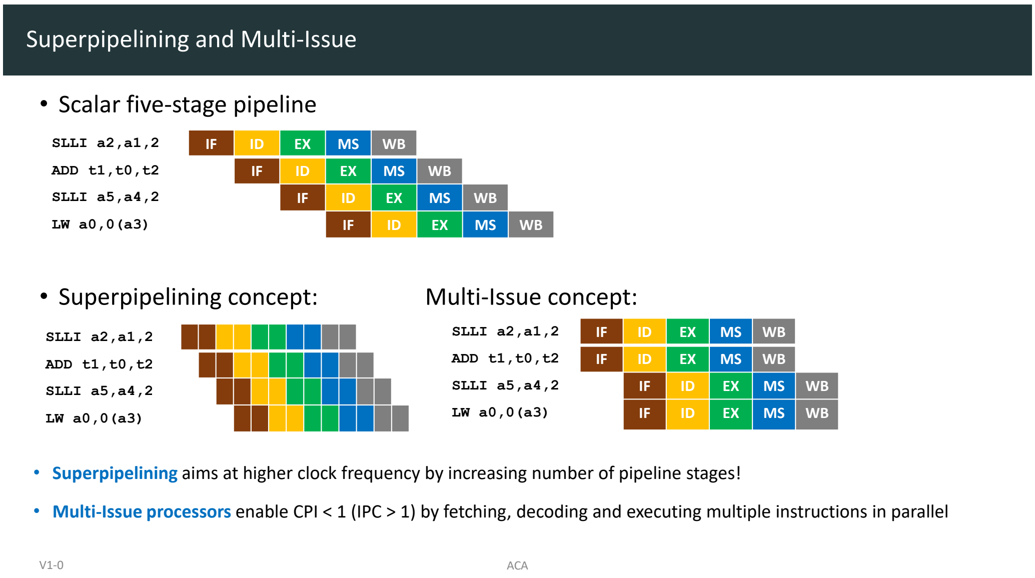
3.3.1 Increasing Processors' Performance

- recap: superscalar processor reach CPI $= 1$
- performance is defined as

$$\text{Performance} = \frac{1}{\text{IC}} \cdot \frac{\text{Instructions}}{\text{Cycle}} \cdot \frac{1}{\text{Cycle Time}} = \frac{\text{IPC} \cdot \text{Freq}}{\text{IC}} = \frac{\text{Freq}}{\text{IC} \cdot \text{CPI}} = \frac{1}{\text{runtime}}$$

- IC ... Instruction Count
- CPI ... Cycles per Instruction
- IPC... Instructions per Cycle

- superpipelining aims at increasing performance via frequency
- superscalar, VLIW aims at increasing performance via IPC
- compiler optimizations can improve instruction count and IPC



Superpipelining and Multi-Issue

- Scalar five-stage pipeline

```
SLLI a2,a1,2
ADD t1,t0,t2
SLLI a5,a4,2
LW a0,0(a3)
```

- Superpipelining concept:

```
SLLI a2,a1,2
ADD t1,t0,t2
SLLI a5,a4,2
LW a0,0(a3)
```

Multi-Issue concept:

```
SLLI a2,a1,2
ADD t1,t0,t2
SLLI a5,a4,2
LW a0,0(a3)
```

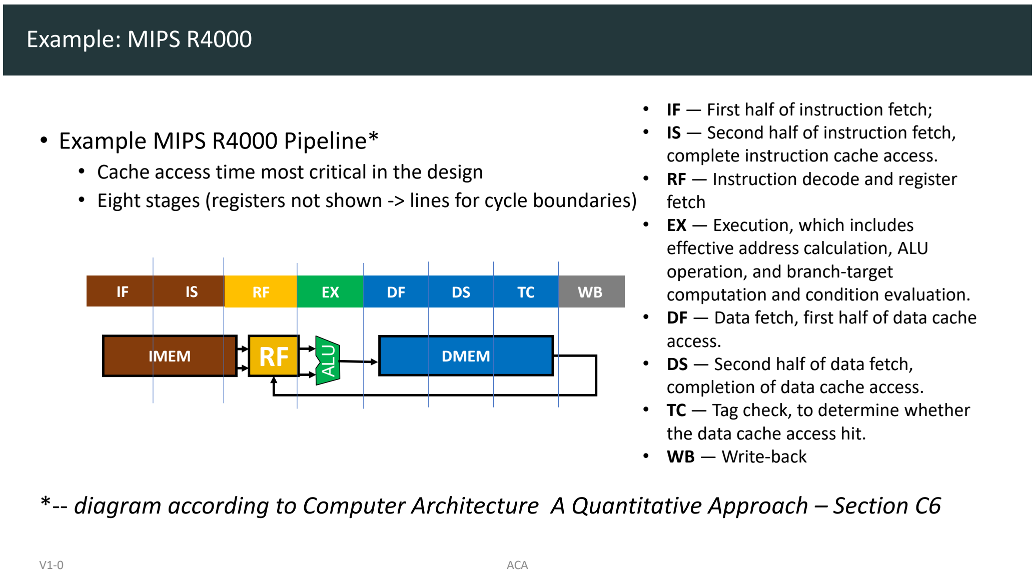
- **Superpipelining** aims at higher clock frequency by increasing number of pipeline stages!
- **Multi-Issue processors** enable CPI < 1 (IPC > 1) by fetching, decoding and executing multiple instructions in parallel

V1-0 ACA

Figure 3.35: Superpipelining and Multi-Issue

3.3.2 Superpipelining

- aims to reduce cycle time (increase clock frequency)
- *deep pipelining* or *superpipelining*: having more stages than a given baseline (e.g. five-stage pipeline)
  - pipeline stages do not need to be split evenly



Example: MIPS R4000

- Example MIPS R4000 Pipeline*
  - Cache access time most critical in the design
  - Eight stages (registers not shown -> lines for cycle boundaries)

- **IF** — First half of instruction fetch;
- **IS** — Second half of instruction fetch, complete instruction cache access.
- **RF** — Instruction decode and register fetch
- **EX** — Execution, which includes effective address calculation, ALU operation, and branch-target computation and condition evaluation.
- **DF** — Data fetch, first half of data cache access.
- **DS** — Second half of data fetch, completion of data cache access.
- **TC** — Tag check, to determine whether the data cache access hit.
- **WB** — Write-back

*-- *diagram according to Computer Architecture A Quantitative Approach – Section C6*

V1-0 ACA

Figure 3.36: Example: MIPS R4000

- instruction dependencies have higher penalties (due to deeper pipeline)
  - branch decision later available → prediction even more important as more instructions must be flushed
    - in MIPS R4000: 3 cycles branch penalty
  - forwarding can't remove all stall cycles for RAW dependencies
    - Load-use delay: 3 cycles

Limits of Superpipelining

- number of stages in Desktop: 12-20 stages
- number of stages in embedded cpus: 1-20 stages

3.3.3 Multi-Issue

Static Multiple Issue

- *at compile time*
- compiler groups instructions to be issued together in a bundle
- sorts them into *"issue slots"*
- compiler detects and avoids hazards

Dynamic Multiple Issue

- *during execution*
- CPU examines instruction stream and chooses instructions to issue each cycle
- compiler can help by reordering instructions
- CPU resolves hazards using advanced techniques at runtime

Specualtion

- *guess* what to do with an instruction
  - start operation as soon as possible
  - check wether guess was right
    - if so, complte the operation
    - if not, roll-back and to the right thing
- common to static and dynamic multiple issue
- examples
  - **Speculate on branch outcome**
    - execute instructions after branch, roll back if different path is taken
  - **Speculate on store**
    - precedes load does not refer to same address
    - can execute load instruction before the store instruction
      - roll back if the store writes the same address the load reads from

Compiler or Hardware Speculation

- compiler can *reorder* instructions
  - e.g. move load before branch
  - insert “fix-up instructions” to recover from incorrect guess
- hardware can *look ahead* for instructions to execute
  - buffers results until it determines they are actually needed
  - flush buffers on incorrect speculation

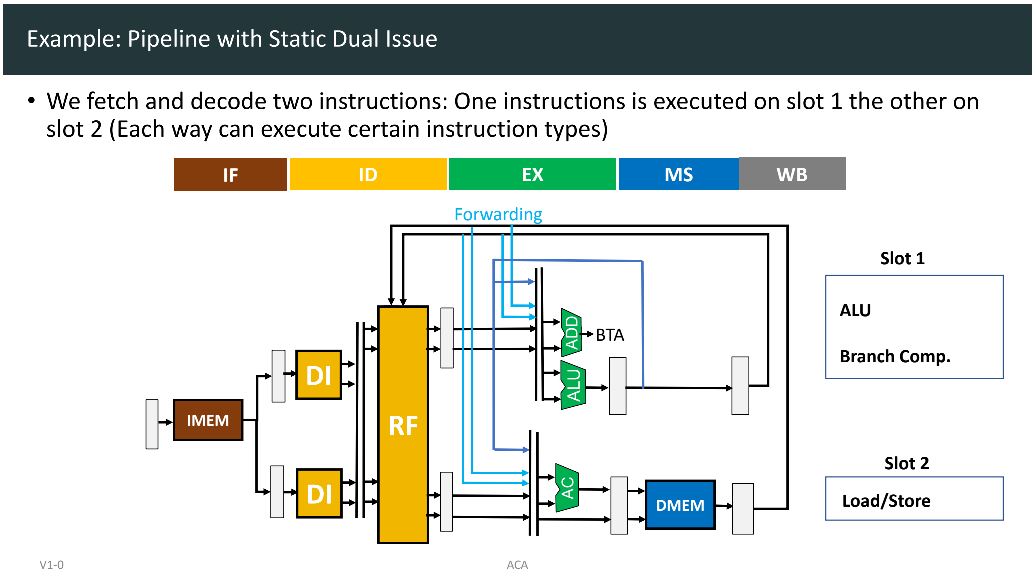
3.3.4 Very Long Instruction Word (VLIW) Static Multi-Issue

Static Multiple Issue

- compiler groups instructions into *issue packets* (aka *bundles*)
  - group of instructions that can be issued on a single cycle
  - determined by pipeline resources required
- specified multiple concurrent operations
  - ⇒ **Very Long Instruction Word (VLIW)**

Scheduling Static Multiple Issue

- compiler must remove some/all hazards
  - reorder instructions into issue packets
  - no dependencies within a packet
  - **but** if pipeline is known, all WAR dependencies are allowed if read operand happens for all instructions in a packet before write back
    - WAW and RAW must be still avoided inside a packet
- all dependencies between packets must be considered in the pipeline
  - pad with nop if necessary



Example: Pipeline with Static Dual Issue

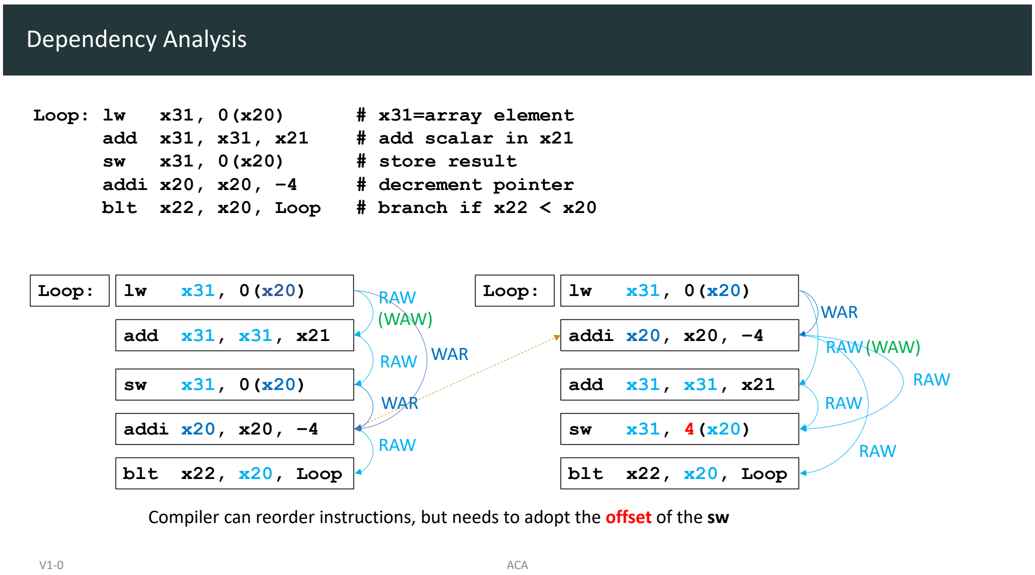
- We fetch and decode two instructions: One instructions is executed on slot 1 the other on slot 2 (Each way can execute certain instruction types)

Figure 3.37: Example: Pipeline with Static Dual Issue

Hazards in the Dual-Issue RISC-V

- more instructions executing in parallel
- RAW data hazard
  - forwarding avoided stalls with single-issue
  - now can’t use ALU result in Load/Store Unit in same packet
- Dependencies are handled as follows without register renaming:
  - RAW hazards are handled by the scoreboard. The instruction can be issued when all previous instructions with RAW dependency are at least in their finish state (last cycle of execute), hence, values are ready to be forwarded or available in the register file.

  - WAR hazards are resolved by the scoreboard. The instruction can be issued when all previous instructions with WAR dependency are at least in their RO state (cycle before execute).
  - WAW hazards are resolved by a ROB. Instructions can only be committed one cycle after all previous instructions with WAW dependency committed.
- Load-Use hazard
  - still one cycle use latency, but now two instructions



Dependency Analysis

```
Loop: lw   x31, 0(x20)       # x31=array element
      add  x31, x31, x21     # add scalar in x21
      sw   x31, 0(x20)       # store result
      addi x20, x20, -4      # decrement pointer
      blt  x22, x20, Loop    # branch if x22 < x20
```

Compiler can reorder instructions, but needs to adopt the **offset** of the **sw**

Figure 3.38: Dependency Analysis

Compiler Optimizations

- Loop Unrolling
  - replicate loop body to expose more parallelism (reduce loop-control overhead)
  - use different registers per replication
    - compiler applies *register renaming* to eliminate all data dependencies that are not true data dependencies
  - avoid loop-carried *anti-dependencies*
    - store followed by a load of the same register
    - aka *name dependencies* - reuse of a register name
  - *Unroll Factor*: number of loop body replications
  - *Fully Unrolled*: unroll factor is equal to number of iterations

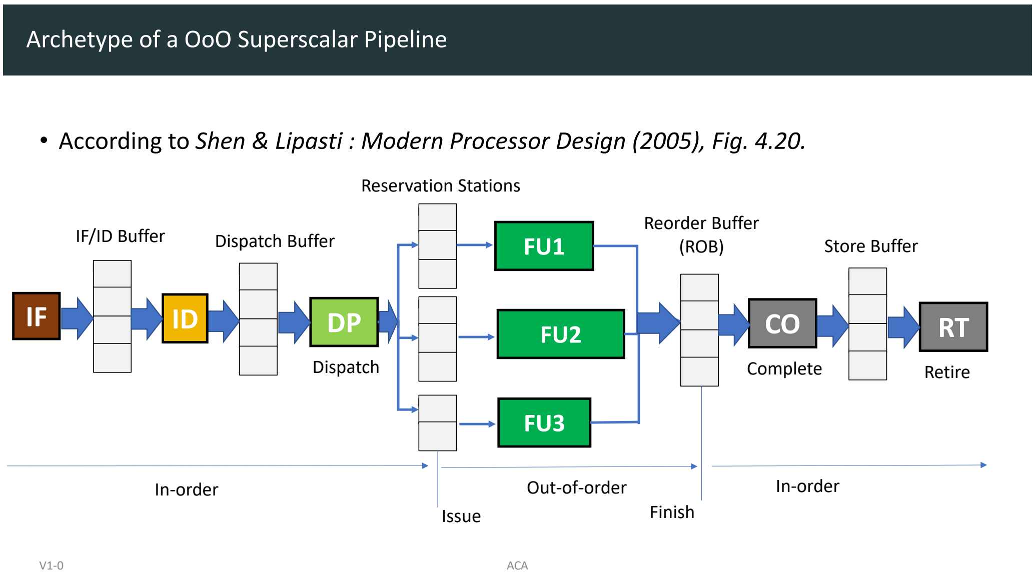
Limits of VLIW

- branches and lables break sequential instruction execution
- hard to find sufficient instruction level parallelism in single Basic Block (BB)
- Compiler Optimizations
  - loop unrolling
  - function inlining
  - SW pipelining: schedule instructions from different iterations together
  - trace scheduling & superblocks: schedule beyond basic block boundaries
- code size increases (due to loop unrolling, function inlining, ...)
- binary compatibility: if microarchitecture changes, VLIW code may not be compatible anymore

3.3.5 Superscalar Dynamic Multi-Issue

- exploits Instruction Level Parallelism
- in-order: in order issue but pipeline selects issues bundles
- out-of-order: dynamically scheduled
- Phases of Instruction Execution:

1. Fetch
2. Decode
3. Rename
4. Dispatch
5. Issue
6. Execute
7. Complete
8. Commit (Retire)



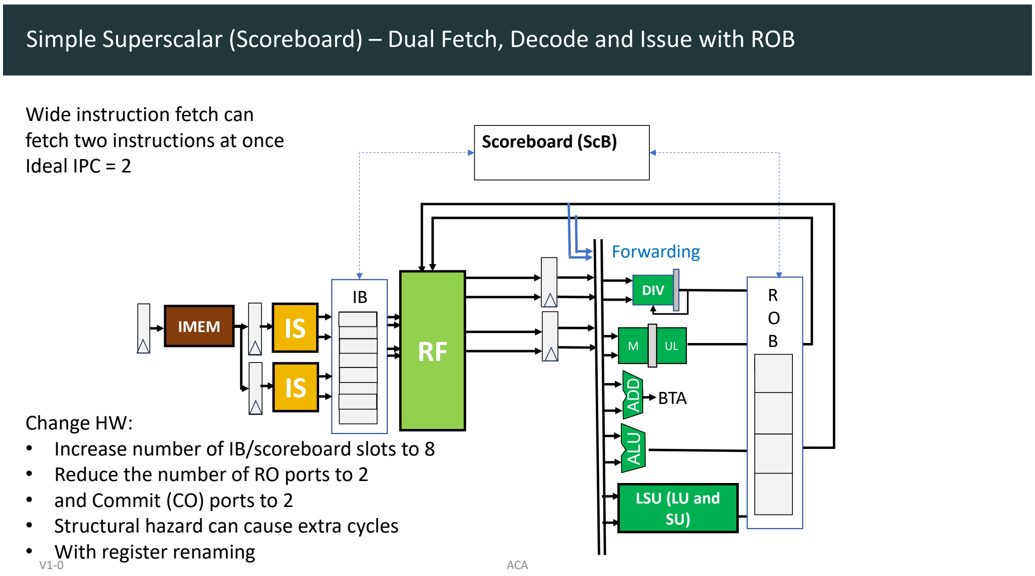
Archetype of a OoO Superscalar Pipeline

- According to *Shen & Lipasti : Modern Processor Design (2005), Fig. 4.20.*

Figure 3.39: Archetype of OoO Superscalar Pipeline

Superscalar vs. VLIW

- superscalar requires more complex hardware for instruction scheduling
- issue buffers for OoO execution
- complicated multiplexing between instruction issue structure & FU
- dependence checking logic between parallel instructions
- functional unit hazard checking
- VLIW requires a complex compiler and higher code size
- superscalars can execute pipeline-dependent code more efficiently

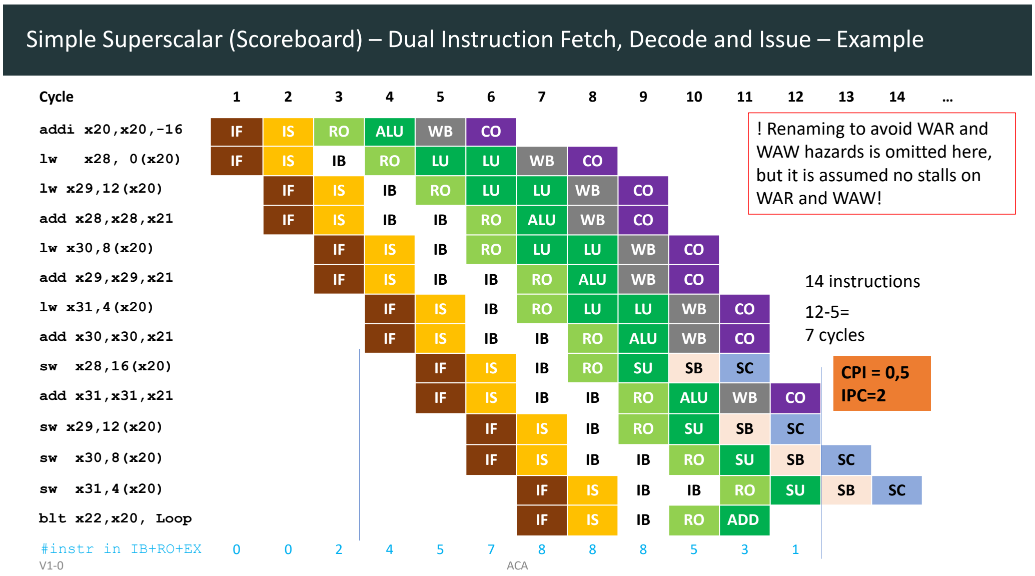


Simple Superscalar (Scoreboard) – Dual Fetch, Decode and Issue with ROB

Wide instruction fetch can
fetch two instructions at once
Ideal IPC = 2

Change HW:
- Increase number of IB/scoreboard slots to 8
- Reduce the number of RO ports to 2
- and Commit (CO) ports to 2
- Structural hazard can cause extra cycles
- With register renaming

Figure 3.40: Simple Superscalar



Simple Superscalar (Scoreboard) – Dual Instruction Fetch, Decode and Issue – Example

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
addi x20,x20,-16	IF	IS	RO	ALU	WB	CO									
lw x28, 0(x20)	IF	IS	IB	RO	LU	LU	WB	CO							
lw x29,12(x20)		IF	IS	IB	RO	LU	LU	WB	CO						
add x28,x28,x21		IF	IS	IB	IB	RO	ALU	WB	CO						
lw x30,8(x20)			IF	IS	IB	RO	LU	LU	WB	CO					
add x29,x29,x21			IF	IS	IB	IB	RO	ALU	WB	CO					
lw x31,4(x20)				IF	IS	IB	RO	LU	LU	WB	CO				
add x30,x30,x21				IF	IS	IB	IB	RO	ALU	WB	CO				
sw x28,16(x20)					IF	IS	IB	RO	SU	SB	SC				
add x31,x31,x21					IF	IS	IB	IB	RO	ALU	WB	CO			
sw x29,12(x20)						IF	IS	IB	RO	SU	SB	SC			
sw x30,8(x20)						IF	IS	IB	IB	RO	SU	SB	SC		
sw x31,4(x20)							IF	IS	IB	IB	RO	SU	SB	SC	
blt x22,x20, Loop							IF	IS	IB	RO	ADD				
#instr in IB+RO+EX	0	0	2	4	5	7	8	8	8	5	3	1			

! Renaming to avoid WAR and WAW hazards is omitted here, but it is assumed no stalls on WAR and WAW!

14 instructions

12-5=
7 cycles

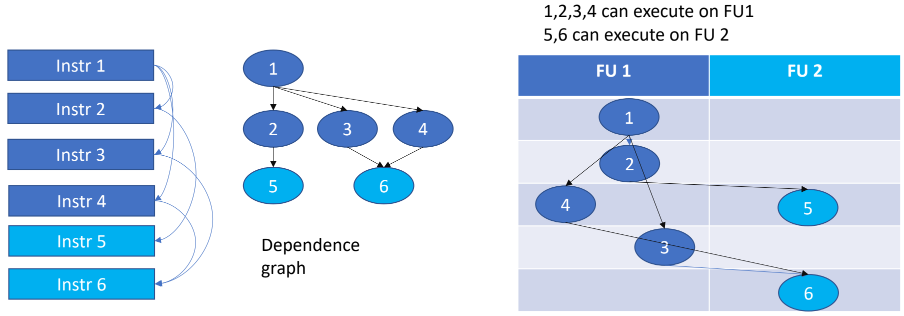
CPI = 0,5
IPC=2

V1-0 ACA

Figure 3.41: Simple Superscalar - Example

Instruction Scheduling for Superscalar

- The process of mapping a series of instructions into execution resources
- Decides when and where an instruction is executed



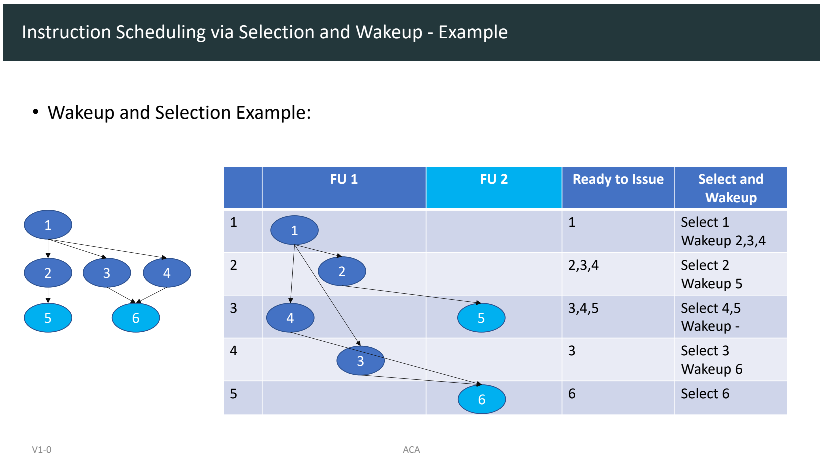
Derived from CA course of Mikko Lipasti-University of Wisconsin

V1-0 ACA

Figure 3.42: Instruction Scheduling for Superscalar

Instruction Scheduling via Selection and Wakeup

- A set of wakeup and select operations
- Wakeup
  - Broadcasts the tags of parent instructions selected
  - Dependent instruction gets matching tags, determines if source operands are ready
  - Resolves RAW data dependencies
- Select
  - Picks instructions to issue among a pool of ready instructions
  - Resolves resource conflicts
  - Issue bandwidth
  - Limited number of functional units / memory ports



Instruction Scheduling via Selection and Wakeup - Example

- Wakeup and Selection Example:

	FU 1	FU 2	Ready to Issue	Select and Wakeup
1	1		1	Select 1 Wakeup 2,3,4
2	2		2,3,4	Select 2 Wakeup 5
3	4	5	3,4,5	Select 4,5 Wakeup -
4	3		3	Select 3 Wakeup 6
5		6	6	Select 6

V1-0 ACA

Figure 3.43: Instruction Scheduling via Selection and Wakeup - Example

3.3.6 Hardware Multi-Threading

- Thread
  - has state and a current program counter
  - shares the address space of a single process, allowing a thread to easily access data of other threads within the same process.
- Multithreading
  - multiple threads share a processor without requiring an intervening process switch.
  - The ability to switch between threads rapidly is what enables multithreading to be used to hide pipeline and memory latencies.
  - Exploiting Thread-Level Parallelsim (TLP) to improve uniprocessor throughput (IPC)

Thread-Level Parallelsim (TLP)

- Multithreading (MT) targets to exploit thread-level parallelism (TLP)
- MT allows multiple threads to share the FUs of a single processor
- MT does not duplicate the entire processor, duplicating only private state, such as the registers and PC.
- A more general method to exploit TLP is to use a multi-core processor that can execute multiple independent threads in parallel.
- Many recent compute platforms incorporate multi-core processors, for which each single core additionally provides multithreading support.

Fine-Grained vs. Coarse-Grained MT

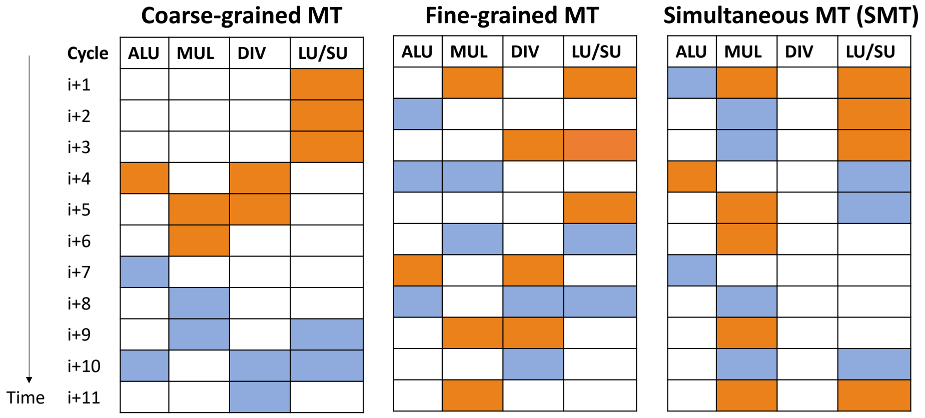
- **Fine-grained multithreading**
  - switches between threads on each clock cycle,
  - execution of instructions from multiple threads to be interleaved. (often round-robin skipping stalled threads)
  - Advantage: hide the throughput losses that arise from both short and long stalls because instructions from other threads can be executed when one thread stalls, even if the stall is only for a few cycles.
  - Disadvantage: slows down the execution of an individual thread because a thread that is ready to execute without stalls will be delayed by instructions from other threads.
- **Coarse-grained multithreading**
  - switches threads only on costly stalls, such as level two or three cache misses.
  - Advantage: less likely to slow down the execution of any one thread
  - Disadvantage: it is limited in its ability to overcome throughput losses, especially from shorter stalls.

Simultaneous Multithreading (SMT)

- dynamically scheduled (OoO) processors already have many of the hardware mechanisms needed to support SMT
- Multithreading can be built on top of an out-of-order processor by adding

  - separate PCs and register files, and
  - the capability for instructions from multiple threads to commit.
- Instructions from different threads can be issued in same cycle.

Patterns for Types of Multithreading (MT)



	Coarse-grained MT				Fine-grained MT				Simultaneous MT (SMT)			
Cycle	ALU	MUL	DIV	LU/SU	ALU	MUL	DIV	LU/SU	ALU	MUL	DIV	LU/SU
i+1				🟧		🟧		🟧	🟦	🟧		🟧
i+2				🟧	🟦					🟦		🟧
i+3				🟧			🟧	🟧		🟦		🟧
i+4	🟧		🟧		🟦	🟦			🟧			🟦
i+5		🟧	🟧					🟧		🟧		🟦
i+6		🟧				🟦		🟦		🟧		
i+7	🟦				🟧		🟧		🟦			
i+8		🟦			🟦		🟦	🟦		🟦		
i+9		🟦		🟦		🟧	🟧			🟧		
i+10	🟦		🟦	🟦			🟦			🟦		🟦
i+11			🟦			🟧				🟧		🟧

V1-0 ACA

Figure 3.44: Patterns for Types of Multithreading

3.3.7 A Look at Real Processors - A15 & BOOM

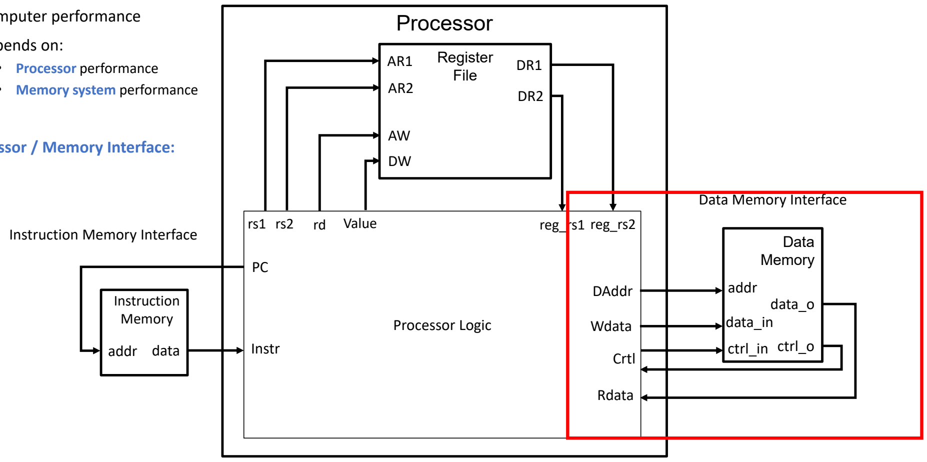
This is not relevant for the exam

3.4 Caches and Memory

Introduction

- Computer performance
- depends on:
  - Processor performance
  - Memory system performance

Processor / Memory Interface:

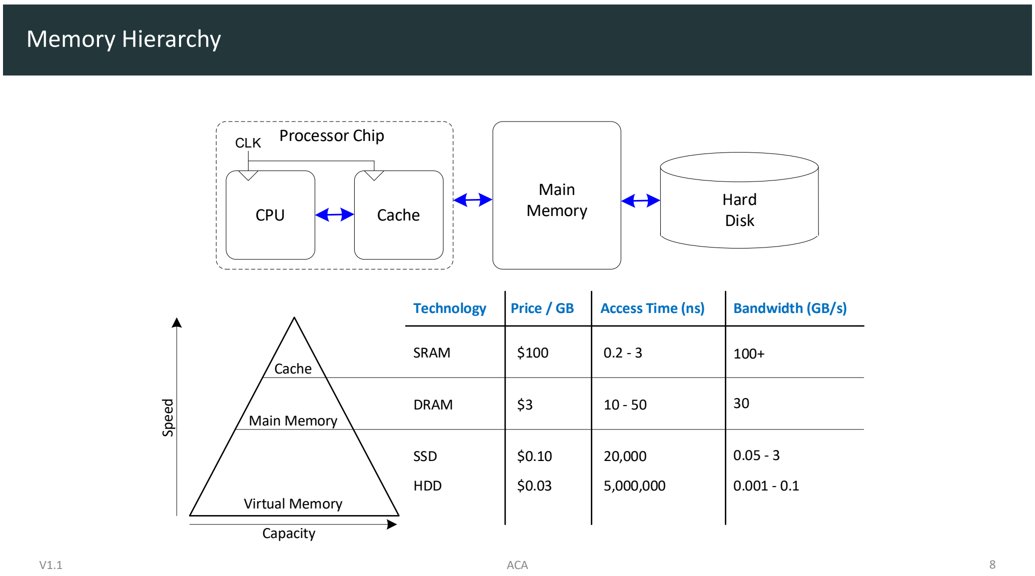


V1.1 ACA 5

Figure 3.45: Introduction to Caches

- in previous chapters it was assumed that memory access takes 1 clock cycles
  - this hasn't been true since 1980s
- Memory System Challenges
  - make memory system appear as fast as processor
  - use hierarchy of memories

- *ideal memory*
  - fast
  - cheap
  - large



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$100	0.2 - 3	100+
DRAM	$3	10 - 50	30
SSD	$0.10	20,000	0.05 - 3
HDD	$0.03	5,000,000	0.001 - 0.1

Figure 3.46: Memory Hierarchy

Locality

- exploit locality to make memory fast
- **temporal locality**
  - *locality of time*
  - if data used recently, likely to use it again soon
- **spatial locality**
  - *locality of space*
  - if data used recently, likely to use nearby data soon

3.4.1 Memory Performance

- **Hit**: data found in that level of memory hierarchy
- **Miss**: data not found
- **Average memory access time (*AMAT*)**: average time for processor to acces data

$$\text{Hit Rate} = \frac{\#\text{ hits}}{\#\text{ memory accesses}} = 1 - \text{Miss Rate} \qquad \text{Miss Rate} = \frac{\#\text{ misses}}{\#\text{ memory accesses}} = 1 - \text{Hit Rate}$$

$$\text{AMAT} = t_{\text{cache}} + \text{MR}_{\text{cache}}\left[t_{\text{MM}} + \text{MR}_{\text{MM}(t_{\text{VM}})}\right]$$

✍ **Example**

- A Program has 2000 loads and stores
- 1250 of these data values in cache
- rest supplied by other levels of memory hierarchy

What are the hit and miss rates?

$$\text{Hit Rate} = \frac{1250}{200} = 0.625$$

$$\text{Miss Rate} = \frac{750}{2000} = 0.375$$

Example

- Suppose processor has 2 levels of hierarchy: cache and main memory
- $t_{\text{cache}} = 1$ cycle, $t_{\text{MM}} = 100$ cycles

What is the AMAT of the example 1?

$$\text{AMAT} = t_{\text{cache}} + \text{MR}_{\text{cache}}\left[t_{\text{MM}} + \text{MR}_{\text{MM}(t_{\text{VM}})}\right]$$

Tip

$\text{MR}_{\text{MM}} = 0$ because it has all the data

$$\text{AMAT} = t_{\text{cache}} + \text{MR}_{\text{cache}}(t_{\text{MM}}) = [1 + 0.375 \cdot 100] \text{ cycles} = 38.5 \text{ cycles}$$

3.4.2 Caches

- highest level in memory hierarchy
- fast (typically ≈ 1 cycle access time)
- ideally supplies most data to processor
- usually holds most recently accessed data

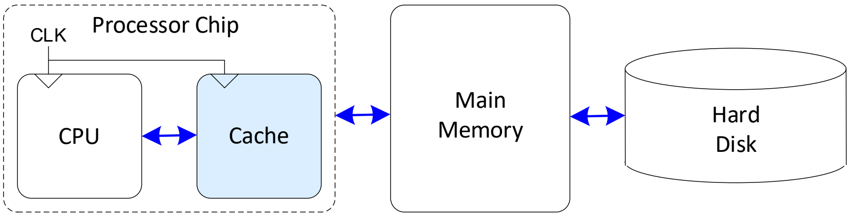


Figure 3.47: Structure of Cache and CPU

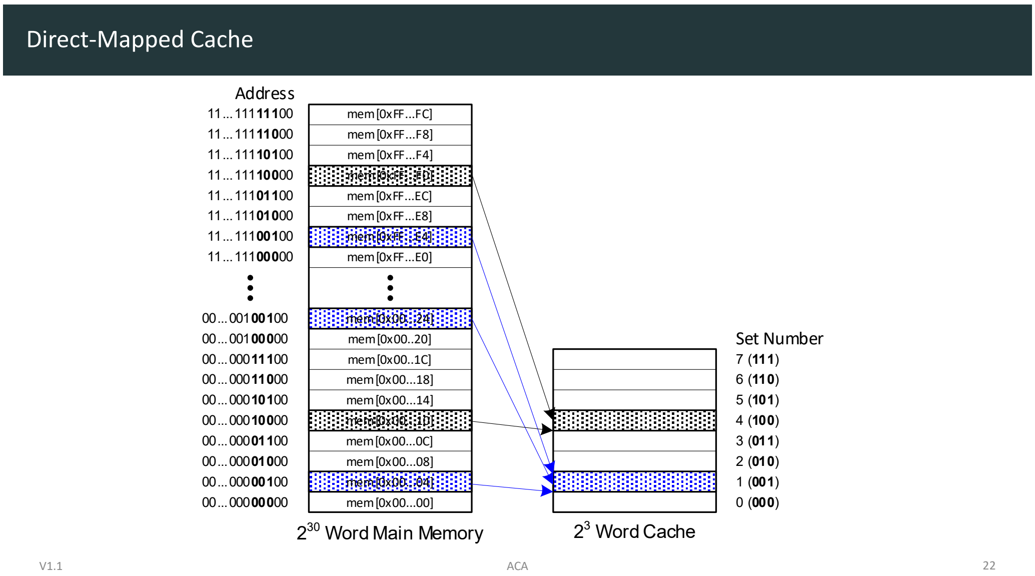
Cache Design Principles

- what data is held in the cache?
  - ideally, cache anticipates needed data and puts it in cache
  - but impossible to predict future
    - use past to predict the future (temporal and spatial locality)
- how is data found?
  - cache organized into S sets
  - each memory address maps to exactly one set
  - caches categorized by # of blocks in a set:
    - *direct mapped*: 1 block per set
    - *N-way set associative*: N blocks per set
    - *fully associative*: all cache blocks in 1 set
- what data is replaced?

Cache Terminology

- **Capacity C**
  - number of data bytes in cache
- **Block Size B**
  - bytes of data brought into cache at once
- **Degree of associativity N**
  - number of blocks in a set
- **Number of sets S**
  - each memory address maps to exactly one cache set
  - $S = \frac{B}{N}$

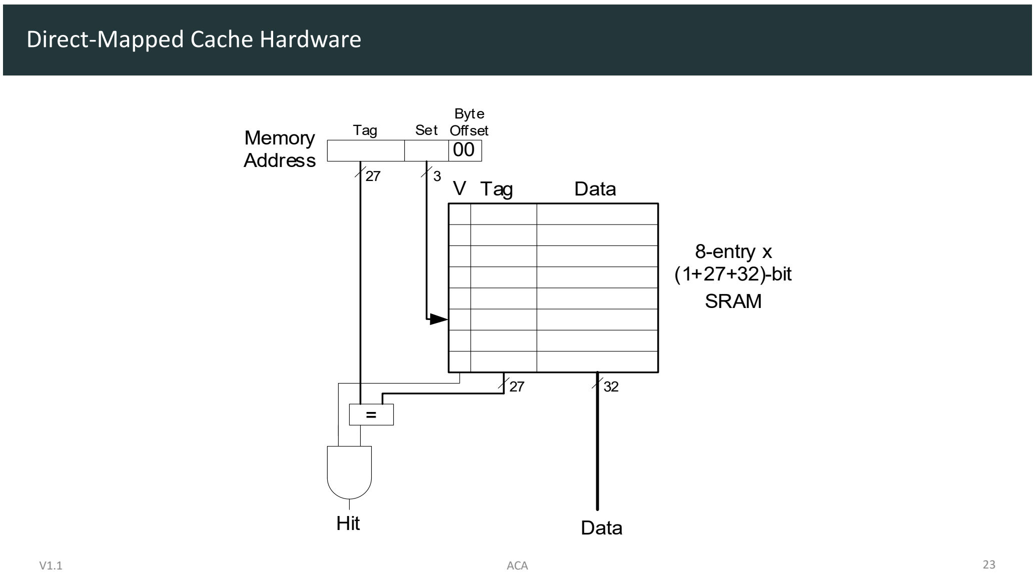
3.4.3 Direct-Mapped Caches



Direct-Mapped Cache

V1.1 ACA 22

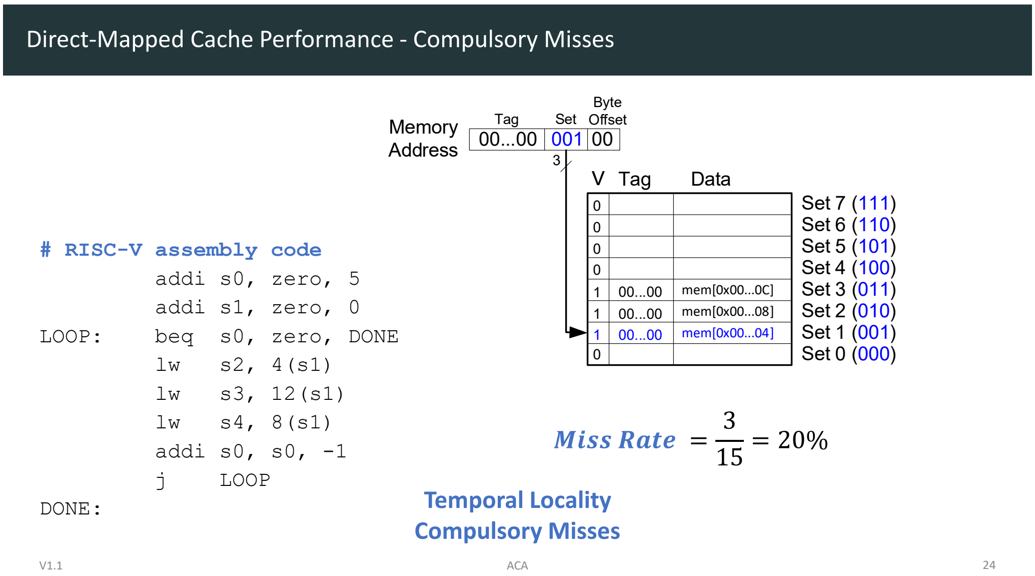
Figure 3.48: Direct-Mapped Cache



Direct-Mapped Cache Hardware

V1.1 ACA 23

Figure 3.49: Direct-Mapped Cache Hardware



Direct-Mapped Cache Performance - Compulsory Misses

V	Tag	Data	
0			Set 7 (111)
0			Set 6 (110)
0			Set 5 (101)
0			Set 4 (100)
1	00...00	mem[0x00...0C]	Set 3 (011)
1	00...00	mem[0x00...08]	Set 2 (010)
1	00...00	mem[0x00...04]	Set 1 (001)
0			Set 0 (000)

```
# RISC-V assembly code
        addi s0, zero, 5
        addi s1, zero, 0
LOOP:   beq  s0, zero, DONE
        lw   s2, 4(s1)
        lw   s3, 12(s1)
        lw   s4, 8(s1)
        addi s0, s0, -1
        j    LOOP
DONE:
```

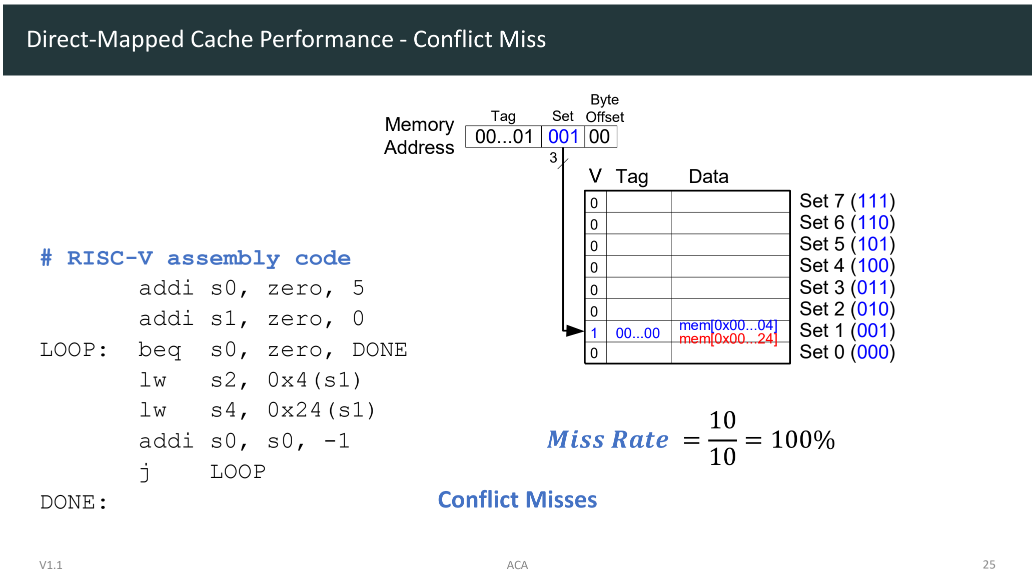
$$\textit{Miss Rate} = \frac{3}{15} = 20\%$$

Temporal Locality
Compulsory Misses

V1.1 ACA 24

Figure 3.50: Direct-Mapped Cache Performance - Compulsory Misses

- compulsory misses are misses if the cache is empty, so it has to get the data from the memory



Direct-Mapped Cache Performance - Conflict Miss

V	Tag	Data	
0			Set 7 (111)
0			Set 6 (110)
0			Set 5 (101)
0			Set 4 (100)
0			Set 3 (011)
0			Set 2 (010)
1	00...00	mem[0x00...04] mem[0x00...24]	Set 1 (001)
0			Set 0 (000)

```
# RISC-V assembly code
        addi s0, zero, 5
        addi s1, zero, 0
LOOP:   beq  s0, zero, DONE
        lw   s2, 0x4(s1)
        lw   s4, 0x24(s1)
        addi s0, s0, -1
        j    LOOP
DONE:
```

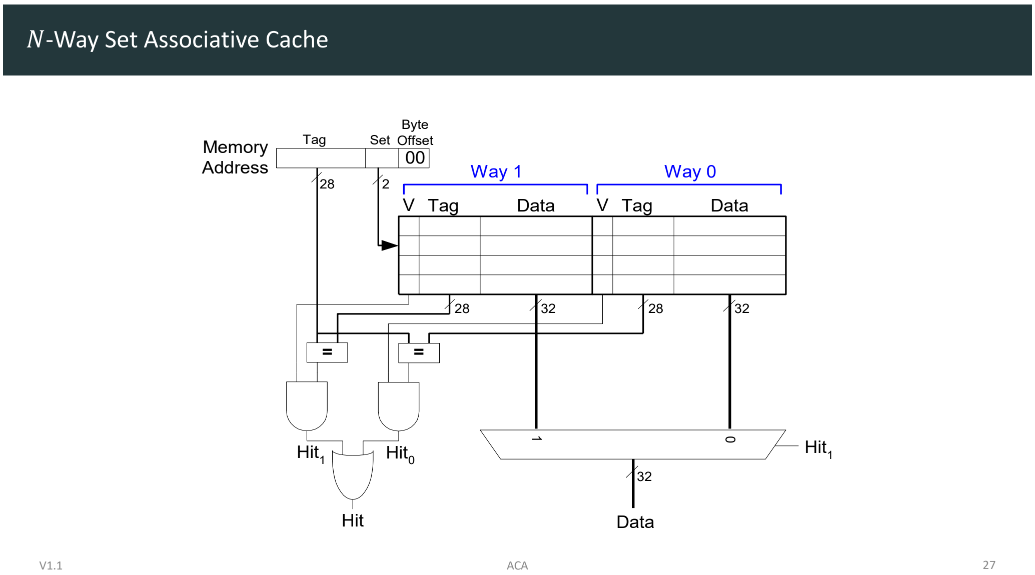
$$\textit{Miss Rate} = \frac{10}{10} = 100\%$$

Conflict Misses

V1.1 ACA 25

Figure 3.51: Direct-Mapped Cache Performance - Conflict Misses

3.4.4 Associative Caches



N-Way Set Associative Cache

V1.1 ACA 27

Figure 3.52: N-Way Set Associative Cache

- N-Way set associative cache reduce reduce conflict misses but are more expensive to build

3.4.5 Spatial Locality

- caches with larger block size, use the spatial locality better, because they load more neighbour data

Cache Organization Recap

Organization	Number of Ways (N)	Number of Sets (S)
Direct Mapped	1	B
N-Way Set Associative	$1 < N < B$	$\frac{B}{N}$
Fully Associative	B	1

Table 3.4: Cache Organization Recap

3.4.6 Cache Replacement Policy

- **compulsory**: first time data accessed
- **capacity**: cache too small to hold all data of interest
- **conflict**: data of interest maps to same location in cache which is currently used
- **Miss penalty**: time it takes to retrieve a block from lower level of hierarchy
- if cache is full: program access data X and evicts data Y
  - **capacity miss** if access Y again
- how to choose Yto minimize chance of needing it again?
  - *Least recently used (LRU) replacement*

Multilevel Caches

- larger caches have lower miss rates, longer access times
- expand memory hierarchy to multiple levels of caches
- *Level 1*: small and fast (e.g. 16 KB, 1 cycle)
- *Level 2*: larger and slower (e.g. 256 KB, 2-6 cycles)
- (most modern PCs have L1, L2 and L3 cache)

3.4.7 Cache Miss/Hit Strategy

Hit & Miss on reading/load

- *load hit*: valid bit is set and tag maches ⇒ data is found in cache
- *load miss*: data is not found in cache, pipeline needs to be stalled, slower memory must deliver the data

Hit & Mis on writing/store

- *write hit*
  - write-through
    - updates the cache **and** the main memory immediately
    - 👍 simple
    - 👍 data consistency with main memory guaranteed
    - 👎 frequent access to the main memory
    - 👎 loss of performance
  - copy-back (aka write-back)
    - refresh the cache and mark the block as *dirty*
      - only update the main memory later when the block is removed from cache
    - 👍 write hit is much faster
    - 👍 less frequent accesses to the main memory
    - 👎 data inconsistency with the main memory
    - 👎 read miss is slower (due to copy-back)
  - write- buffer
    - for data consistency and fast write operations
    - new value is entered in the cache and second fast cache
    - processor can continue with further processing
    - if buffer is full, processor must wait
- *write miss*
  - write-around
    - ignore the cache and write directly to memory
    - mostly in combination with Write-Through
  - fetch-on-write
    - replace the current content of the cache and update the tag
    - if block size > 1, load the remaining data belonging to the block from the main memory after
    - read access to the main memory and the write hit depending on the strategy
    - most frequently used method

3.5 Vector Processors

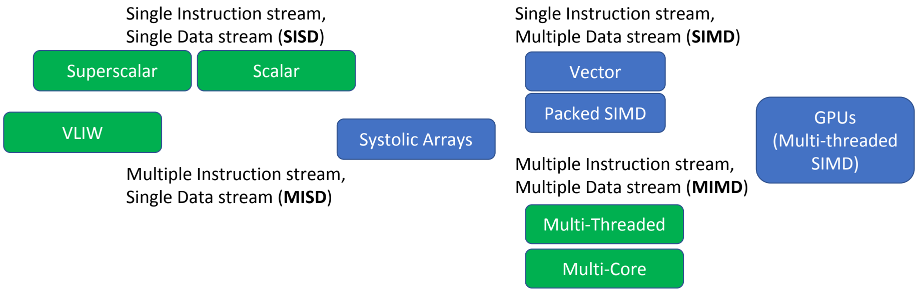
- System-on-Chip (SoC)s are often multi-core systems
- general-purpose SoC may have many replicates of general-purpose processors (e.g. many ARM or standard RISC-V cores)
- to improve energy-efficiency many SoC use specialized cores (*heterogeneity*)

Types of Specialized Cores

- Vector Processors
  - introduced in the 70s (*Cray*)
  - got new attention recently especially due to machine learning workloads
- GPUs
  - were initially introduced for redering graphics in real time (video games)
  - *General Purpose GPU (GP-GPU)*: programming language such as CUDO from NVIDIA allowed to use GPUs for other computations beside rendering
- HW Accelerators
  - processing cores that are specialized for a certain task (with very limited programmability)
  - usually faster and more energy efficient than software running on programmable core
  - different types:

  - deep learning: Tensor Processing Units / Neural Processing Units
  - security: encryption and decryption
  - Video En/Decoders
- Application-specific Instruction Set Processors (ASIPs)
  - between general-purpose programmable cores and accelerators
  - some programmability but tailored towards a certain application
  - example: Audio/Video Digital Signal Processors (DSPs)

3.5.1 Flynn's Taxonomy



Single Instruction stream, Single Data stream (**SISD**)

Superscalar

Scalar

VLIW

Single Instruction stream, Multiple Data stream (**SIMD**)

Vector

Packed SIMD

GPUs (Multi-threaded SIMD)

Systolic Arrays

Multiple Instruction stream, Single Data stream (**MISD**)

Multiple Instruction stream, Multiple Data stream (**MIMD**)

Multi-Threaded

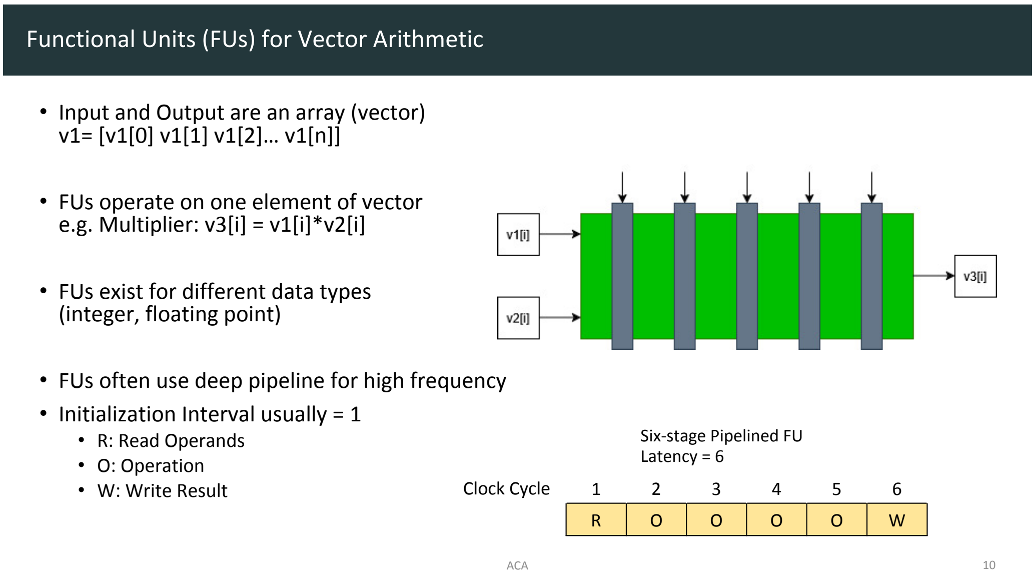
Multi-Core

Figure 3.53: Classification of Computing Cores

3.5.2 Vector Units

Vector Instruction Sets

- **one instruction** operates on **several data values** (SIMD)
- the data values are independent
- operation use the same type of functional unit for all data
- data values are store in separate registers
- data values are aranged in uniform structure (vector)
- load/store access
  - a continuous range of memory
  - use a regular pattern (strided access)
- one instruction stream for parallel pipelines (so called *lanes*)



Functional Units (FUs) for Vector Arithmetic

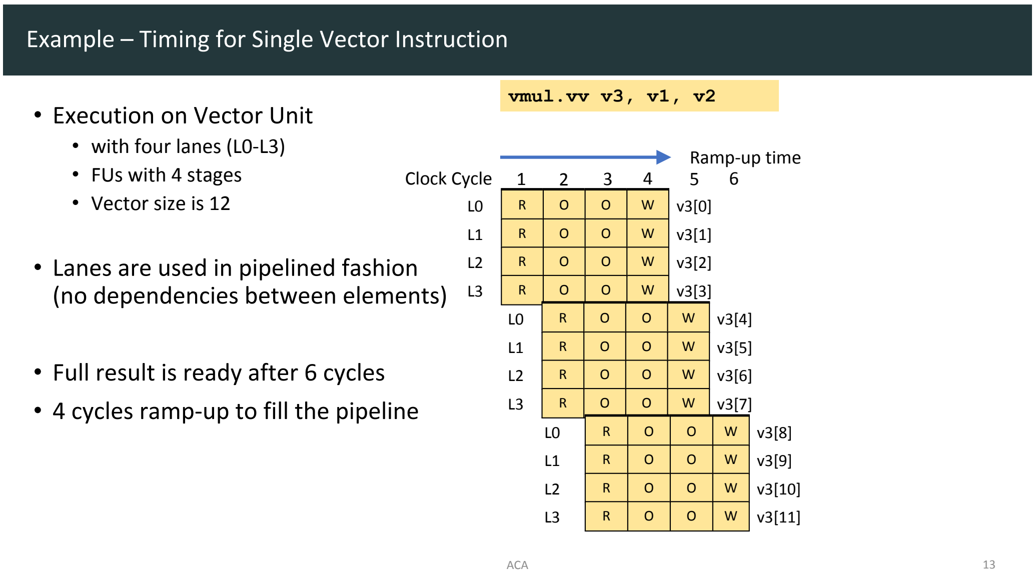
- Input and Output are an array (vector)
  v1= [v1[0] v1[1] v1[2]… v1[n]]
- FUs operate on one element of vector
  e.g. Multiplier: v3[i] = v1[i]*v2[i]
- FUs exist for different data types
  (integer, floating point)
- FUs often use deep pipeline for high frequency
- Initialization Interval usually = 1
  - R: Read Operands
  - O: Operation
  - W: Write Result

Six-stage Pipelined FU
Latency = 6

Clock Cycle	1	2	3	4	5	6
	R	O	O	O	O	W

ACA 10

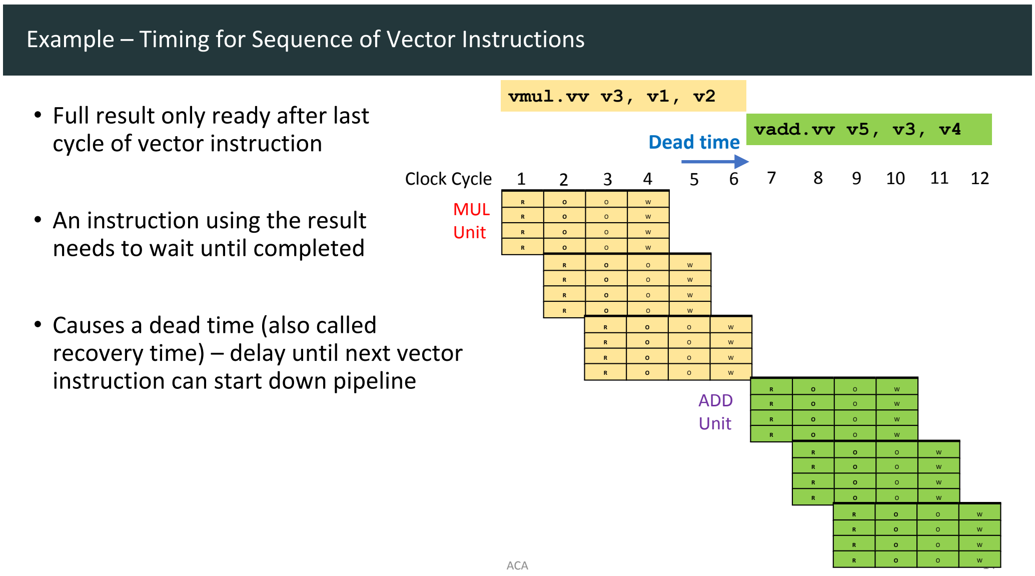
Figure 3.54: FU for Vector Arithmetic



Example – Timing for Single Vector Instruction

- Execution on Vector Unit
  - with four lanes (L0-L3)
  - FUs with 4 stages
  - Vector size is 12
- Lanes are used in pipelined fashion (no dependencies between elements)
- Full result is ready after 6 cycles
- 4 cycles ramp-up to fill the pipeline

Figure 3.55: Example - Timing for Single Vector Instruction



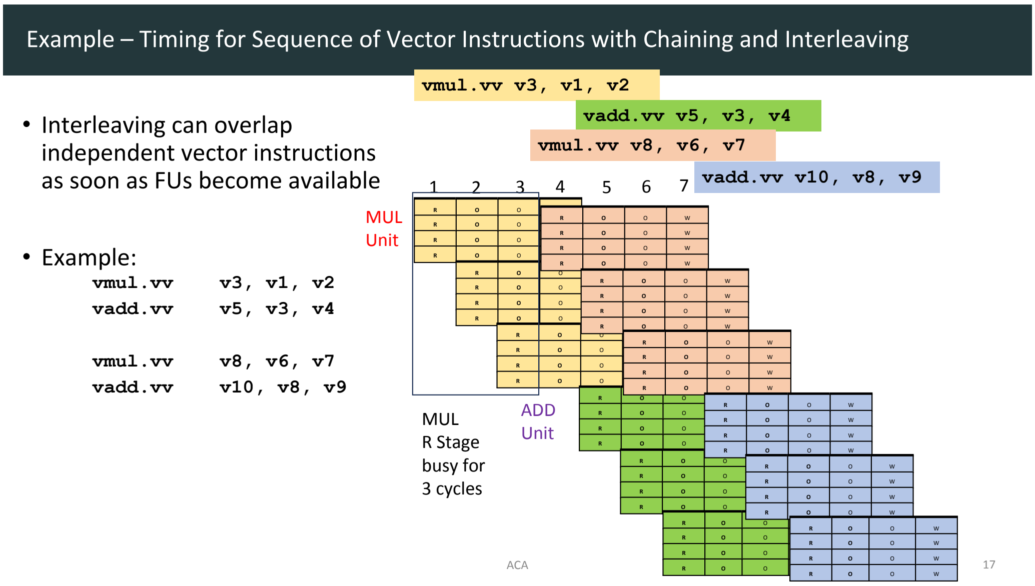
Example – Timing for Sequence of Vector Instructions

- Full result only ready after last cycle of vector instruction
- An instruction using the result needs to wait until completed
- Causes a dead time (also called recovery time) – delay until next vector instruction can start down pipeline

Figure 3.56: Example - Timing for Sequence of Vector Instruction

Vector Chaining

- vector version of *forwarding paths*
- results are forwarded element-wise to next FU via chaining



Example – Timing for Sequence of Vector Instructions with Chaining and Interleaving

- Interleaving can overlap independent vector instructions as soon as FUs become available
- Example:

```
vmul.vv    v3, v1, v2
vadd.vv    v5, v3, v4

vmul.vv    v8, v6, v7
vadd.vv    v10, v8, v9
```

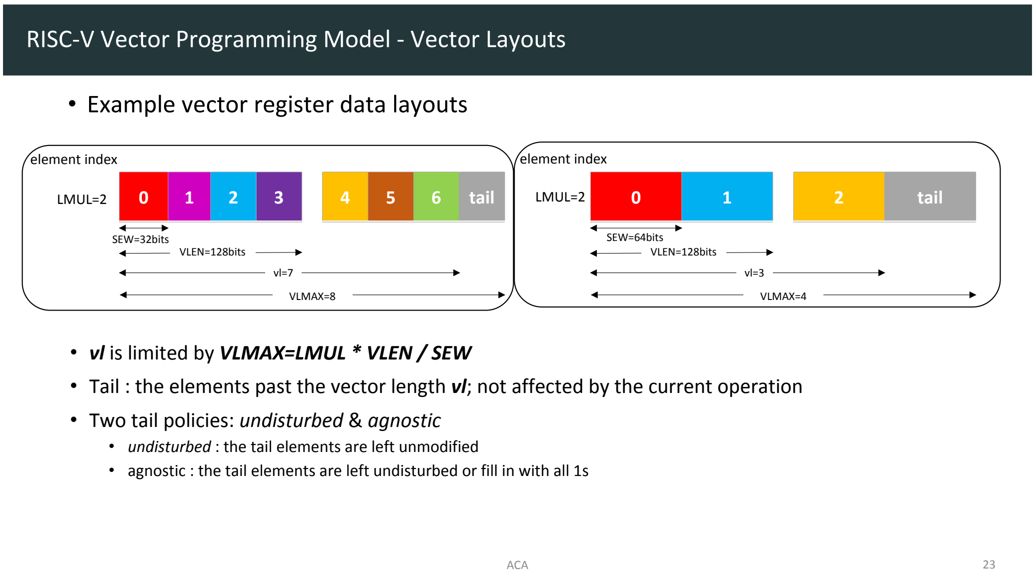
Figure 3.57: Example - Timing for Sequence Vector with Chaining and Interleaving

3.5.3 The RISC-V Vector Instruction Set

- RISC-V "V" Vector Extension
  - standard extension for the RISC-V ISA
- memory-register vector instructions (operations on registers)
- vector and vector element sizes are configurable (vectors can be longer than one vector register)
- Control Status Register (CSR): specialized register to save configuration and status of processor

Programming Model

- vector register and vector length
  - 32 vector data registers (`v0` - `v31`) each **VLEN** bits long
  - vector length register **VL**
    - defines on how many elements will the next vector operation be executed
  - vector type register **VLTYPE**
    - used to define vector length via parameter **SEW** (selected element width) and **LMUL**
    - used to define *tail* and *mask policy* via **vta** and **vma**
  - vector byte length **VLENB**
    - read-only, holds value **VLEN/8**
    - used to define vector register length **VLEN** (fixed)
  - vector length register **VL**
    - read-only, can be updated by the **vset{i}vl{i}** instructions
    - used to define on how many elements will the next vector operations be executed
    - **vl** is limited by $\mathrm{VLMAX} = \mathrm{LMUL} \cdot \frac{\mathrm{VLEN}}{\mathrm{SEW}}$
  - vector start **vstart**
    - used to define the index of first element to be executed by a vecto r instruction



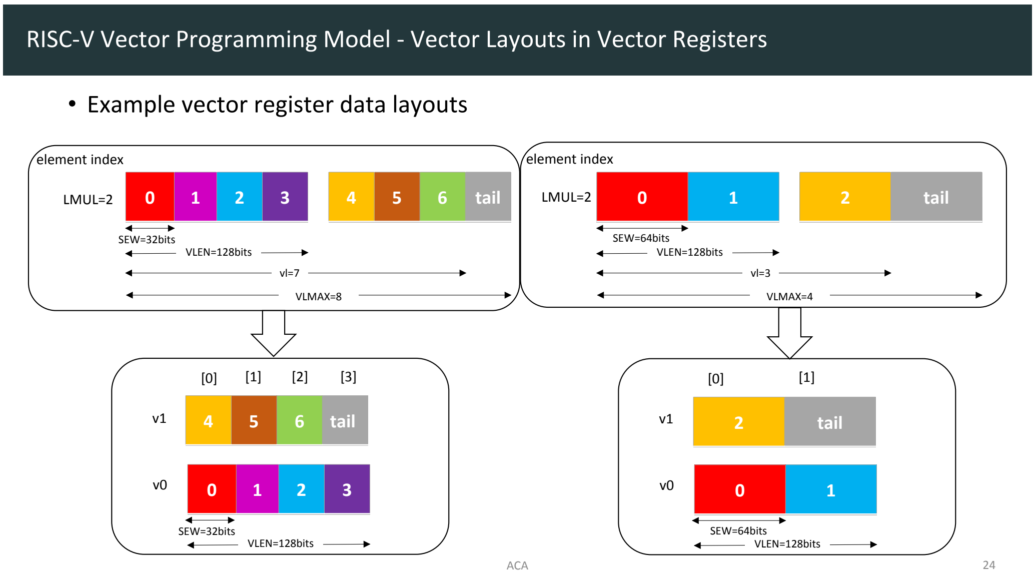
RISC-V Vector Programming Model - Vector Layouts

- Example vector register data layouts

- ***vl*** is limited by ***VLMAX=LMUL * VLEN / SEW***
- Tail : the elements past the vector length ***vl***; not affected by the current operation
- Two tail policies: *undisturbed* & *agnostic*
  - *undisturbed* : the tail elements are left unmodified
  - agnostic : the tail elements are left undisturbed or fill in with all 1s

ACA 23

Figure 3.58: Vector Layouts



RISC-V Vector Programming Model - Vector Layouts in Vector Registers

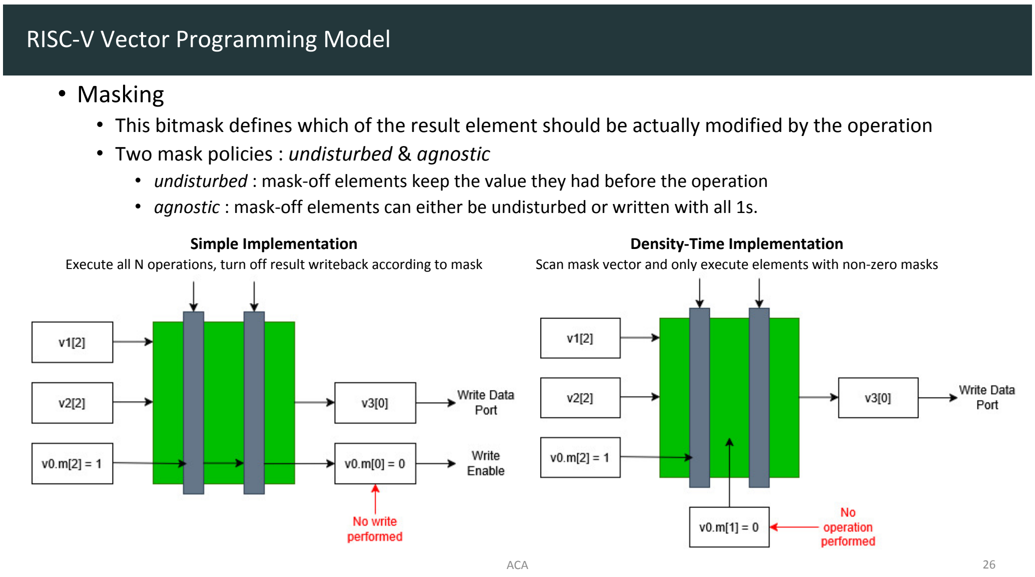
- Example vector register data layouts

ACA 24

Figure 3.59: Vector Layouts in Vector Registers

Vector Masking

- the mask value used to control execution of a masked vector instruction is always supplied by vector register **v0**
- where available, masking is encoded in a *single-bit* ***vm*** field in the instruction word



RISC-V Vector Programming Model

- Masking
  - This bitmask defines which of the result element should be actually modified by the operation
  - Two mask policies : *undisturbed* & *agnostic*
    - *undisturbed* : mask-off elements keep the value they had before the operation
    - *agnostic* : mask-off elements can either be undisturbed or written with all 1s.

Simple Implementation

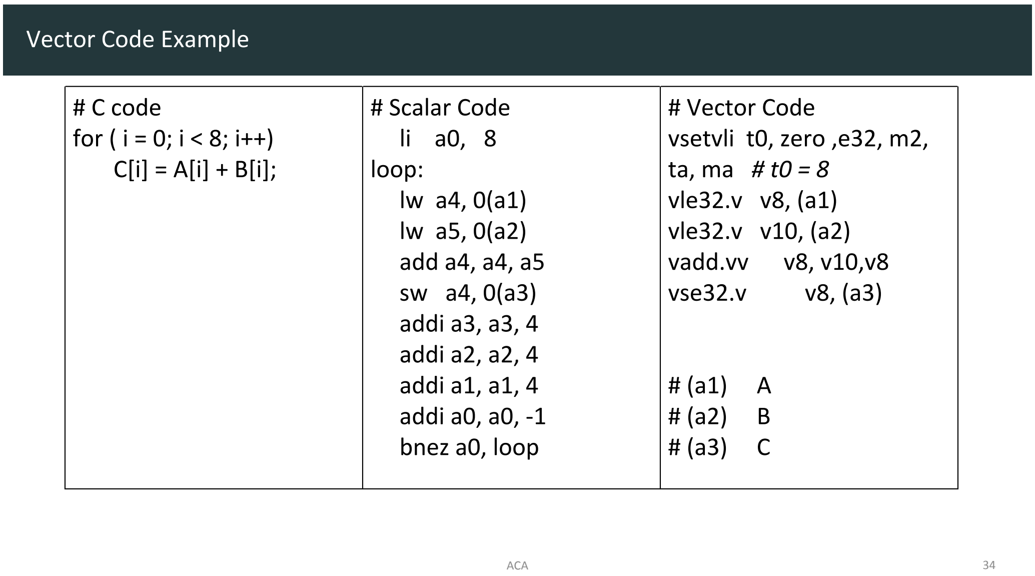
Execute all N operations, turn off result writeback according to mask

Density-Time Implementation

Scan mask vector and only execute elements with non-zero masks

ACA 26

Figure 3.60: Vector Masking



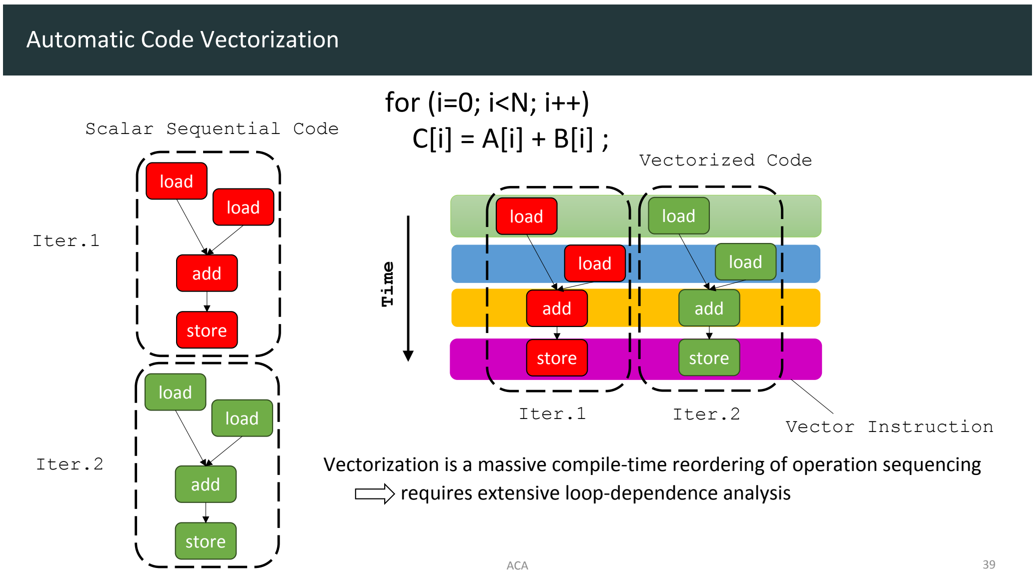
Vector Code Example

# C code	# Scalar Code	# Vector Code
for (i = 0; i < 8; i++)	li a0, 8	vsetvli t0, zero ,e32, m2,
C[i] = A[i] + B[i];	loop:	ta, ma *# t0 = 8*
	lw a4, 0(a1)	vle32.v v8, (a1)
	lw a5, 0(a2)	vle32.v v10, (a2)
	add a4, a4, a5	vadd.vv v8, v10,v8
	sw a4, 0(a3)	vse32.v v8, (a3)
	addi a3, a3, 4	
	addi a2, a2, 4	
	addi a1, a1, 4	# (a1) A
	addi a0, a0, -1	# (a2) B
	bnez a0, loop	# (a3) C

ACA 34

Figure 3.61: Vector Code Example

3.5.4 Vectorization



Automatic Code Vectorization

```
for (i=0; i<N; i++)
  C[i] = A[i] + B[i] ;
```

Vectorization is a massive compile-time reordering of operation sequencing

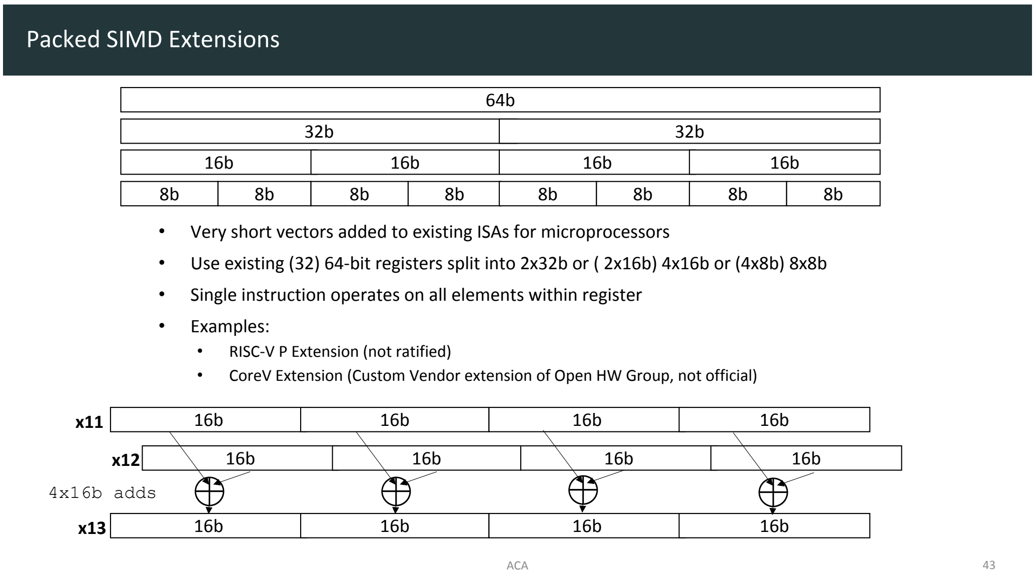
⇨ requires extensive loop-dependence analysis

Figure 3.62: Automatic Code Vectorization

3.5.5 C Vector Intrinsics

Please see Section 8.1

3.5.6 Packed SIMD



Packed SIMD Extensions

- Very short vectors added to existing ISAs for microprocessors
- Use existing (32) 64-bit registers split into 2x32b or (2x16b) 4x16b or (4x8b) 8x8b
- Single instruction operates on all elements within register
- Examples:
  - RISC-V P Extension (not ratified)
  - CoreV Extension (Custom Vendor extension of Open HW Group, not official)

Figure 3.63: Automatic Code Vectorization

- **Pros of Packed SIMD**
  👍 no extra HW co-processor
  👍 SIMD unit can share resources in pipeline (make ALU a SIMD ALU)
- **Cons of Packed SIMD**
  👎 no configurable vector length
  👎 usually no wider load/store unit
  👎 limited by scalar register sizes

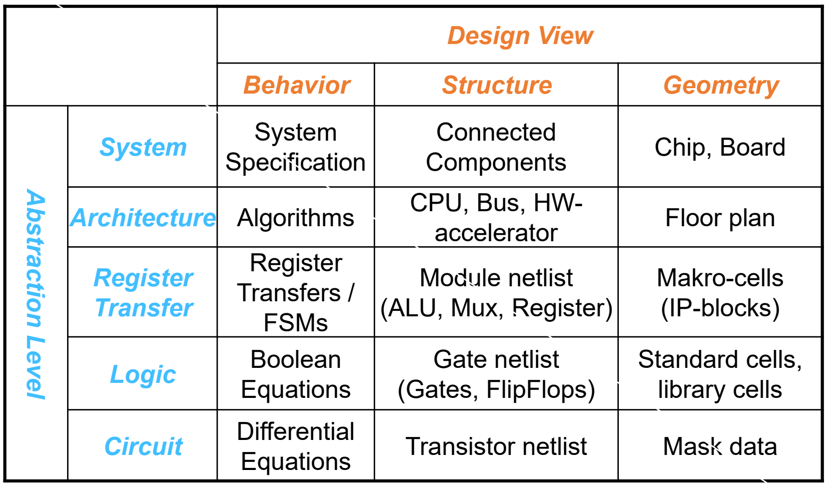
3.5.7 A look at a real vector unit - ARA

This is not relevant for the exam

4 Block D

4.1 Introduction to High Level Synthesis (HLS)

4.1.1 HW Design Flow in a Nutshell



		Design View		
		Behavior	Structure	Geometry
Abstraction Level	System	System Specification	Connected Components	Chip, Board
	Architecture	Algorithms	CPU, Bus, HW-accelerator	Floor plan
	Register Transfer	Register Transfers / FSMs	Module netlist (ALU, Mux, Register)	Makro-cells (IP-blocks)
	Logic	Boolean Equations	Gate netlist (Gates, FlipFlops)	Standard cells, library cells
	Circuit	Differential Equations	Transistor netlist	Mask data

Figure 4.1: Abstraction Levels & Design Views

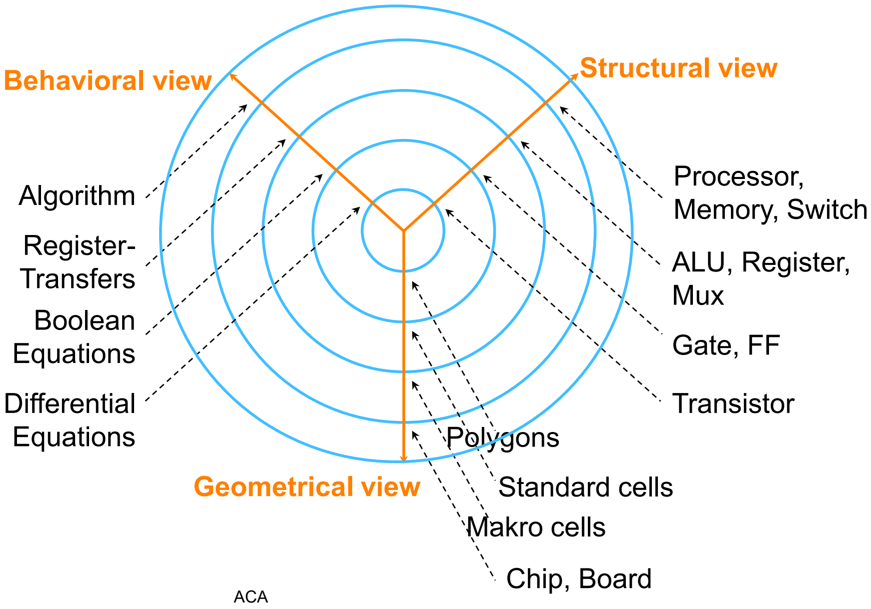


Figure 4.2: Y-Diagram

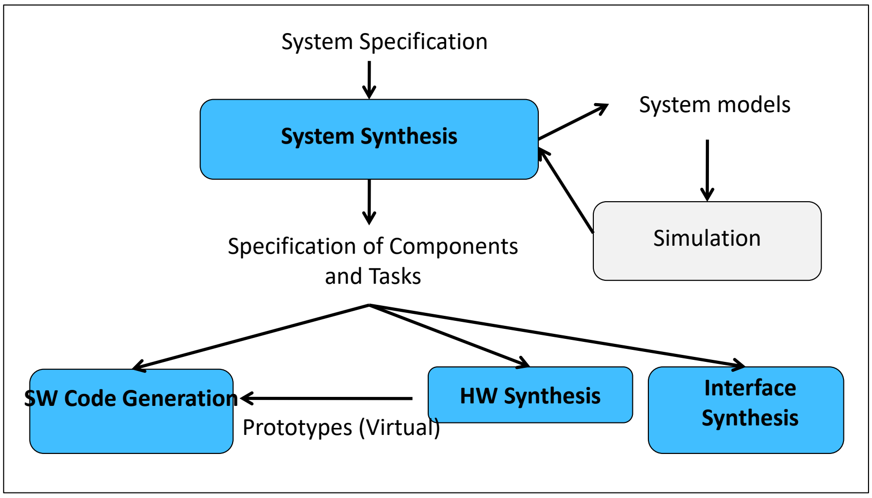


Figure 4.3: System Specification

System Synthesis

- **Inputs**
  - specification of the system
    - description of the functionality and design constraints
- **Typical Synthesis Steps**
  - description of
    - functionality as a set of communicating tasks
    - behavior of tasks at an algorithmic level
    - task communication
  - allocation of system components such as processors, buses, memory, ...
  - binding of tasks and inter-task communication to system components (HW/SW partitioning)
- **Output**
  - An output specification of components, tasks, and inter-task communication that guarantees to meet the system specification

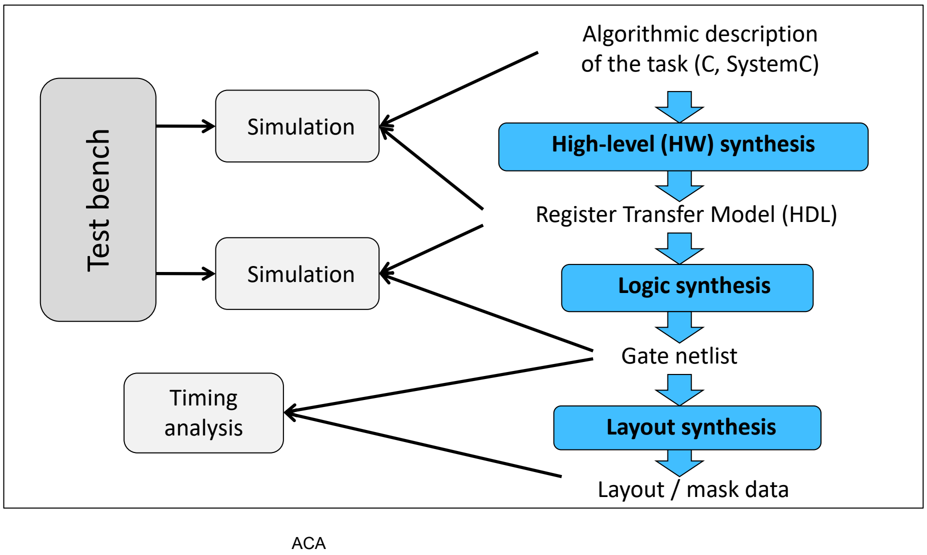


Figure 4.4: ASIC HW Synthesis Flow

HLS Synthesis Step

- Input
  - Algorithmic description of a task (e.g., in C, C++, SystemC)
  - Design constraints (maximal latency, available resources, ...)
- Synthesis steps
  - Static code analysis and code optimization
  - Datapath synthesis (Scheduling, allocation, binding)
  - Control unit synthesis (FSM implementation)
- Output:
  - Description of hardware module at RT level

Logic Synthesis Step

- Input
  - Description of HW module on RT level
  - Design constraints (minimal clock frequency, maximal area, ...)
  - Gate library
- Synthesis steps
  - Logic optimization
  - Technology mapping
- Output
  - Gate netlist

Physical Synthesis Step

- Input
  - Gate library
  - Design constraints
  - Layout library (P-cells)
- Synthesis steps
  - Placement of modules
  - Routing of signal nets
- Output
  - Layout, mask data

Software Compilation

- Inputs
  - Algorithmic description of a task
- Synthesis steps
  - Static Code Analysis and Optimization
  - Code Generation (instruction selection, register allocation, and assignment)
  - Assembler, linker, loader
- Outputs
  - Assembly code/machine code for the target processor

Interface Synthesis

- Input
  - Description of inter-task communication
  - Design constraints (protocols, data rates, ...)
- Outputs
  - Drivers, bus interfaces, ...

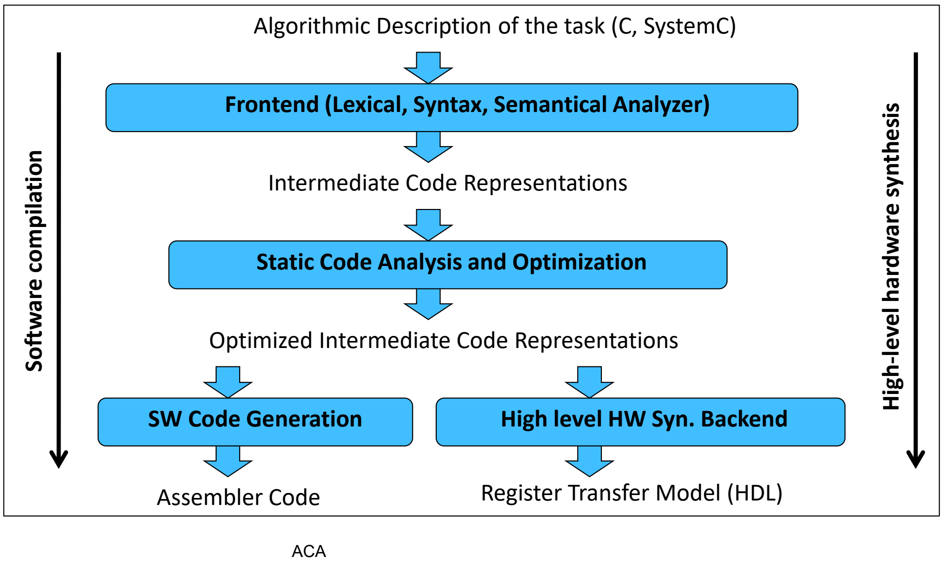
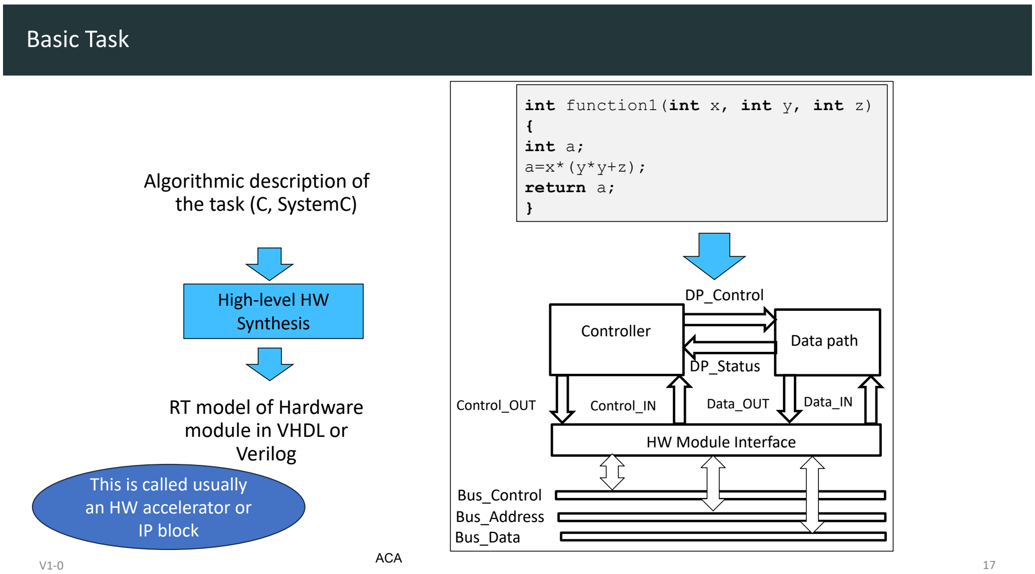

Algorithmic Description of the task (C, SystemC)
Frontend (Lexical, Syntax, Semantical Analyzer)
Intermediate Code Representations
Static Code Analysis and Optimization
Optimized Intermediate Code Representations
SW Code Generation
High level HW Syn. Backend
Assembler Code
Register Transfer Model (HDL)
Software compilation
High-level hardware synthesis
ACA

Figure 4.5: HLS and SW Compilation Flow

4.1.2 The HLS Synthesis Task


Basic Task

```
int function1(int x, int y, int z)
{
int a;
a=x*(y*y+z);
return a;
}
```

Figure 4.6: Basic Task

Other names for HLS

- High-level Hardware synthesis
- algorithmic synthesis
- behavioral synthesis
- C synthesis

Classes of Hardware Components

- *Data-oriented designs*
  - examples: video signal processing, compression, encryption, ...
- *Control-oriented designs*
  - examples: traffic light control, industrial machine control, ...
- *HLS works better on data-oriented designs*

Performance Metrics

- **Clock Cycle Time ΔT**

  - cycle duration of the driving clock of the HW module
  - the combinatorial path in the circuit with the largest delay places a lower limit on the clock cycle time (critical path).
- **Latency Λ**
  - number of clock cycles between the start of processing a block of data and the point of time at which the result is ready at the output.
- **Processing time $t_{\text{exe}} = \Lambda \cdot \Delta T$**
- **Throughput T**
  - number of blocks of data that can be processed in a fixed time
- **Chip Area (Application-specific integrated circuit (ASIC))**
  - estimated via gate count
  - Datapath: Number of Hardware Operation Units, such as multipliers, ALUs, registers, multiplexers, …
- **FPGA Resources**
  - number of LUTs, number of Digital Signal Processor (DSP) Blocks, …
- **Power/Energy Consumption**
  - Dynamic power consumption: Power consumed by switching transistors in the circuit
  - Static power consumption: Power consumed due to leakage currents.

Design Goals and Constraints

- Synthesis algorithms handle two typical cases:
- *Timing-constrained*
  - constrained: implement task such that it can compute results within a maximum number of clock cycles (maximal latency)
  - second goal: minimize the number of registers (register sharing), multiplexers, control unit states, …
- *Resource-constrained*
  - constrained: Implement task with a fixed maximum number of functional units (adders, ALUs, multipliers) in the datapath.
  - goal: minimize latency
  - second goal: minimize the number of registers (register sharing), multiplexers, control unit states, …

Synchronous HW Design

- All registers in the control unit and the datapath share the same clock
- Assumptions for simplification:
  - Functional units have a fixed and known delay, such that the number of clock cycles to execute an operation is assumed to be fixed and data-independent.
  - The delay of interconnects and multiplexers can be neglected.
- Real-life:
  - Longest combinatorial path in the circuit will determine the maximal clock frequency.
  - Logic synthesis will try to optimize the circuit depending on the target clock frequency and area.

Datapath Synthesis Steps

- Scheduling:
  - Determines the start time of each operation
- Binding:
  - Determines on which functional units the operation is executed.
  - Determines in which registers variables are saved.
- Allocation:
  - Selection of resources, such as functional units, registers, and multiplexers.

Interface Synthesis

- Interfaces can differ strongly
- Interfaces may consist of memory, registers, FIFOs and FSMs for communication protocols
- Crossing of clock domains is possible

  - e.g. between bus clock and HW module clock

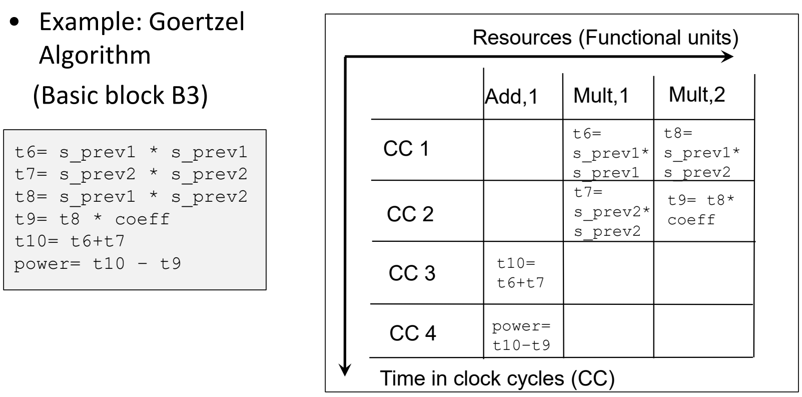
4.1.3 Datapath Synthesis HW Resources

HW Resources in the Datapath

- Functional units: Adders, multipliers, ALUs, …
  - Execute operations on data (e.g., Add, Shift, AND, OR, Mult, …)
  - Fixed and known delay
  - Fixed and known area demand
- Signal nets and multiplexers
  - Delay and area demand is neglected.
- Memory elements: Registers
  - Delay and area demand is neglected.
- NFU (Non-functional Unit)
  - Non-existent helper resource
  - used to execute special `NOP`, `LOOP`, `BRANCH`, `CALL` operations (more on this later)
- Functional units are identified by a pair (k_r, z_r)
  - type: $k_r \in K$ with $K = \{\text{ALU}, \text{MULT}, ...\}$
  - index: $z_r = 1, 2, ...$
  - example
    - $(\text{ALU}, 1), (\text{ALU}, 2), (\text{MULT}, 1)$

Time-Resource-Plane (TRP)

- X-axis: *Resources*
  - List allocated functional units
  - Assign operations to functional units (Binding)
- y-axis: *Time*
  - Division into clock cycles.
  - Plan temporal order of the operations
  - Select start times of operations (Scheduling)
  - Values must be saved in registers between clock cycles.



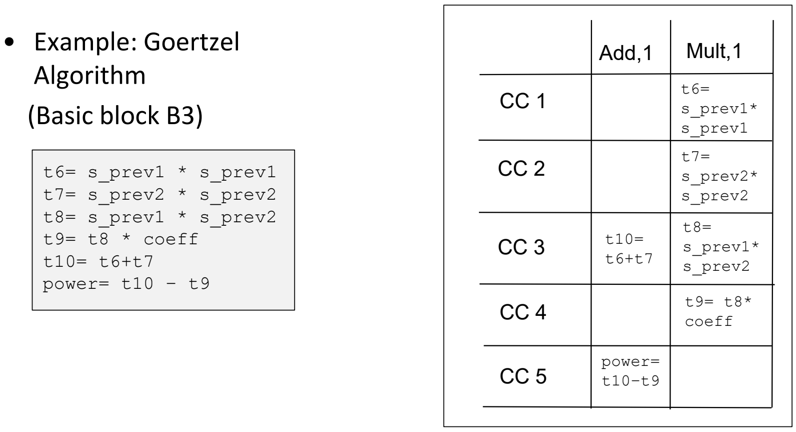
- Example: Goertzel Algorithm

(Basic block B3)

```
t6= s_prev1 * s_prev1
t7= s_prev2 * s_prev2
t8= s_prev1 * s_prev2
t9= t8 * coeff
t10= t6+t7
power= t10 - t9
```

	Add,1	Mult,1	Mult,2
CC 1		t6= s_prev1* s_prev1	t8= s_prev1* s_prev2
CC 2		t7= s_prev2* s_prev2	t9= t8* coeff
CC 3	t10= t6+t7		
CC 4	power= t10-t9		

Figure 4.7: TRP Example Goertzel (1)



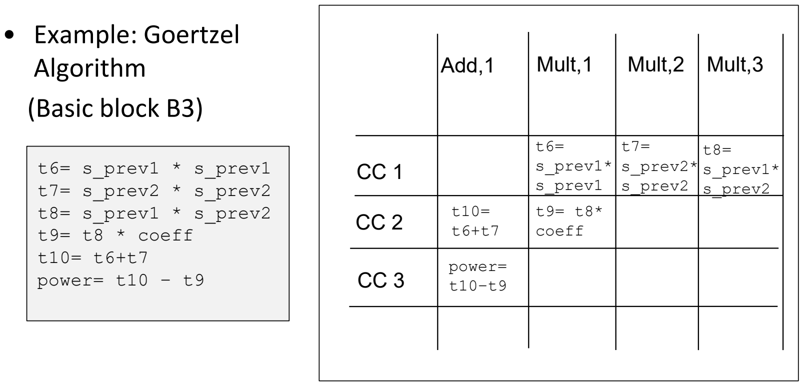
- Example: Goertzel Algorithm

(Basic block B3)

```
t6= s_prev1 * s_prev1
t7= s_prev2 * s_prev2
t8= s_prev1 * s_prev2
t9= t8 * coeff
t10= t6+t7
power= t10 - t9
```

	Add,1	Mult,1
CC 1		t6= s_prev1* s_prev1
CC 2		t7= s_prev2* s_prev2
CC 3	t10= t6+t7	t8= s_prev1* s_prev2
CC 4		t9= t8* coeff
CC 5	power= t10-t9	

Figure 4.8: TRP Example Goertzel (2)



- Example: Goertzel Algorithm

(Basic block B3)

```
t6= s_prev1 * s_prev1
t7= s_prev2 * s_prev2
t8= s_prev1 * s_prev2
t9= t8 * coeff
t10= t6+t7
power= t10 - t9
```

	Add,1	Mult,1	Mult,2	Mult,3
CC 1		t6= s_prev1* s_prev1	t7= s_prev2* s_prev2	t8= s_prev1* s_prev2
CC 2	t10= t6+t7	t9= t8* coeff		
CC 3	power= t10-t9			

Figure 4.9: TRP Example Goertzel (3)

Pareto-Optimality

- A solution is Pareto optimal if no other solution is better in **all** design performance metrics.
- Different Pareto-optimal solutions allow different *trade-offs* between the design performance metrics.
- The best solution is picked based on preferences for design performance metrics.

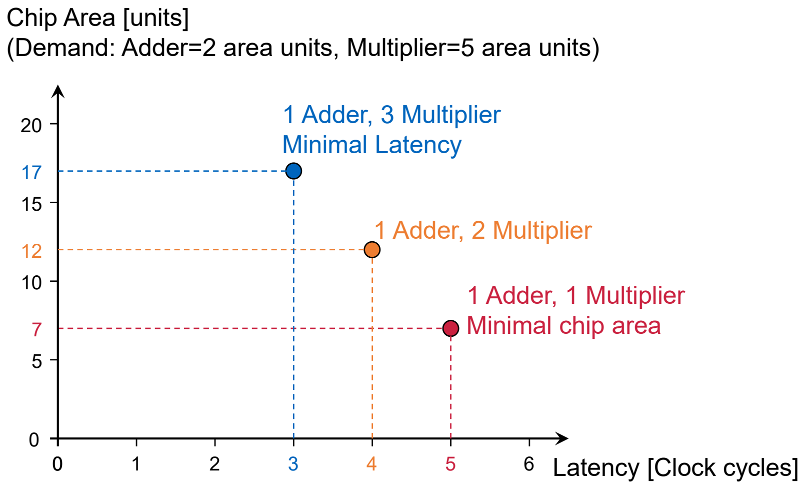
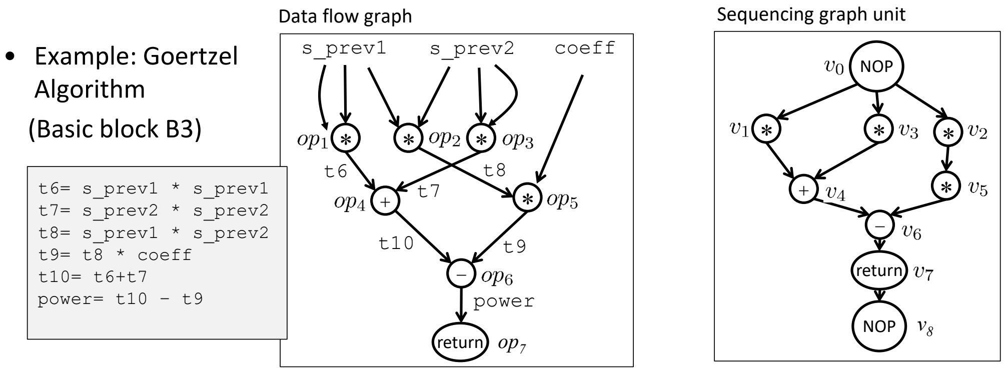


Figure 4.10: Example Pareto

4.1.4 Sequencing Graphs

- Sequencing graph: $G_s = (V_s, E_s)$
  - hierarchy of Directed Acyclic Grpah (DAG)s
  - each graph is called a *Sequencing Graph Unit (SGU)*
  - SGUs are polar: one source and one sink node is added which are labeled `NOP`
- Nodes: $V_s = \{v_i : i = 0, ..., n\}$
  - `NOP`s
  - operation $\in \{+, <, >, \cdot, ...\}$
  - hierarchical node (`CALL`, `BRANCH`, `LOOP`)
- Edges: $E_s = \{(v_i, v_j) : i, j = 0, ...n\}$
  - between nodes in one SGU (data dependency between two operations)
  - between source and sink (connection between SGU on different hierarchical levels)
- paths describe the concurrent operations that may possibly be executed in parallel

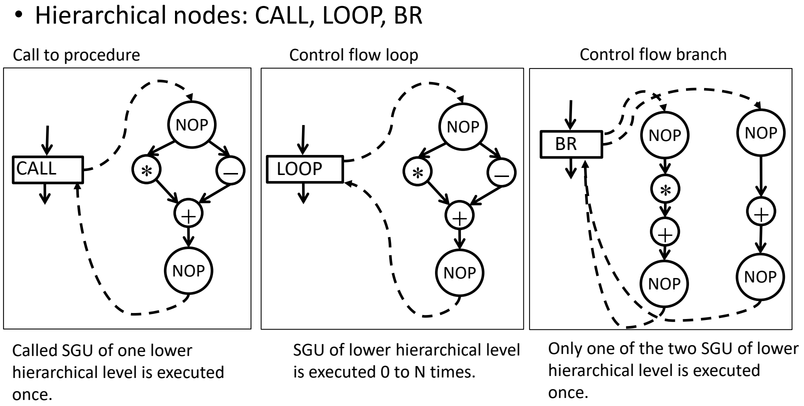


- Example: Goertzel Algorithm

(Basic block B3)

```
t6= s_prev1 * s_prev1
t7= s_prev2 * s_prev2
t8= s_prev1 * s_prev2
t9= t8 * coeff
t10= t6+t7
power= t10 - t9
```

Figure 4.11: Example Goertzel Algorithms with SGU



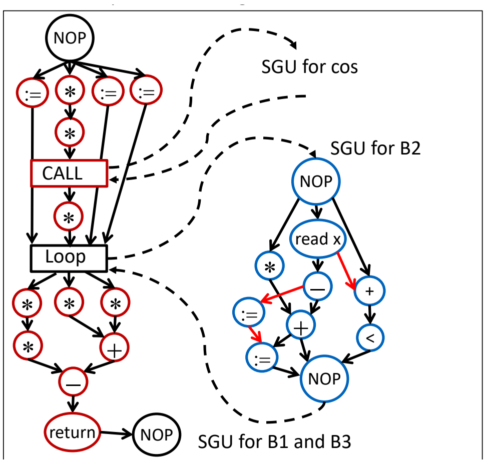
- Hierarchical nodes: CALL, LOOP, BR

Called SGU of one lower hierarchical level is executed once.

SGU of lower hierarchical level is executed 0 to N times.

Only one of the two SGU of lower hierarchical level is executed once.

Figure 4.12: Hierarchical Nodes in SGU



```
B1: s_prev1 := 0.0
    s_prev2 := 0.0
    i:=0
    t1 := 2*3.14
    f := t1 * freq
    param f
    t2 := call cos,1
    coeff:=2.0*t2
```

```
B2: t3:= coeff * s_prev1
    t4:= x[i]
    t5 := t4 - s_prev2
    s := t3 + t5
    s_prev2 := s_prev1
    s_prev1 := s
    i:=i+1
    if i < 64 goto B2
```

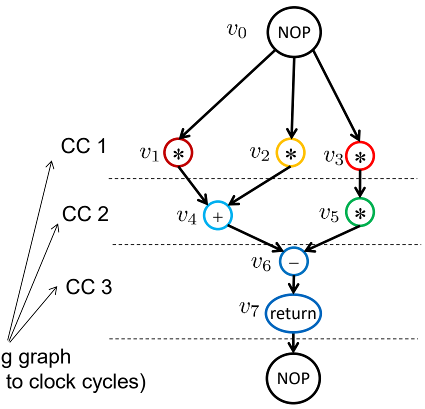
```
B3: t6:= s_prev1 * s_prev1
    t7:= s_prev2 * s_prev2
    t8:= s_prev1 * s_prev2
    t9:= t8 * coeff
    t10:= t6+t7
    power:= t10 - t9
    return power
```

Figure 4.13: Full Example Goertzel Algorithm

- Example: Goertzel Algorithm

(Basic block B3)

```
t6= s_prev1 * s_prev1
t7= s_prev2 * s_prev2
t8= s_prev1 * s_prev2
t9= t8 * coeff
t10= t6+t7
power= t10 - t9
return power
```



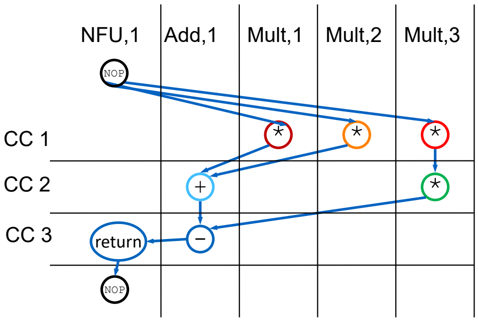
Scheduled sequencing graph
(Operations assigned to clock cycles)

Figure 4.14: Sequencing Graph in the TRP (1)

- Example: Goertzel Algorithm

(Basic block B3)

```
t6= s_prev1 * s_prev1
t7= s_prev2 * s_prev2
t8= s_prev1 * s_prev2
t9= t8 * coeff
t10= t6+t7
power= t10 - t9
return power
```



Scheduled sequencing graph with binding
(Operations assigned to clock cycles and operational units)

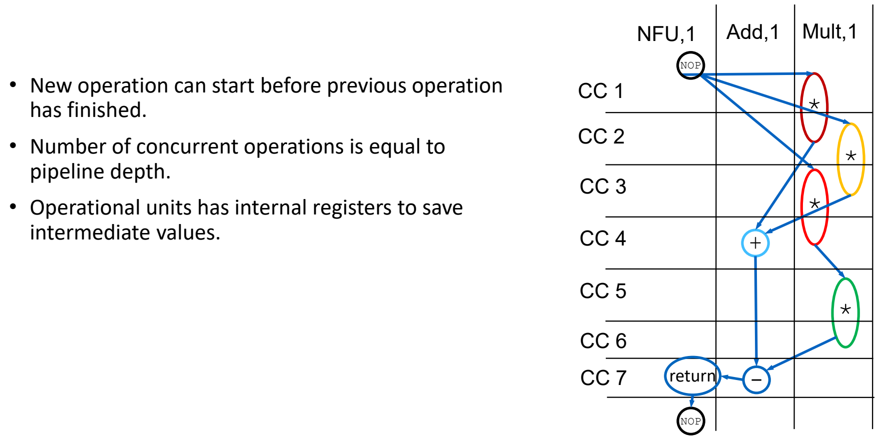
Figure 4.15: Sequencing Graph in the TRP (2)

Operation Chaining

- The delay of operational units can allow for two operations to be executed in one clock cycle.

Multi-Cycle Operations

- The delay of functional elements may require several clock cycles for the execution of the operation.



- New operation can start before previous operation has finished.
- Number of concurrent operations is equal to pipeline depth.
- Operational units has internal registers to save intermediate values.

Figure 4.16: Pipelined Operational Units

4.2 Scheduling for High Level Synthesis

4.2.1 The Scheduling Task

Recap <u>SGU</u>

- $G_{s,u} = (V_u, E_u) \quad V_u = v_x, ..., v_y \subset V_s$
  - x: index of *Source NOP Node*
  - y: index of the *Sink NOP Node* (with $y > x$)
- execution delay of operation $D = \{d_i : i = x, ..., y\}$
  - *NOPs have an execution delay of zero*
- Wanted:
  - start time for each operation $T = \{t_i : i = x, ..., y\}$
- Scheduling is a function τ
  - Constraints: The starting time of an operation must be at least as large as the starting time of all predecessor operations plus their execution delay.
  - Result: A scheduled sequence graph where each node is marked with its starting time.

$$\tau : V_u \to \mathbb{Z}^+; \tau(v_i) = t_i$$
$$t_i \geq t_j + d_j \quad \forall i, j : (v_j, v_i) \in E_u$$

- Latency of a schedule: $\Lambda = t_y - 1$

4.2.2 As-soon-as-possible (ASAP) Schedule

- Schedule for unconstrained resources
- Goal: minimal latency
- Solution: topological sorting of the sequencing graph
- ASAP start time for node v_i

$$t_i^S = \max_{j:(v_j,v_i)\in E_u} \left(t_j^S + d_j\right)$$

- quadratic complexity: $O(|V^2|)$

```
ASAP_schedule(G_s,u(V_u,E_u)) {
Start time of node v[x]: t_S[x]=1;
repeat {
        Select node v[i], whose direct predecessors v[j] all have been assigned a
starting time.
    Set start time for node v[i]:
        t_S[i]=max(t_S[j]+d[j]);
    } until node v[y] has been assigned a starting time.
    return (t_S);
}
```

Code 4.1: Pseudo-Code for ASAP Algorithm

4.2.3 As-late-as-possible (ALAP) Schedule

- Schedule with fixed latency (time-constrained)
- Given latency

$$\Lambda^L = t_y^L - 1 = \Lambda_{\max}$$

- Goal: find the latest starting time for all operations such that the maximum latency constraint is met

$$t_i^L = \min_{j:(v_j,v_i)\in E_u} \left(t_j^L - d_i\right)$$

- same complexity as ASAP

```
ALAP_schedule(G_s,u(V,E),Lambda_max) {
    Start time for node v[y]: t_L[y]=Lambda_max+1
    repeat {
        Select node v[i], whose direct successors v[j] all have been assigned a
starting time.
        Set start time for node v[i]:
        t_L[i]=min(t_L[j]-d[i])
    } until node v[x] has been assigned starting time
    return (t_L)
}
```

Code 4.2: Pseudo-Code for ALAP Algorithm

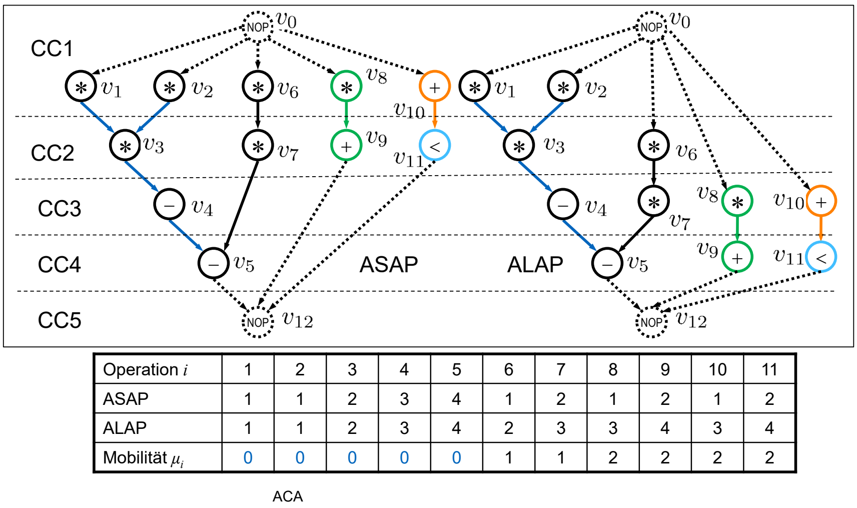
4.2.4 Mobility of Operations

- Given an upper constraint on latency: $\Lambda = t_y - 1 \leq \Lambda_{\max}$
- ASAP Schedule: minimal start times for operations
- ALAP Schedule: maximal start times for operations
- Mobility of operations on the time axis:

$$\mu_i = t_i^L - t_i^S, \quad i = x, ..., y$$

- For opeations with $\mu_i = 0$
  - the start time is fixed: $t_i = t_i^L = t_i^S$
  - These operations are located on the critical path

    ❗ **not the same as critical paths in logic circuits**
- There is no schedule for latency constraint $\Lambda \leq \Lambda_{\max}$
  - possible if $t_y^S > \Lambda_{\max} + 1$
  - or $t_x^L < 1$



Operation i	1	2	3	4	5	6	7	8	9	10	11
ASAP	1	1	2	3	4	1	2	1	2	1	2
ALAP	1	1	2	3	4	2	3	3	4	3	4
Mobilität μ_i	0	0	0	0	0	1	1	2	2	2	2

Figure 4.17: Example: DE-Solver with ASAP and ALAP

4.2.5 Hu's Algorithm

- goal: minimize latency
- resource constraint: maximum number of resources $= a$
- requirements
  - only one type of resource
  - All execution delays are 1
    - Split up operations into multiple if a larger delay exists
- properties
  - linear complexity: $O(n)$
  - greedy algorithm
  - optimal: finds a schedule with minimal latency

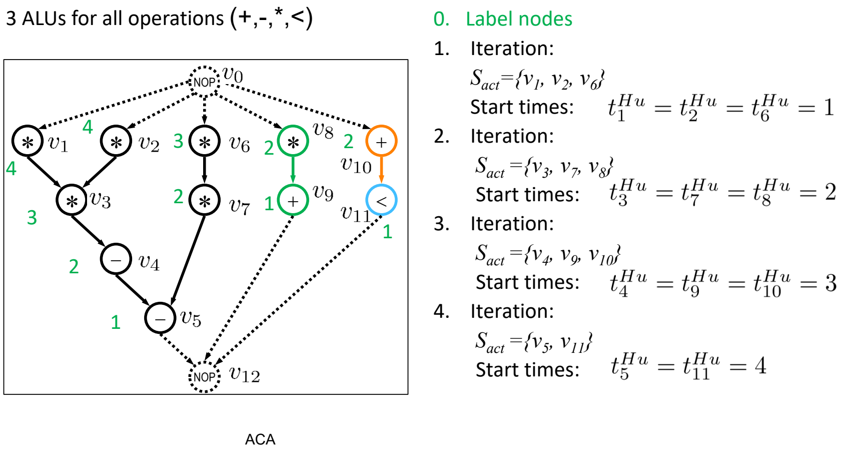
Set of ready operations

$$U_{\text{act}} = \left\{ v_i \mid \forall_{j:(v_j,v_i)\in E_u} t_j + d_j \leq t_{\text{act}} \right\}$$
$$\Longleftrightarrow \text{direct predecessors finished}$$

- label each node with length of longest path from this node to the sink (α_i)
- set of operations to start S_{act}
  - must be operations that are ready
  - must be less than or equal to the number of available resources a
  - the label α_i should be maximal

```
HU(G_s,u(V,E),a) {
    Label nodes v[i] with max. path length alpha[i] to sink v[y]
    Set start time for source node v[x]: t_HU[x]=1
    Set t_act=1
    repeat {
        Select set of nodes S_act, such that for v[i] in S_act:
            1. v[i] is in U_act
            2. alpha[i] of v[i] in S_act is maximal
            3. Number of elements in S_act: |S|<=a
        Set start time of all v[i] in S_act: t_HU[i]=t_act
        Set t_act=t_act+1
    } until sink node v[y] was assigned a start time
    return (t_HU)
}
```

Code 4.3: Hu's Algorithm



3 ALUs for all operations (+,-,*,<)

0. Label nodes
1. Iteration:
   S_{act}={v_1, v_2, v_6}
   Start times: $t_1^{Hu} = t_2^{Hu} = t_6^{Hu} = 1$
2. Iteration:
   S_{act}={v_3, v_7, v_8}
   Start times: $t_3^{Hu} = t_7^{Hu} = t_8^{Hu} = 2$
3. Iteration:
   S_{act}={v_4, v_9, v_{10}}
   Start times: $t_4^{Hu} = t_9^{Hu} = t_{10}^{Hu} = 3$
4. Iteration:
   S_{act}={v_5, v_{11}}
   Start times: $t_5^{Hu} = t_{11}^{Hu} = 4$

ACA

Figure 4.18: Example: DE-Solver with Hu's Algorithm

4.2.6 List Scheduling

Priorities

- resource constrained (goal: minimize latency)
  - number of resources of type k : α_k
  - priority equals maximal sum of execution delays on paths to sink

$$\text{Prio}(v_i) = \max_v \left(\sum_{w \in H_v} d_w \right) \quad \text{with} \quad H_v \text{ equals to paths from } v_i \text{ to sink}$$

- time constrained (goal: minimize resources)
  - maximal latency: $\Lambda \leq \Lambda_{\text{max}}$
  - *slack of a node*: distance to ALAP start time
  - priority at time t_{act} equals slack: $s_i = t_i^L - t_{\text{act}}$
- heuristic and greedy algorithm based on priorities

$$\text{Prio}(v_i) = s_i = t_i^L - t_{\text{act}}$$

Operation Sets

- Set of candidate operations ready to be executed on a resource of type k

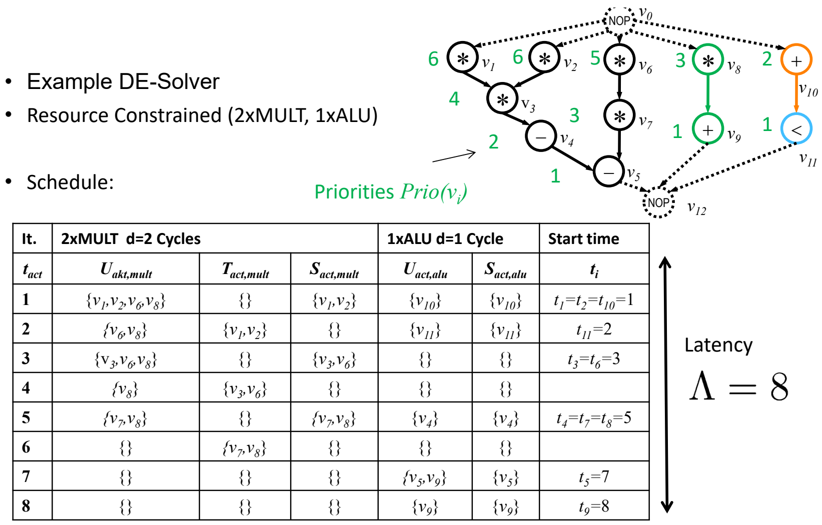
$$U_{\text{act},k} = \left\{ v_i \mid v_i \text{ of type } k \wedge \forall_{j:(v_j,v_i) \in E_u} t_j + d_j \leq t_{\text{act}} \right\}$$

- Set of running operations on a resource of type k

$$T_{\text{act},k} = \{v_l \mid v_l \text{ of type } k \wedge t_l + d_l > t_{\text{act}}\}$$

```
LIST_L(G_s,u(V,E),a) {
    Set start time of source node v[x]: t_LR[x]=1
    t_act=1
    repeat {
        foreach type of resource k=1,2,… {
            Find set of candidate operations U_act[k]
            Find set of running operations T_act[k]
            Select starting operations v[i] in S_act[k] such that:
                1. v[i] in U_act[k]
                2. Priorities Prio(v[i]) maximal
                3. Number of running and starting operations smaller than resource
number:|S_act[k]| + |T_act[k]| <= a[k]
                Set start time of v[i] in S_act[k]: t_LR[i]=t_act
        }
        t_act=t_act+1
    } until sink node v[y] was assigned a start time
    return (t_LR)
```

Code 4.4: List Scheduling Algorithm: Resource Constraint



- Example DE-Solver
- Resource Constrained (2xMULT, 1xALU)
- Schedule:

Priorities $Prio(v_i)$

It.	2xMULT d=2 Cycles			1xALU d=1 Cycle		Start time
t_{act}	$U_{akt,mult}$	$T_{act,mult}$	$S_{act,mult}$	$U_{act,alu}$	$S_{act,alu}$	t_i
1	$\{v_1,v_2,v_6,v_8\}$	{}	$\{v_1,v_2\}$	$\{v_{10}\}$	$\{v_{10}\}$	$t_1=t_2=t_{10}=1$
2	$\{v_6,v_8\}$	$\{v_1,v_2\}$	{}	$\{v_{11}\}$	$\{v_{11}\}$	$t_{11}=2$
3	$\{v_3,v_6,v_8\}$	{}	$\{v_3,v_6\}$	{}	{}	$t_3=t_6=3$
4	$\{v_8\}$	$\{v_3,v_6\}$	{}	{}	{}	
5	$\{v_7,v_8\}$	{}	$\{v_7,v_8\}$	$\{v_4\}$	$\{v_4\}$	$t_4=t_7=t_8=5$
6	{}	$\{v_7,v_8\}$	{}	{}	{}	
7	{}	{}	{}	$\{v_5,v_9\}$	$\{v_5\}$	$t_5=7$
8	{}	{}	{}	$\{v_9\}$	$\{v_9\}$	$t_9=8$

Latency

$$\Lambda = 8$$

Figure 4.19: Example: List Scheduling with Resource Contraint

```
LIST_R(G_s,u(V,E),Lambda_max) {
    Set Number of resources: a[k]=1 for all k
    t_L = ALAP_Schedule(G_s,u(V,E),Lambda_max)
    if t_L[x] < 1 then return("No schedule possible")
    Set start time of node v[x]: t_LT[x]=1
    t_act=1
    repeat {
        foreach type of resource k {
            Find set of candidate nodes U_act[k]
            Find set of running nodes T_act[k]
            Compute slack s[i] = t_L[i] – t_act for v[i] in U_act[k]
            Place all v[i] from U_act[k] into S_act[k], with slack s[i]=0
            Set start time of v[i] in S_act[k]: t_LT[i]=t_act
            if |S_act[k]| + |T_act[k]| > a[k] then {
                Update a[k]: a[k] = |S_act[k]| + |T_act[k]|
            }
            if |S_act[k]| + |T_act[k]| < a[k] then {
                {
                    Place nodes v[l] from U_act[k] without S_act[k]  into R_act[k],
                    Such that slack s[l] for v[l] in R_act[k] minimal }
                until |S_act[k]| + |T_act[k]| +  | R_act[k] | = a[k] or no more nodes
in U_act[k]
                Set start time of nodes v[l] in R_act[k]:
t_LT[l]=t_act
}
}
t_act=t_act+1
} until sink node v[y] was assigned a start time
return (t_LT);
```

Code 4.5: List Scheduling Algorithm: Timing Constraint

List Scheduling with Timing Constraint – Improved Version with Restart

- Improved Algorithm: Timing-constrained resource minimization.
- Restart the algorithm each time the number of resources is increased; do not reset the number of resources to 1, but start with the last value.

4.2.7 Force-Directed Scheduling

- Heuristic based on a force-based model
- Timing constrained resource minimization
- Published by Paulin & Knight, TCAD 1989

Distribution of Start Times

- The time frame of possible starting times for node v_i

$$T_i = \left[t_i^{\text{ASAP}} \quad t_i^{\text{ALAP}}\right]$$

- with a width of the time frame of $\mu_i + 1$
- The distribution for the starting time of node v_i at time t_{act}

$$p_{i(t_{\text{act}})} = \begin{cases} \frac{1}{\mu_i+1} & \forall t_{\text{act}} \in T_i \\ 0 & \forall t_{\text{act}} \notin T_i \end{cases}$$

- uniform distribution in the time frame

Resource Demand

- The demand for resources of type k at clock cycle t_{act}

$$q_{k(t_{\text{act}})} = \sum_{\{i:\text{op}_i \text{ is from type } k\}} p_{i(t_{\text{act}})}$$

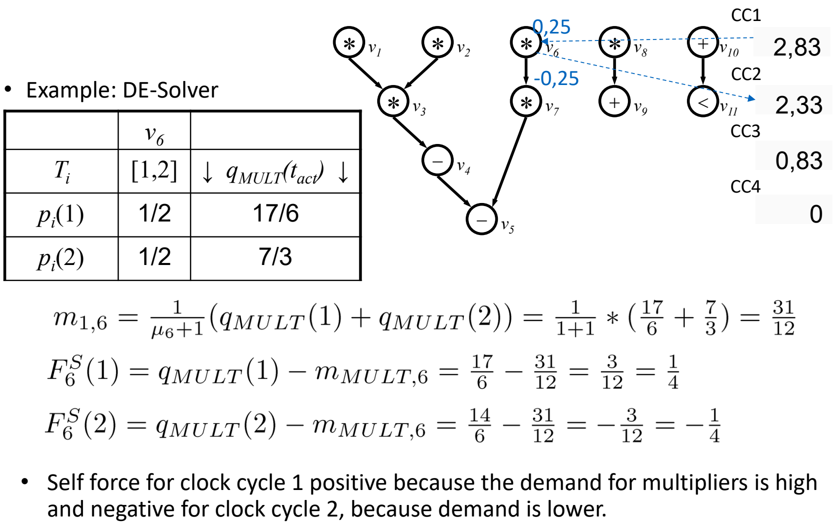
- The mean demand for resources of type k in the time frame T_i

$$m_{k,i} = \frac{1}{\mu_i + 1} \cdot \sum_{t_p = t_i^{\text{ASAP}}}^{t_i^{\text{ALAP}}} q_{k(t_p)}$$

Self Force

$$F_i^{S(t_{\text{act}})} = q_{k(t_{\text{act}})} - m_{k,i}$$

- Difference between the demand of a resource of type k in clock cycle t_{act} and the mean demand for a resource of type k in the time frame T_i of the node
- $F_i^{S(t_{\text{act}}>0}$: In this clock cycle, the demand for the resource is high, pushing node v_i away from t_{act} with a positive self-force.
- $F_i^{S(t_{\text{act}}<0}$: In this clock cycle, the demand for the resource is low, pulling node v_i closer to t_{act} with a negative self-force.



- Example: DE-Solver

	v_6	
T_i	[1,2]	↓ $q_{MULT}(t_{act})$ ↓
$p_i(1)$	1/2	17/6
$p_i(2)$	1/2	7/3

$$m_{1,6} = \frac{1}{\mu_6+1}(q_{MULT}(1) + q_{MULT}(2)) = \frac{1}{1+1} * (\frac{17}{6} + \frac{7}{3}) = \frac{31}{12}$$

$$F_6^S(1) = q_{MULT}(1) - m_{MULT,6} = \frac{17}{6} - \frac{31}{12} = \frac{3}{12} = \frac{1}{4}$$

$$F_6^S(2) = q_{MULT}(2) - m_{MULT,6} = \frac{14}{6} - \frac{31}{12} = -\frac{3}{12} = -\frac{1}{4}$$

- Self force for clock cycle 1 positive because the demand for multipliers is high and negative for clock cycle 2, because demand is lower.

Figure 4.20: Example: Self Force

Shift of Time Frames of Successors / Predecessors

- The selection of a start time for a node changes the time frames for its direct predecessor and successor nodes.
  - node v_j is a direct predecessor or successor of v_i
  - The start time for node v_i is selected as
    - New time frame and mobility for nodes $v_j : t_i = t_{\text{act}}$

$$\tilde{T}_i = \left[\tilde{t}_i^{\text{ASAP}} \;\; \tilde{t}_i^{\text{ALAP}}\right]$$

$$\tilde{\mu}_j = \tilde{t}_j^{\text{ALAP}} - \tilde{t}_j^{\text{ASAP}}$$

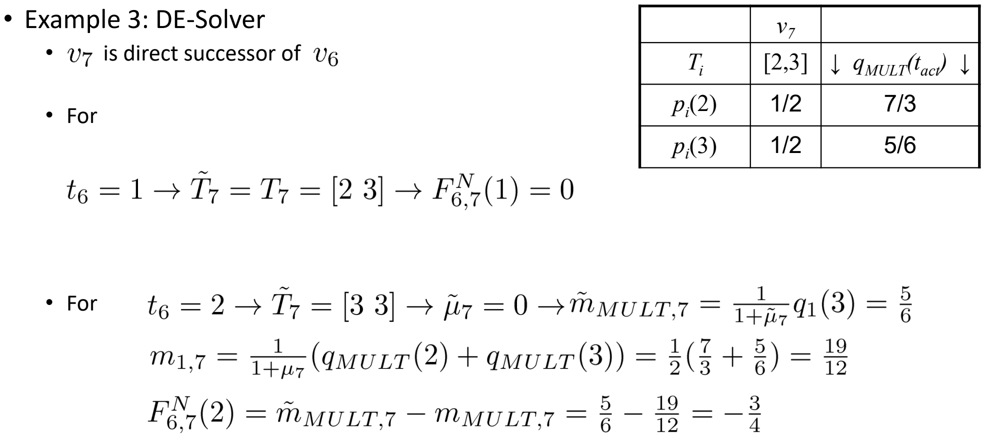
  - New demand for resources

$$\tilde{m}_{k,i} = \frac{1}{\tilde{\mu}_i + 1} \cdot \sum_{t_p = \tilde{t}_i^{\text{ASAP}}}^{\tilde{t}_i^{\text{ALAP}}} q_{k(t_p)}$$

Predecessor and successor forces

$$F_{i,j}^{V,N}(t_{\text{act}}) = \tilde{m}_{k,j} - m_{k,j}$$

- Change of mean demand for resources of type k for predecessor and successor nodes.
- $F_{i,j}^{V,N}(t_{\text{act}}) > 0$: by setting the start time of node v_i to t_{act}, the successor/predecessor node v_j can only be scheduled in clock cycles with a higher demand for resources of type k. Push v_i away from t_{act} by a positive predecessor/successor force.
- $F_{i,j}^{V,N}(t_{\text{act}}) < 0$: The other way around; pull v_j to t_{act} with a negative predecessor/successor force.



- Example 3: DE-Solver
  - v_7 is direct successor of v_6
  - For

$$t_6 = 1 \rightarrow \tilde{T}_7 = T_7 = [2\ 3] \rightarrow F_{6,7}^N(1) = 0$$

	v_7	
T_i	[2,3]	↓ $q_{MULT}(t_{act})$ ↓
$p_i(2)$	1/2	7/3
$p_i(3)$	1/2	5/6

- For $t_6 = 2 \rightarrow \tilde{T}_7 = [3\ 3] \rightarrow \tilde{\mu}_7 = 0 \rightarrow \tilde{m}_{MULT,7} = \frac{1}{1+\tilde{\mu}_7} q_1(3) = \frac{5}{6}$

$$m_{1,7} = \frac{1}{1+\mu_7}(q_{MULT}(2) + q_{MULT}(3)) = \frac{1}{2}(\frac{7}{3} + \frac{5}{6}) = \frac{19}{12}$$

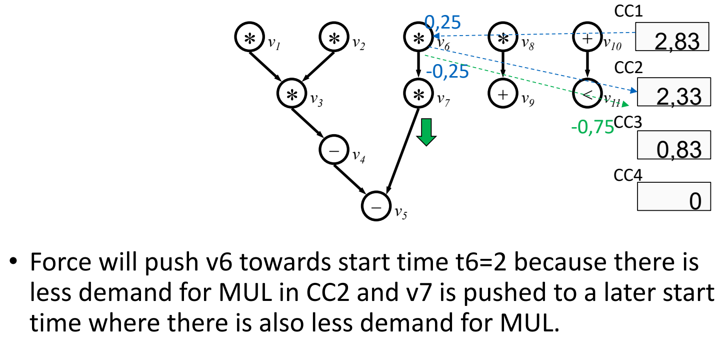
$$F_{6,7}^N(2) = \tilde{m}_{MULT,7} - m_{MULT,7} = \frac{5}{6} - \frac{19}{12} = -\frac{3}{4}$$

Figure 4.21: Example: Predecessor and Successor forces

Total Force

$$F_i^T(t_{\text{act}}) = F_i^{S(t_{\text{act}})} + \sum_{\{j:(\text{op}_i,\text{op}_j)\in E\}} F_{i,j}^N(t_{\text{act}}) + \sum_{\{j:(\text{op}_j,\text{op}_i)\in E\}} F_{i,j}^V(t_{\text{act}})$$

- Sum of self-force, predecessor forces, and successor forces.
- To minimize resources, select starting times with minimal force, which should lead to a minimal mean demand of resources of all types for the schedule.



- Force will push v6 towards start time t6=2 because there is less demand for MUL in CC2 and v7 is pushed to a later start time where there is also less demand for MUL.

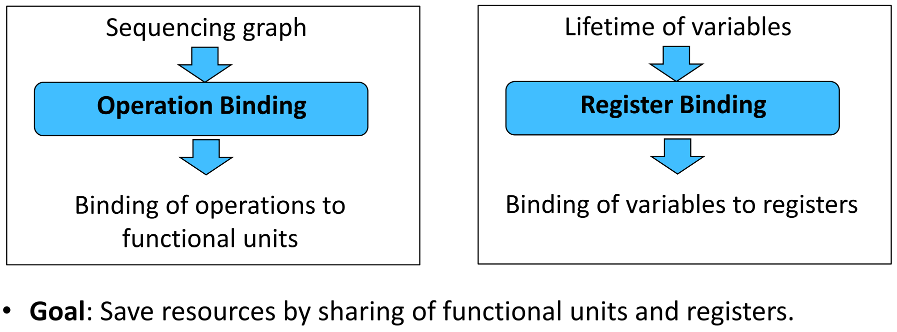
Figure 4.22: Example: Force-directed Scheduling

```
FDS(G_s,u(V,E),Lambda_max) {
    repeat {
        Compute the time frame for all nodes.
        Compute the distribution for the starting time for all nodes and all mean
demands.
        Compute the total force for each node.
        Select the node with the minimal force and assign the starting time to it.
    } until a starting time has been assigned to all nodes.
    return (t-FDS)
}
```

Code 4.6: Force-Directed Scheduling Algorithm

4.3 Binding RTL and FSM Generation

4.3.1 The Binding Task


Sequencing graph
Operation Binding
Binding of operations to functional units
Lifetime of variables
Register Binding
Binding of variables to registers

- **Goal**: Save resources by sharing of functional units and registers.

Figure 4.23: Binding Tasks

- Concurrent operations can be scheduled to be executed in parallel.
  - These cannot be bound to the same FU (in the same clock cycle).

4.3.2 Graph Coloring

- The **cover** of a set S is a set of subsets of S such that their union is equal to S.
- The **partition** of S is a cover of S such that all subsets in the cover are disjoint.
- A **clique** of an undirected graph G is a subset of the nodes V in which all nodes are fully connected with each other.
  - **clique-cover**, **clique-partition** as above
- $\chi(G)$… *chromatic number* is the minimal coloring.

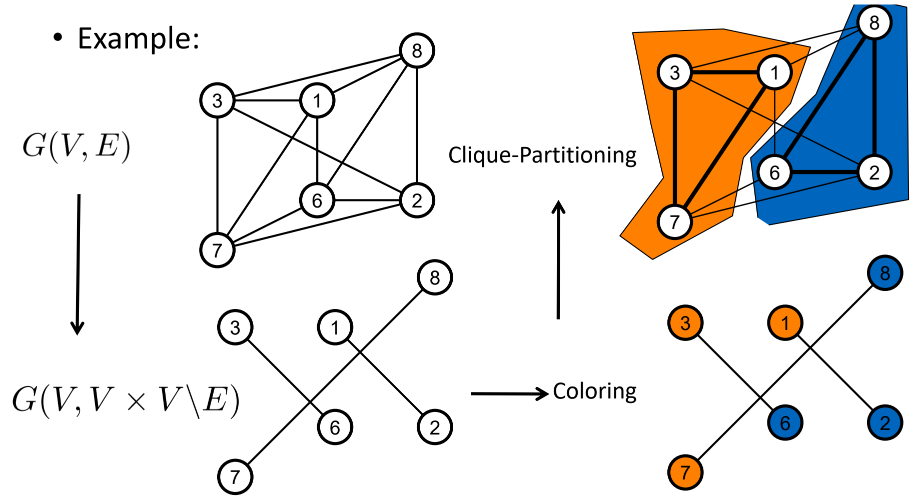


Figure 4.24: Examples: Graph Coloring

4.3.3 Operation Binding

- type of functional units: $k_r \in K$
- set of functional unit types: $K = \{\mathrm{ALU}, \mathrm{MUL}, ...\}$
- A functional unit is defined by type and index: $(\mathrm{ALU}, 1)$
- An operation is executable on a FU if the operation is supported by the unit.

k	+	-	>	<	*	NOP	LOOP	BR	CALL
ALU	x	x	x	x					
MULT					x				
NFU						x	x	x	x

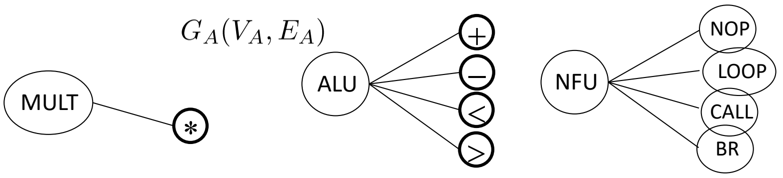


Figure 4.25: FU Executability

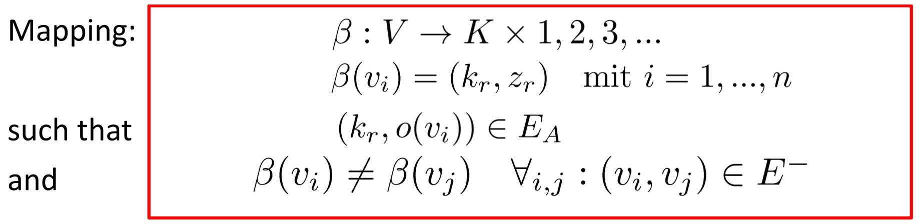
Operation-Compatibility-Graph

- Operations are compatible if and only if:
  - they can be executed on the same FU, $\land$
  - if either one operation always starts after the other has finished, $\lor$
    - they are on alternative paths in the control flow
- operation-compatibility-graph: $G^+_K(V, E^+)$
  - An edge connects compatible operations E^+.
  - Cliques in the compatibility graph are sets of operations that can be bound to the same FU.

Operation-Conflict-Graph

- Operations are incompatible
  - if they must be executed on a different FU, or
  - if they are executed in parallel due to the schedule.
- operation-conflict-graph $G^-_{K(V,E^-)}$
  - Edges connect incompatible operations.
  - *Interval graphs* can be seen as a special case of a conflict-graph.
    - Overlaps of intervals produce conflicts.
- The conflict graph and the compatibility graph are complementary.

Operation Binding



Mapping: $\beta : V \rightarrow K \times 1, 2, 3, \ldots$

$\beta(v_i) = (k_r, z_r) \quad \text{mit } i = 1, \ldots, n$

such that $(k_r, o(v_i)) \in E_A$

and $\beta(v_i) \neq \beta(v_j) \quad \forall_{i,j} : (v_i, v_j) \in E^-$

Figure 4.26: Operation Binding

- Each operation is bound to one FU.
- The operation must be executable on the FU z_r.
  - $\beta(v_i) = (k_r, z_r) \qquad \text{with } i = 1, \ldots, n$
- Two operations that are bound to the same FU must be compatible.
  - $(k_r, o(v_i)) \in E_A$
  - $\beta(v_i) \neq \beta(v_j) \quad \forall i, j : (v_i, v_j) \in E^-$

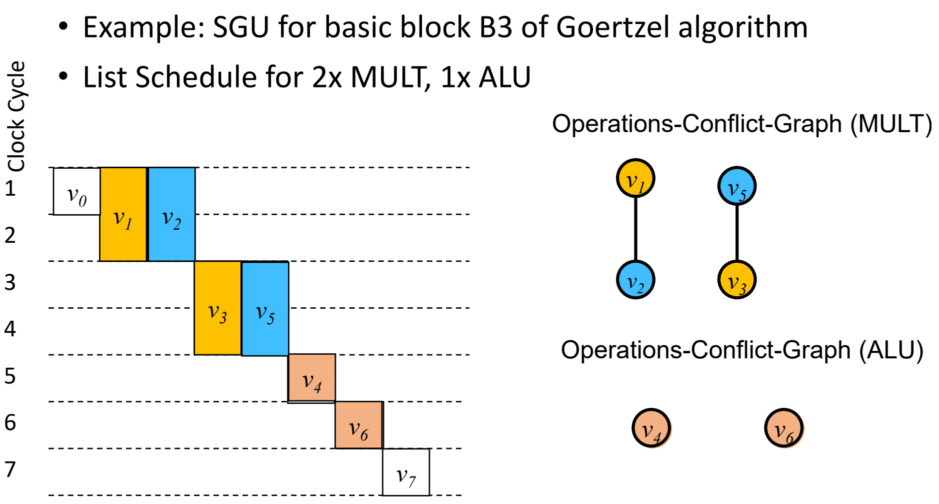
Left-Edge-Algorithm

- To color an interval graph with the minimum number of colors.
- worst-case complexity $O(|V| \cdot \log(|V|))$

```
LeftEdge(G_I(V_I,E_I)) {
    Sort intervals I[i]=[l[i] r[i]] in list L by increasing l[i]
    Set color number c=0;
    repeat {
        Go to the start of List L
        Set S={}
        Set r_act=0;
        repeat {
            Select next interval I[s]=[l[s] r[s]] from List L
            if (l[s] >= r_act) {
                Insert I[s] in set S
                Set r_act=r[s]
                Delete I[s] from List L
            }
        until (End of List L is reached)
        Assign color c to all intervals in set S
        Select next color number: c=c+1;
    }
    until (List L is empty)
}
```

Code 4.7: Left-Edge-Algorithm

- Input:
  - Given Schedule
  - Interval graph of the execution times for the operations.
  - Operation-conflict graph for each functional unit type.
- The Left-Edge Algorithm is applied for each functional unit type with a disjoint set of colors.
- Output: Binding of operations to the functional units.



- Example: SGU for basic block B3 of Goertzel algorithm
- List Schedule for 2x MULT, 1x ALU

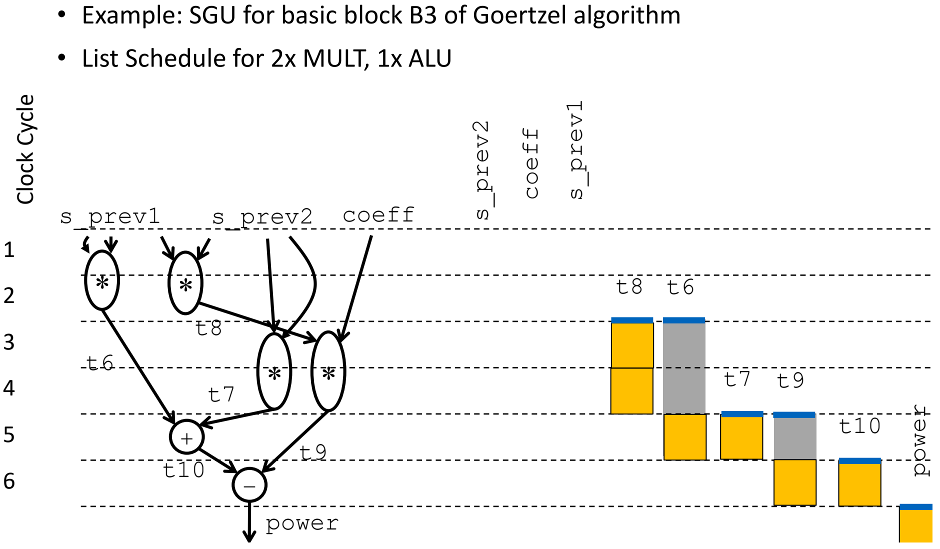
Figure 4.27: Operation Binding with Left-Edge Algorithm

4.3.4 Register Binding

- **Live variables**: Variables are live from their generation until their last use (lifetime). After their last use, they are considered dead.
- **Register Sharing**: The number of registers is decreased by storing variables with non-overlapping lifetimes in the same register.
- Variables are incompatible, if their lifetimes overlap.
- Register-conflict graph:
  - Variables are the nodes.
  - Incompatible variables are connected by an edge.
- Interval graph
  - The Left-Edge Algorithm can compute a minimal coloring.
  - The coloring defines cliques of variables that can share one register.

Local Live Variable Analysis

- First, run *global live variable analysis*.



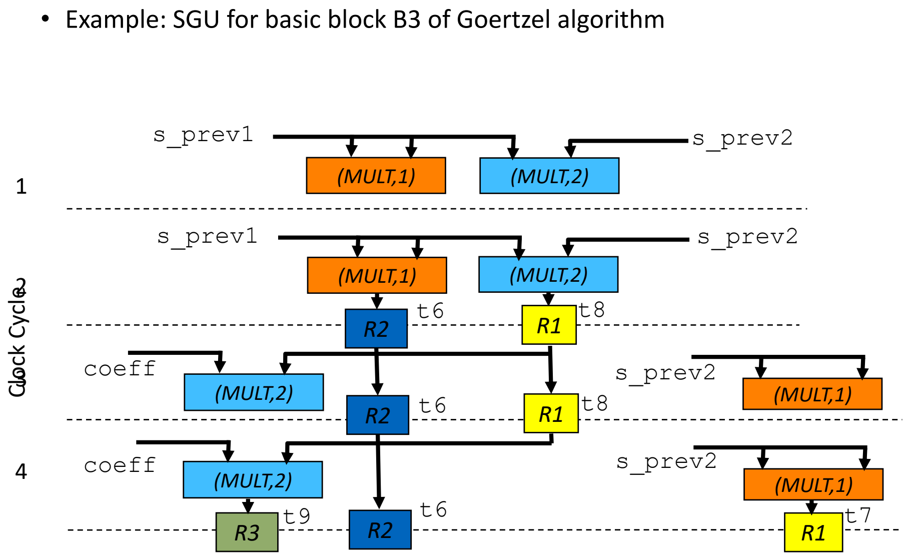
- Example: SGU for basic block B3 of Goertzel algorithm
- List Schedule for 2x MULT, 1x ALU

Figure 4.28: Local Live Variable Analysis: Goertzel Algorithm

4.3.5 Datapath Generation

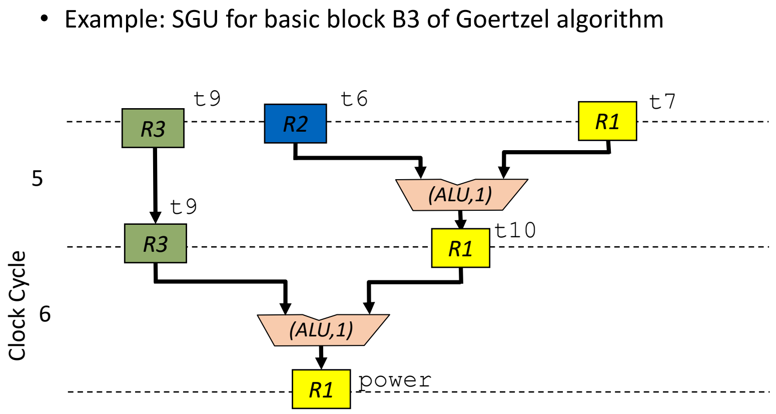
Data Flow Graph with Schedule and Binding

- Shows the data flow between FUs and registers.
- The number of multiplexers can be determined.
- The required control signals can be determined.
- Allows for the generation of an RTL description of the datapath.



- Example: SGU for basic block B3 of Goertzel algorithm

Figure 4.29: Data Flow Graph with Scheduling and Binding: Goertzel Algorithm



- Example: SGU for basic block B3 of Goertzel algorithm

Figure 4.30: Data Flow Graph with Scheduling and Binding (SGU): Goertzel Algorithm

4.3.6 Control Unit Generation

- Generates control signals for:
  - data flow
  - control flow
  - the interface to the hardware module
- Processes status signals from:
  - the datapath
  - the interface

Finite State Machine (FSM)

- Is a 6-tuple:
  - Input alphabet: I
  - Output alphabet: O
  - Set of states: X

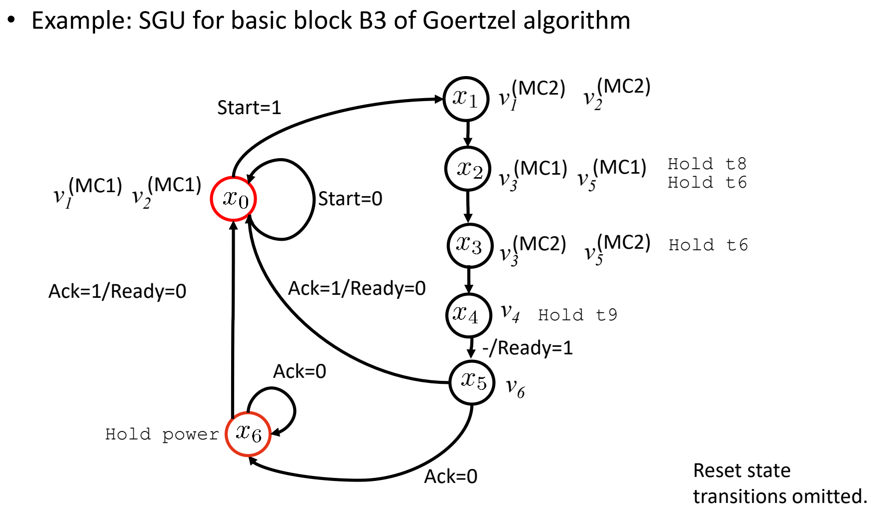
  - Set of starting states: $R \subseteq X$
  - Set of transition relations: $f \subseteq (X \times I \times X)$
  - Output relation: $g \subseteq (X \times O \times X)$
- A FSM is deterministic if and only if there is a single starting state ($|R| = 1$) and state transitions and output relations are functions: $f : (X \times I) \to X \quad g : (X \times O) \to O$
  - An FSM is *completely specified* if and only if these functions are completely defined.

Activation Signals for Operations

- The activation signals of an operation are all control signals required for its execution.
- Activation signals may include:
  - The multiplexer control signals to establish a connection between input registers, functional unit, and output register.
  - Register-enable signal to write the result to the output register.
  - ALU control signals to select the correct operation.
- A read activation signal is used to enter input values into the register.
- A hold signal is used to keep the value of a variable in a register.

FSM with Data Specification

- The FSM with data specification describes the schedule and the control flow of the datapath and the HW module interface.
- The operations of the datapath are assigned to the states of the FSMD.
- The FSMD has one state transition for each clock cycle.
- Transitions may depend on status signals from the datapath or the interface.



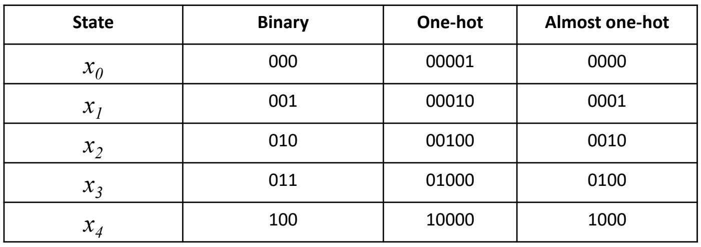
- Example: SGU for basic block B3 of Goertzel algorithm

Reset state transitions omitted.

Figure 4.31: FSM with Data Specification (FSMD)

State Assignment

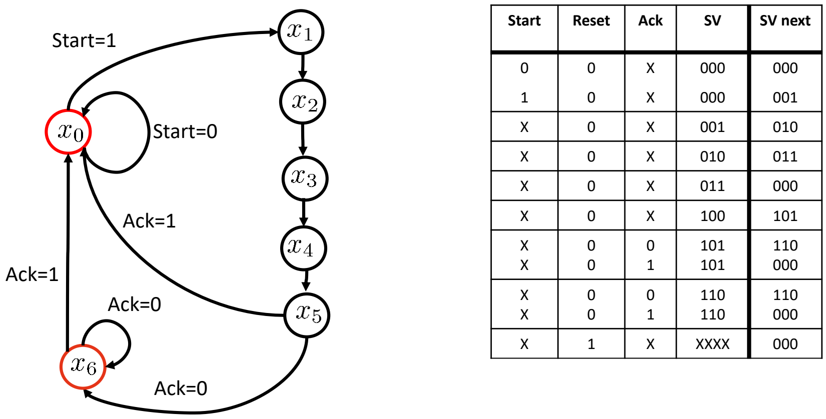
- Number of state variables: n_{bit}
- Number of possible states: $|X| = 2^{n_{\text{bit}}}$
- Number of state variables: $n_{\text{bit}} = \lceil \log_2 |X| \rceil$



State	Binary	One-hot	Almost one-hot
x_0	000	00001	0000
x_1	001	00010	0001
x_2	010	00100	0010
x_3	011	01000	0100
x_4	100	10000	1000

Figure 4.32: State Assignment - State Coding

Next-State and Output Logic



Start	Reset	Ack	SV	SV next
0	0	X	000	000
1	0	X	000	001
X	0	X	001	010
X	0	X	010	011
X	0	X	011	000
X	0	X	100	101
X X	0 0	0 1	101 101	110 000
X X	0 0	0 1	110 110	110 000
X	1	X	XXXX	000

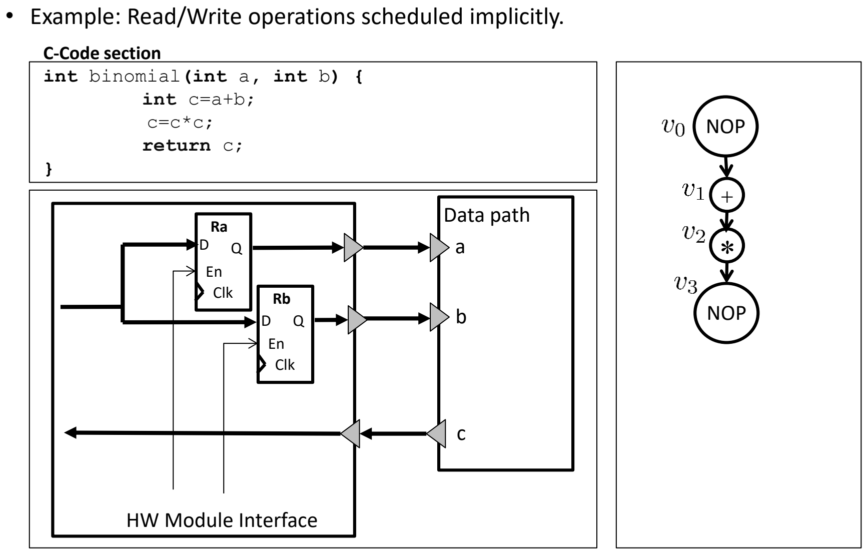
Figure 4.33: Next-State and Output Logic for Goertzel Algorithm

4.4 Loop and IO Optimization

4.4.1 I/O Scheduling

Register Interface

- Data is stored in registers in the interface (access via bus).
- Read/write operations can be scheduled concurrently with zero delay.



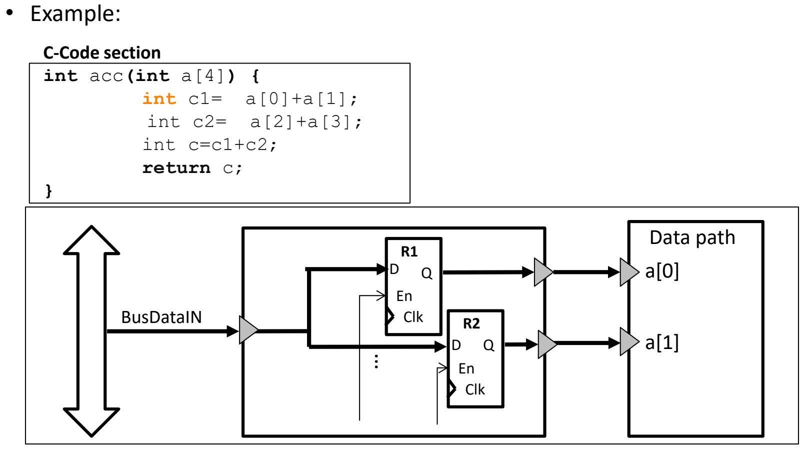
- Example: Read/Write operations scheduled implicitly.

C-Code section

```
int binomial(int a, int b) {
        int c=a+b;
         c=c*c;
        return c;
}
```

Figure 4.34: Register Interface

Array Register Interface



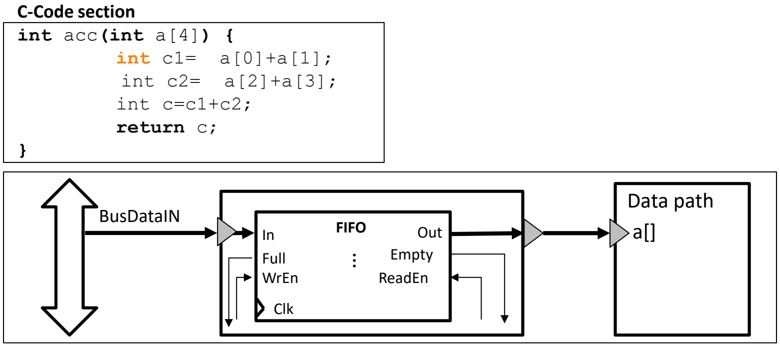
- Example:

C-Code section

```
int acc(int a[4]) {
        int c1=  a[0]+a[1];
         int c2=  a[2]+a[3];
        int c=c1+c2;
        return c;
}
```

Figure 4.35: Array Register Interface

Array FIFO Interface



C-Code section

```
int acc(int a[4]) {
        int c1=  a[0]+a[1];
         int c2=  a[2]+a[3];
        int c=c1+c2;
        return c;
}
```

Figure 4.36: Array FIFO Interface

- array is stored in FIFO
- stalls if FIFO is empty

SRAM Buffers

- On-chip buffers using SRAM cells (different from flip-flops)
- Single-port SRAM
  - Only one port to read or write
- Dual-port SRAM
  - Two ports to read or write
  - Cannot read/write the same location on both ports at the same time
  - True dual-port SRAM: Can read the same location on both ports; writes or read/write still need to be arbitrated.
- Timing
  - Returns data either in the same (zero-delay) or the next clock cycle (pipelined).

Ping Pong Array Memory Interface

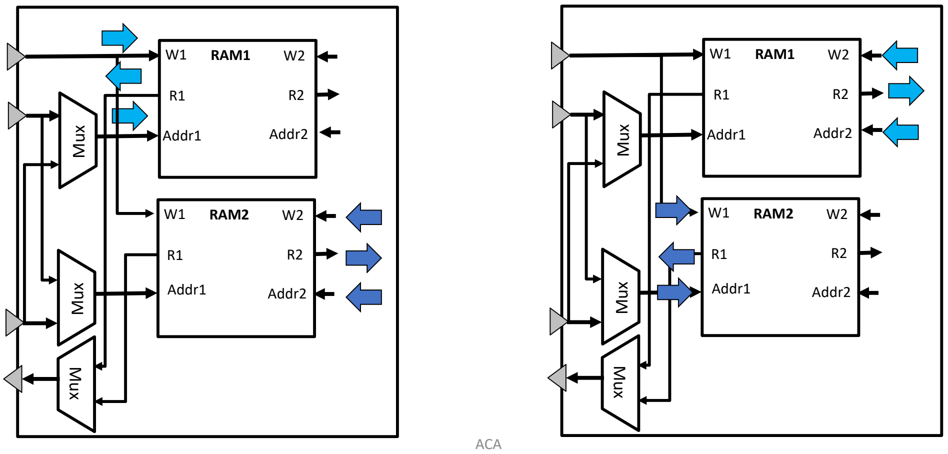


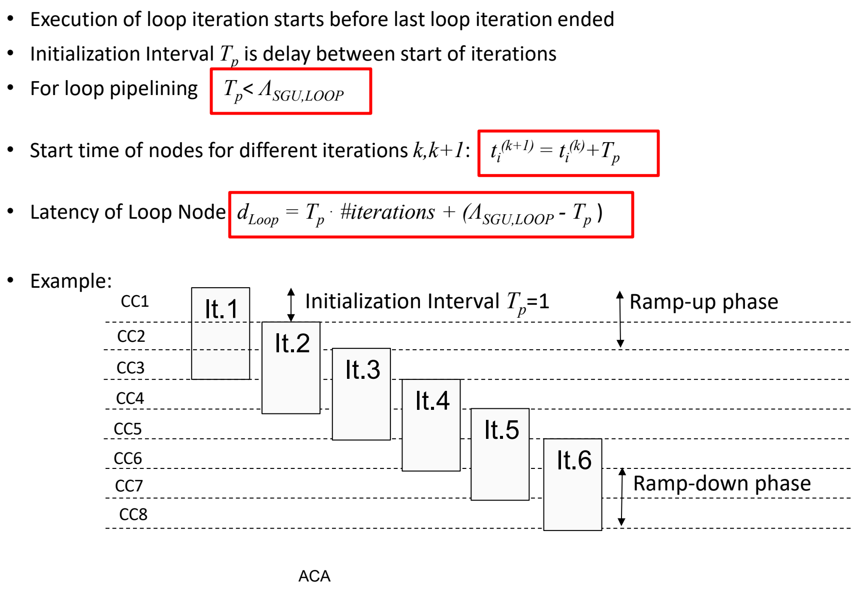
Figure 4.37: Ping Pong Array Memory Interface

- One RAM for input, one RAM for output.
- Switch/overlap between phases (ping-pong scheme).
- All ports can be kept busy if read/write operations from two different execution runs overlap (high utilization of memory ports).

4.4.2 Control Flow and Loop Scheduling

Combining Schedules of SGUs

- Algorithms find a schedule for a single sequencing graph unit (SGU).
- Hierarchy nodes (CALL, BR, LOOP) represent an SGU.
- The schedule for a complete sequencing graph is found by:
  - Compute the schedule for the SGU on the lowest level of the hierarchy first.
  - Extract the execution time of hierarchy nodes from the latency of the schedule of their corresponding SGUs.
  - Schedule the top-level SGU with the hierarchy node.
  - Shift the start time of the schedule of the lower-level SGU to the start time of the corresponding hierarchy node.
- The schedule can be data-dependent or independent.



- Execution of loop iteration starts before last loop iteration ended
- Initialization Interval T_p is delay between start of iterations
- For loop pipelining $\boxed{T_p < \Lambda_{SGU,LOOP}}$
- Start time of nodes for different iterations $k,k+1$: $\boxed{t_i^{(k+1)} = t_i^{(k)} + T_p}$
- Latency of Loop Node $\boxed{d_{Loop} = T_p \cdot \#iterations + (\Lambda_{SGU,LOOP} - T_p)}$
- Example:

Figure 4.38: Loop Pipelining

5 Block E

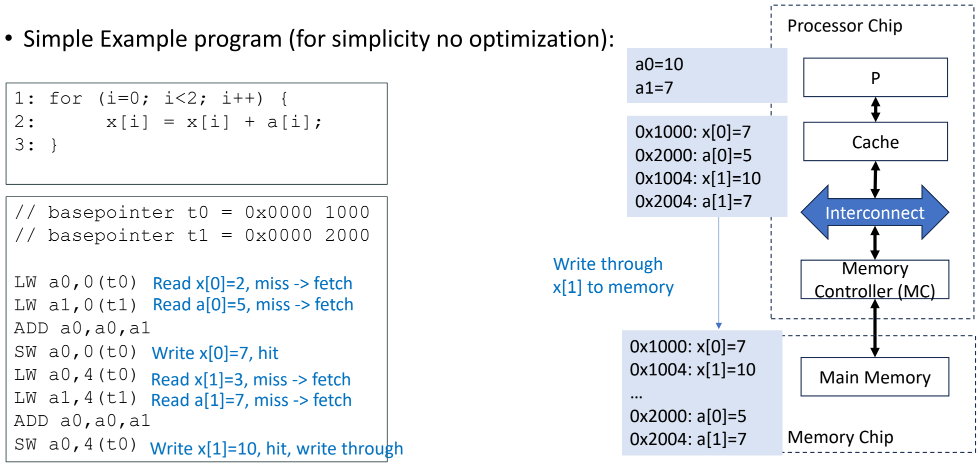
5.1 Multi-Core Challenges

5.1.1 Cache Recap

i Note

Please take a look at Section 3.4

5.1.2 Caching with Write Through



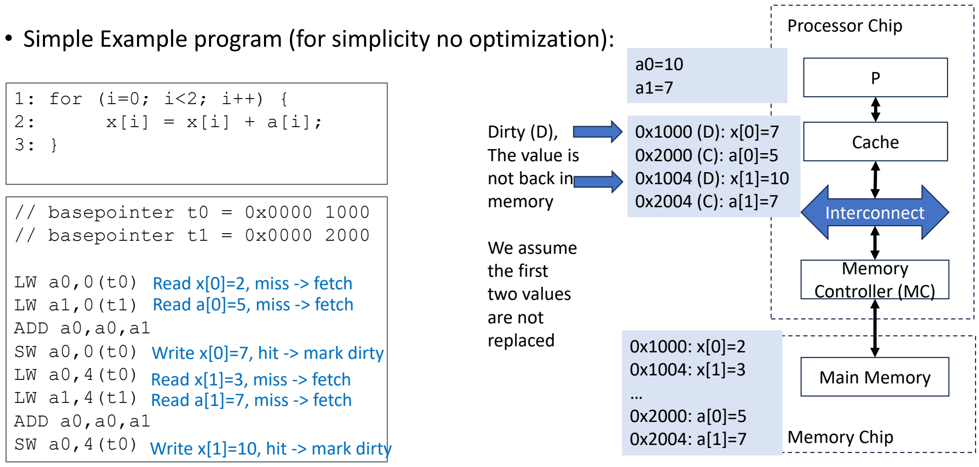
- Simple Example program (for simplicity no optimization):

```
for (i=0; i<2; i++) {
      x[i] = x[i] + a[i];
}
```

```
// basepointer t0 = 0x0000 1000
// basepointer t1 = 0x0000 2000

LW a0,0(t0)  Read x[0]=2, miss -> fetch
LW a1,0(t1)  Read a[0]=5, miss -> fetch
ADD a0,a0,a1
SW a0,0(t0)  Write x[0]=7, hit
LW a0,4(t0)  Read x[1]=3, miss -> fetch
LW a1,4(t1)  Read a[1]=7, miss -> fetch
ADD a0,a0,a1
SW a0,4(t0)  Write x[1]=10, hit, write through
```

Figure 5.1: Caching with Write Through



- Simple Example program (for simplicity no optimization):

```
for (i=0; i<2; i++) {
      x[i] = x[i] + a[i];
}
```

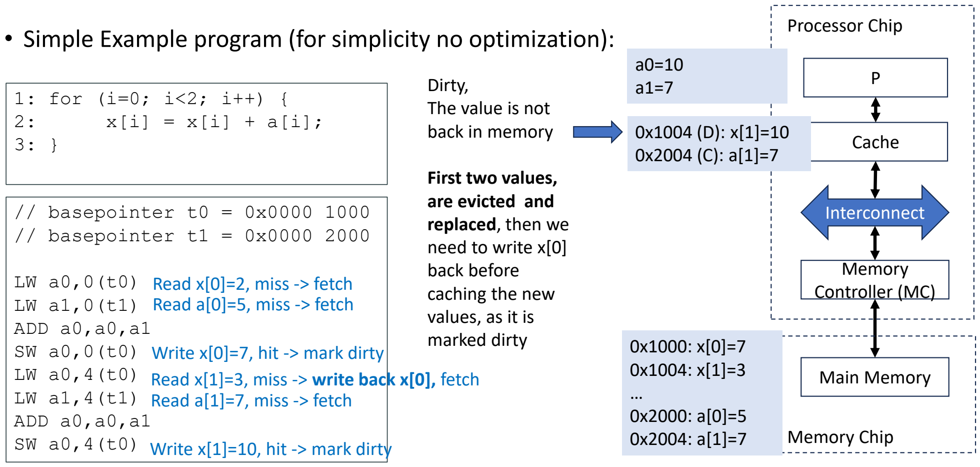
```
// basepointer t0 = 0x0000 1000
// basepointer t1 = 0x0000 2000

LW a0,0(t0)  Read x[0]=2, miss -> fetch
LW a1,0(t1)  Read a[0]=5, miss -> fetch
ADD a0,a0,a1
SW a0,0(t0)  Write x[0]=7, hit -> mark dirty
LW a0,4(t0)  Read x[1]=3, miss -> fetch
LW a1,4(t1)  Read a[1]=7, miss -> fetch
ADD a0,a0,a1
SW a0,4(t0)  Write x[1]=10, hit -> mark dirty
```

Dirty (D), The value is not back in memory

We assume the first two values are not replaced

Figure 5.2: Caching with Write-back (Copy back), no eviction



- Simple Example program (for simplicity no optimization):

```
1: for (i=0; i<2; i++) {
2:      x[i] = x[i] + a[i];
3: }
```

```
// basepointer t0 = 0x0000 1000
// basepointer t1 = 0x0000 2000

LW a0,0(t0)  Read x[0]=2, miss -> fetch
LW a1,0(t1)  Read a[0]=5, miss -> fetch
ADD a0,a0,a1
SW a0,0(t0)  Write x[0]=7, hit -> mark dirty
LW a0,4(t0)  Read x[1]=3, miss -> write back x[0], fetch
LW a1,4(t1)  Read a[1]=7, miss -> fetch
ADD a0,a0,a1
SW a0,4(t0)  Write x[1]=10, hit -> mark dirty
```

Dirty, The value is not back in memory

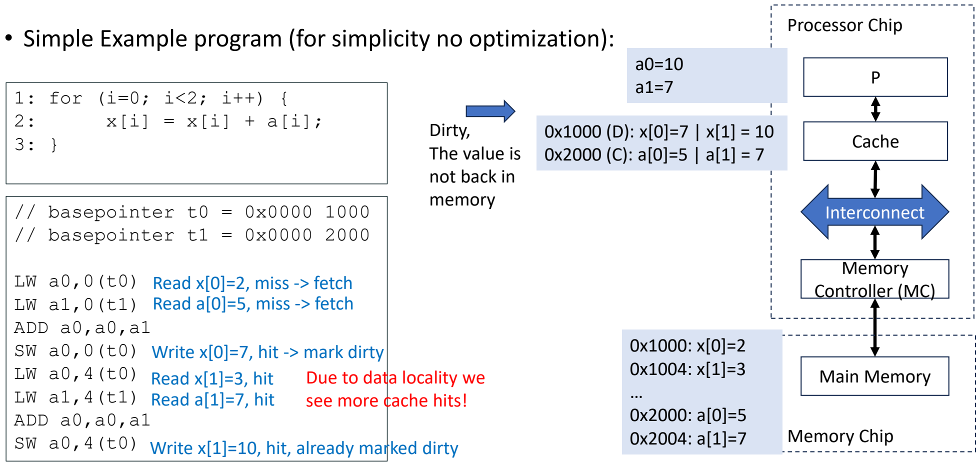
First two values, are evicted and replaced, then we need to write x[0] back before caching the new values, as it is marked dirty

a0=10
a1=7

0x1004 (D): x[1]=10
0x2004 (C): a[1]=7

0x1000: x[0]=7
0x1004: x[1]=3
…
0x2000: a[0]=5
0x2004: a[1]=7

Figure 5.3: Caching with Write-back (Copy back), with eviction



- Simple Example program (for simplicity no optimization):

```
1: for (i=0; i<2; i++) {
2:      x[i] = x[i] + a[i];
3: }
```

```
// basepointer t0 = 0x0000 1000
// basepointer t1 = 0x0000 2000

LW a0,0(t0)  Read x[0]=2, miss -> fetch
LW a1,0(t1)  Read a[0]=5, miss -> fetch
ADD a0,a0,a1
SW a0,0(t0)  Write x[0]=7, hit -> mark dirty
LW a0,4(t0)  Read x[1]=3, hit      Due to data locality we
LW a1,4(t1)  Read a[1]=7, hit      see more cache hits!
ADD a0,a0,a1
SW a0,4(t0)  Write x[1]=10, hit, already marked dirty
```

Dirty, The value is not back in memory

a0=10
a1=7

0x1000 (D): x[0]=7 | x[1] = 10
0x2000 (C): a[0]=5 | a[1] = 7

0x1000: x[0]=2
0x1004: x[1]=3
…
0x2000: a[0]=5
0x2004: a[1]=7

Figure 5.4: Caching with block size > 1

5.1.3 Multi-Processors with Shared Memory

Symmetric Multi-Processor (Symmetric Multi-Processor (SMP)) with Shared Cache

- several processor cores on the chip
- individual private caches (L1)
- shared caches (L2) and shared main memory
- Single-core multi-threading: All threads run on same processor core
- SMP: Threads executed on several processor cores (P0,P1,...) in parallel, presenting three challenges:
  - The Cache Coherency Problem
  - Memory Consistency Problem
  - Synchronization Problem

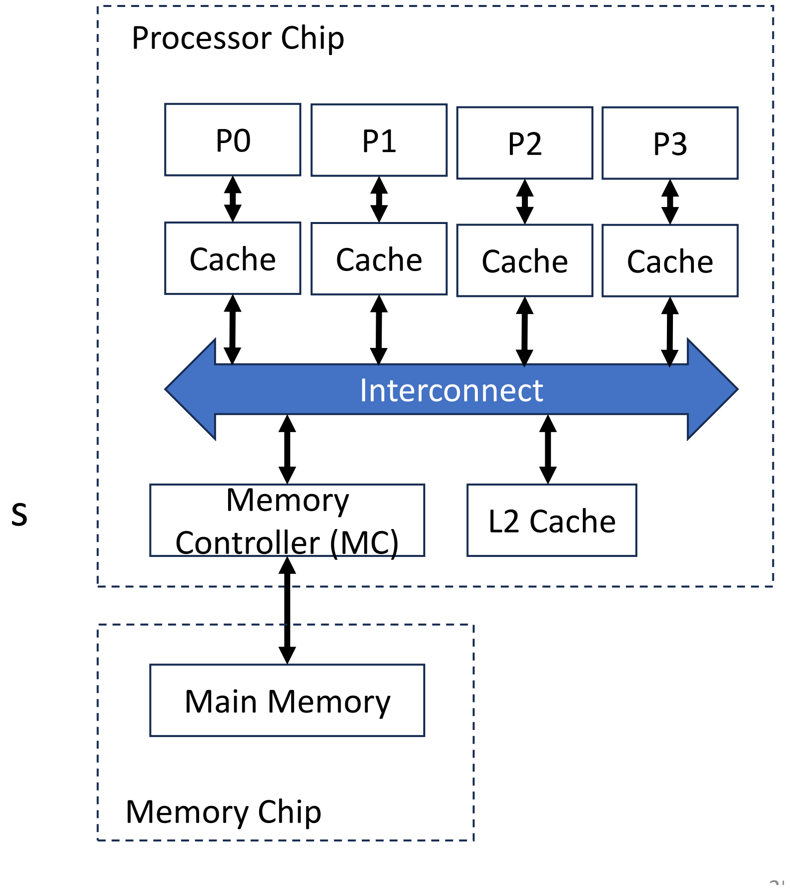
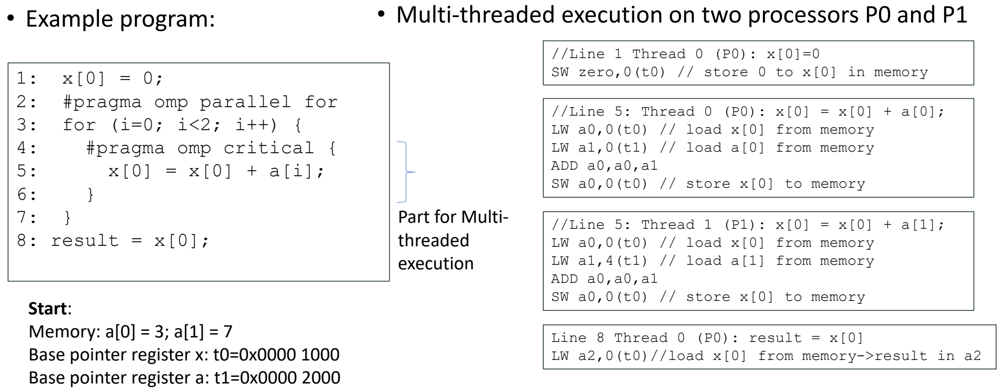


Figure 5.5: Shared Cache

5.1.4 The Cache Coherency Problem



- Example program:

```
1:  x[0] = 0;
2:  #pragma omp parallel for
3:  for (i=0; i<2; i++) {
4:     #pragma omp critical {
5:        x[0] = x[0] + a[i];
6:     }
7:  }
8: result = x[0];
```

Part for Multi-threaded execution

Start:
Memory: a[0] = 3; a[1] = 7
Base pointer register x: t0=0x0000 1000
Base pointer register a: t1=0x0000 2000

- Multi-threaded execution on two processors P0 and P1

```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0) // store 0 to x[0] in memory
```

```
//Line 5: Thread 0 (P0): x[0] = x[0] + a[0];
LW a0,0(t0) // load x[0] from memory
LW a1,0(t1) // load a[0] from memory
ADD a0,a0,a1
SW a0,0(t0) // store x[0] to memory
```

```
//Line 5: Thread 1 (P1): x[0] = x[0] + a[1];
LW a0,0(t0) // load x[0] from memory
LW a1,4(t1) // load a[1] from memory
ADD a0,a0,a1
SW a0,0(t0) // store x[0] to memory
```

```
Line 8 Thread 0 (P0): result = x[0]
LW a2,0(t0)//load x[0] from memory->result in a2
```

Figure 5.6: Cache Coherency Problem Code Example

5.1.5 Multi-threaded Execution without Caches

⚠ **Warning**

Too many graphical slides

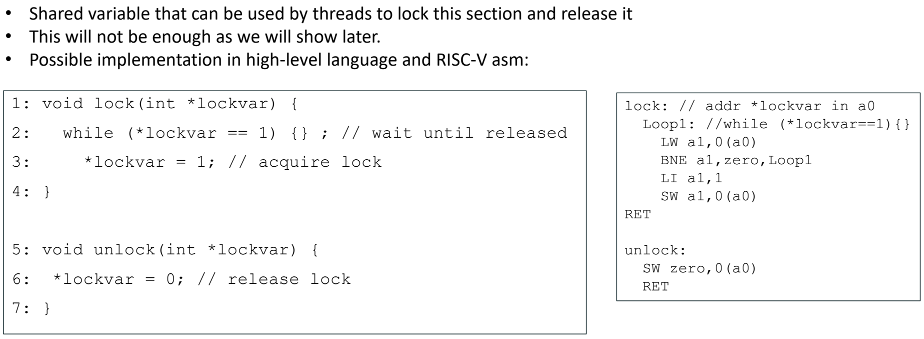
5.1.6 The Memory Consistency Problem

The Memory Consistency Problem for SMP

- Preserving program order on each single processor is insufficient for correct code execution.

- Mechanisms are needed to ensure that accesses of one processor appear to execute in program order to all others.
- Ensuring full program order across processors is expensive in terms of performance.
- Often, processors only enforce partial program ordering.
- Memory consistency model: Types of enforced program ordering by the processor.

5.1.7 The Synchronization Problem



- Shared variable that can be used by threads to lock this section and release it
- This will not be enough as we will show later.
- Possible implementation in high-level language and RISC-V asm:

```
void lock(int *lockvar) {
   while (*lockvar == 1) {} ; // wait until released
     *lockvar = 1; // acquire lock
}

void unlock(int *lockvar) {
  *lockvar = 0; // release lock
}
```

```
lock: // addr *lockvar in a0
  Loop1: //while (*lockvar==1){}
    LW a1,0(a0)
    BNE a1,zero,Loop1
    LI a1,1
    SW a1,0(a0)
RET

unlock:
  SW zero,0(a0)
  RET
```

Figure 5.7: Lock – Variable – High-level and RISC-V Implementation

- A race condition at the instruction level caused both threads to enter the critical section.
- Programmers need other primitives to achieve synchronization.
- Software solution: **Peterson’s algorithm**
  - Can achieve mutual exclusion
  - Suffers from a lack of scalability for many threads.
- HW support for synchronization
  - Reduce synchronization overhead
  - Enable scalable synchronization for many parallel threads

5.2 Cache Coherency

5.2.1 Cache Controllers

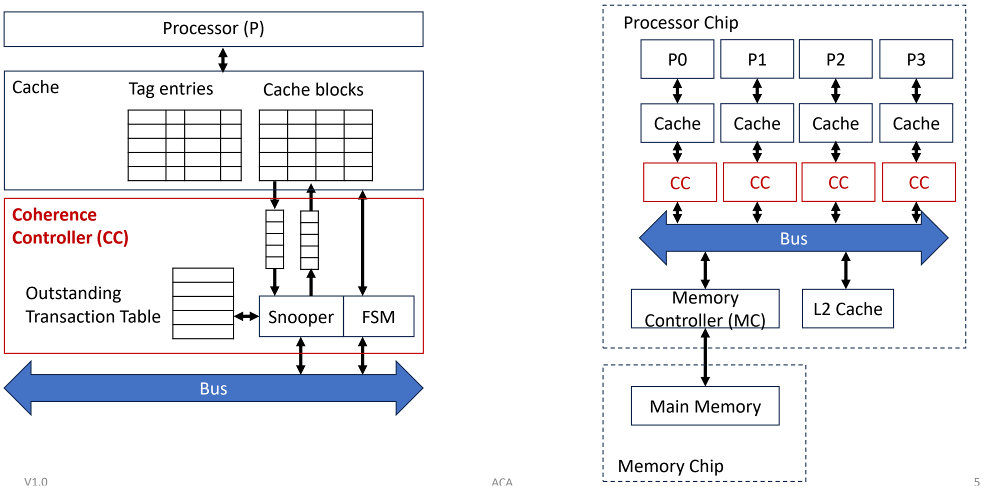


Figure 5.8: Coherence Controller

Outstanding transaction table & Snooper

- Outstanding transaction table
  - In the split transaction bus, multiple requests to different addresses can be placed on the bus even when the oldest request has not obtained its data.

- keeps track of bus transactions that have not completed.

- Bus snooper.
  - Snoops each bus transaction
  - checks the cache tag array to see if it has the block that is involved in the transaction
  - checks the current state of the block (if the block is found)
  - changes the state of the block
  - New state of block -> a finite state machine (FSM) that implements the cache coherence protocol
  - Data that is sent out is placed in a queue called the write back buffer

5.2.2 Coherence Protocol for Write Through Caches

Coherence Protocol for Write Through Caches - Requests

- The simplest cache coherence protocol: write-through caches.

- Requests from the processor side, as well as from the bus side are snooped by the snooper.

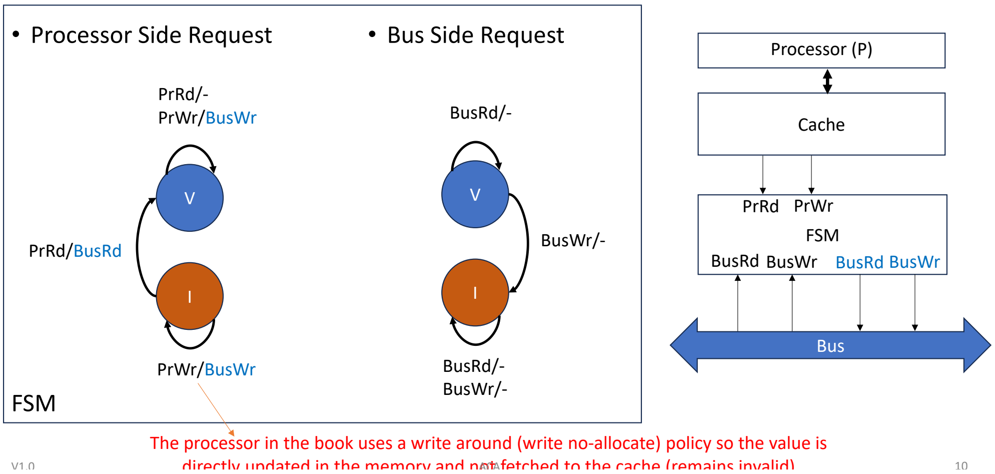
- Processor requests to the cache include:
  - 1.PrRd: processor-side request to read a cache block.
  - 2.PrWr: processor-side request to write to a cache block.

- Snooped requests to the cache include:
  - 1.BusRd: snooped request that indicates there is a read request to a block made by another processor.
  - 2.BusWr: snooped request that indicates there is a write request to a block made by another processor. In the case of a write-through cache, the BusWr is a write-through to the main memory performed by another processor.

Coherence Protocol for Write Through Caches – Cache Block States

Each cache block has an associated state which can have one of the following values:

1. Valid (V): the cache block is valid and clean, meaning that the cached value is the same with that in the lower-level memory component (in this case the main memory).
2. Invalid (I): the cache block is invalid. Accesses to this cache block will generate cache misses.



The processor in the book uses a write around (write no-allocate) policy so the value is directly updated in the memory and not fetched to the cache (remains invalid)

Figure 5.9: Snooper FSM

5.2.3 MSI Protocol with Write Back Caches

In the MSI protocol, processor requests to the cache include:

1. PrRd: processor-side request to read from a cache block.
2. PrWr: processor-side request to write to a cache block.

Bus-side requests include:

1. BusRd: snooped request that indicates there is a read request to a cache block made by another processor.

2. BusRdX: snooped request that indicates there is a read-exclusive (write) request to a cache block made by another processor which does not already have the block.
3. Flush: snooped request that indicates that an entire cache block is written back to the main memory by another processor.

Cache Block States

Each cache block has an associated state which can have one of the following values:

1. **Modified (M)**: the cache block is valid in only one cache, and the value is (likely) different from the one in the main memory. This state extends the meaning of the dirty state in a write-back cache for a single-processor system, except that now it also implies exclusive ownership. Whereas dirty means the cached value is potentially different from the value in the main memory, modified means both the cached value is potentially different from the value in the main memory, and it is cached only in one location.
2. **Shared (S)**: the cache block is valid, potentially shared by multiple processors, and is clean (the value is the same as the one in the main memory). The shared state is similar to the valid state in the coherence protocol for write through caches.
3. **Invalid (I)**: the cache block is invalid (either not cached, or cached but outdated).

	Read Permission	**Write Permission**
Modified State (M)	✅	✅
Shared State (S)	✅	❌
Invalid State (I)	❌	❌

Table 5.5: MSI Protocol Permissions

- **Intervention**: downgrade to S state
- **Invalidation**: downgrade to I state

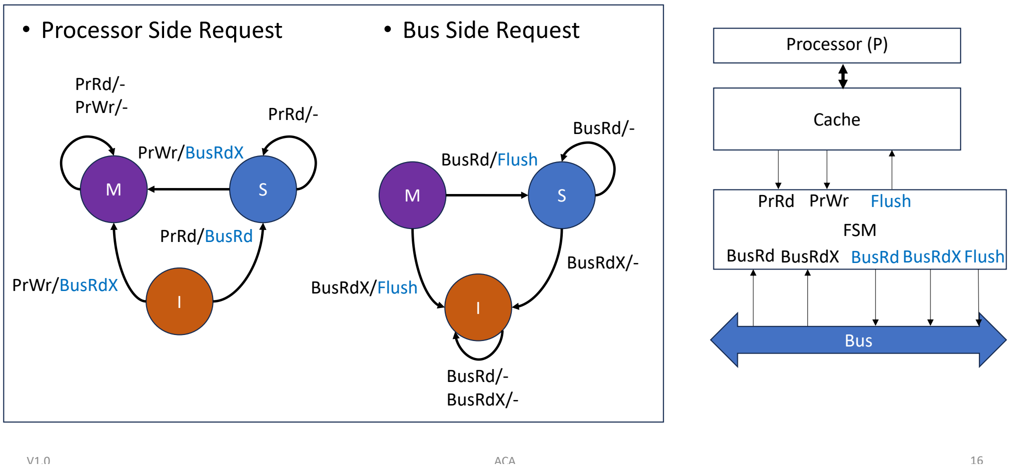
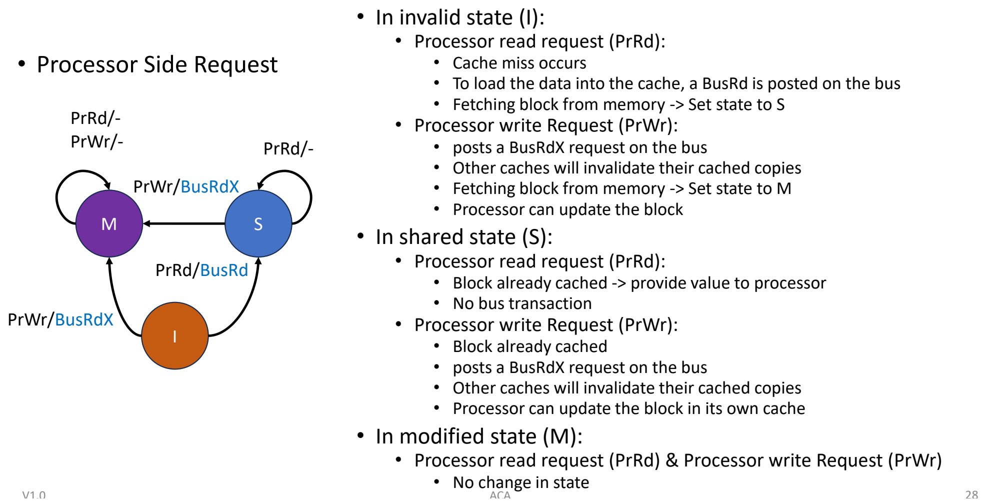


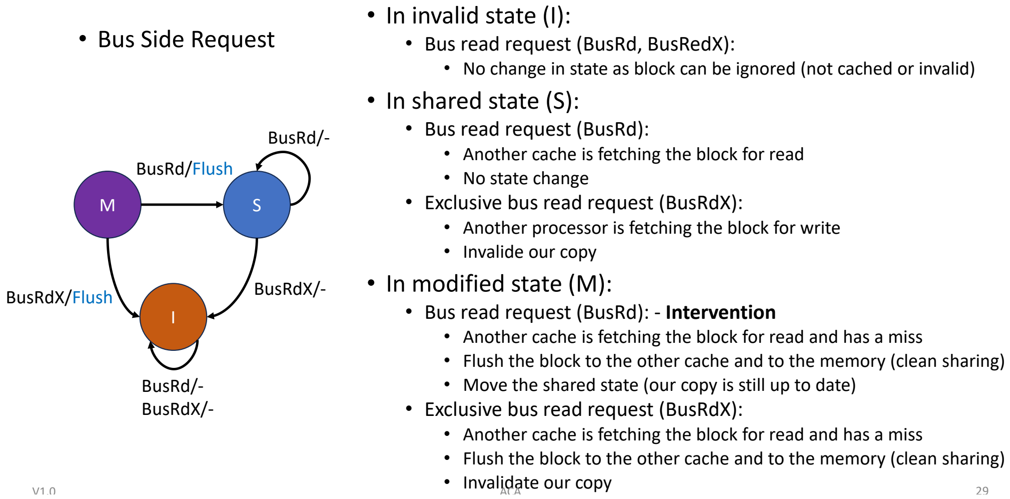
Figure 5.10: MSI Protocol - Snooper FSM - Snooper FSM



- Processor Side Request

- In invalid state (I):
  - Processor read request (PrRd):
    - Cache miss occurs
    - To load the data into the cache, a BusRd is posted on the bus
    - Fetching block from memory -> Set state to S
  - Processor write Request (PrWr):
    - posts a BusRdX request on the bus
    - Other caches will invalidate their cached copies
    - Fetching block from memory -> Set state to M
    - Processor can update the block
- In shared state (S):
  - Processor read request (PrRd):
    - Block already cached -> provide value to processor
    - No bus transaction
  - Processor write Request (PrWr):
    - Block already cached
    - posts a BusRdX request on the bus
    - Other caches will invalidate their cached copies
    - Processor can update the block in its own cache
- In modified state (M):
  - Processor read request (PrRd) & Processor write Request (PrWr)
    - No change in state

Figure 5.11: MSI Protocol - Snooper FSM - Processor Side Request



- Bus Side Request

- In invalid state (I):
  - Bus read request (BusRd, BusRedX):
    - No change in state as block can be ignored (not cached or invalid)
- In shared state (S):
  - Bus read request (BusRd):
    - Another cache is fetching the block for read
    - No state change
  - Exclusive bus read request (BusRdX):
    - Another processor is fetching the block for write
    - Invalide our copy
- In modified state (M):
  - Bus read request (BusRd): - **Intervention**
    - Another cache is fetching the block for read and has a miss
    - Flush the block to the other cache and to the memory (clean sharing)
    - Move the shared state (our copy is still up to date)
  - Exclusive bus read request (BusRdX):
    - Another cache is fetching the block for read and has a miss
    - Flush the block to the other cache and to the memory (clean sharing)
    - Invalidate our copy

Figure 5.12: MSI Protocol - Snooper FSM - Bus Side Request

5.2.4 MESI Protocol with Write-Back Caches

In the MESI protocol, processor requests to the cache include:

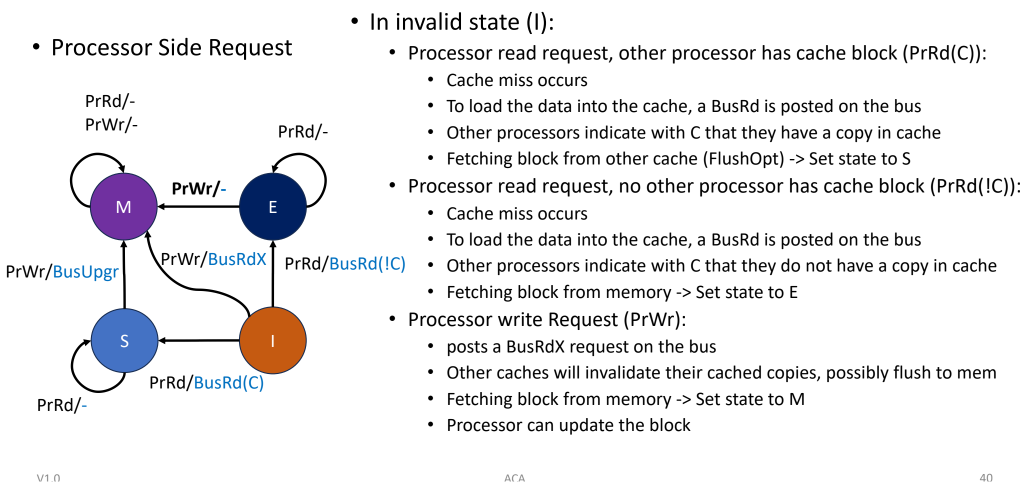
1. PrRd: processor-side request to read from a cache block.
2. PrWr: processor-side request to write to a cache block.

Bus-side requests include:

1. BusRd: snooped request that indicates there is a read request to a cache block made by another processor.
2. BusRdX: snooped request that indicates there is a read-exclusive (write) request to a cache block made by another processor which does not already have the block.
3. Flush: snooped request that indicates that an entire cache block is written back to the main memory by another processor.
4. FlushOpt: snooped request that indicates an entire cache block is posted on the bus in order to supply it to another processor. We refer to such an optional block flush as a cache-to-cache transfer.

Each cache block has an associated state which can have one of the following values:

1. Modified (M): the cache block is valid in only one cache, and the value is (likely) different from the one in the main memory. This state has the same meaning as the dirty state in a write back cache for a single processor system.
2. Exclusive (E): the cache block is valid, clean, and only resides in one cache.
3. Shared (S): the cache block is valid, clean, and may reside in multiple caches.
4. Invalid (I): the cache block is invalid.

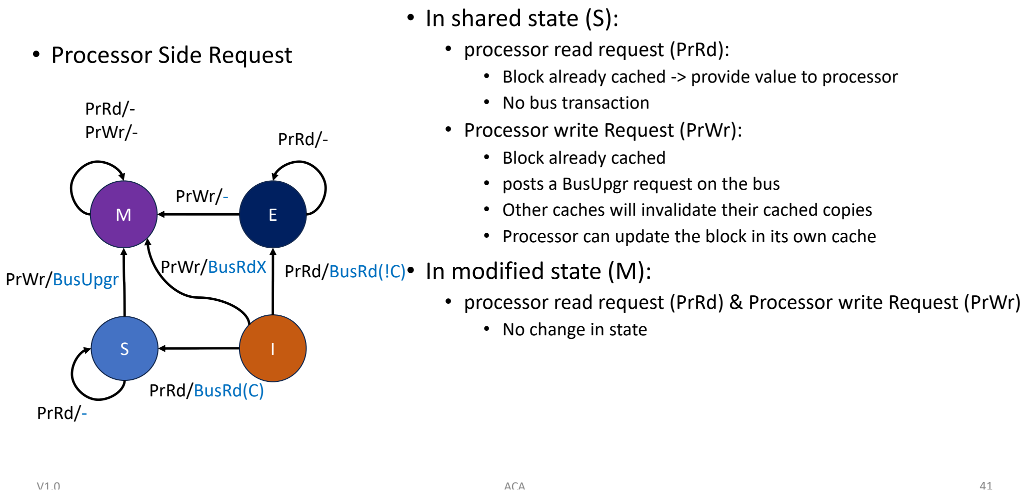


- Processor Side Request

- In invalid state (I):
  - Processor read request, other processor has cache block (PrRd(C)):
    - Cache miss occurs
    - To load the data into the cache, a BusRd is posted on the bus
    - Other processors indicate with C that they have a copy in cache
    - Fetching block from other cache (FlushOpt) -> Set state to S
  - Processor read request, no other processor has cache block (PrRd(!C)):
    - Cache miss occurs
    - To load the data into the cache, a BusRd is posted on the bus
    - Other processors indicate with C that they do not have a copy in cache
    - Fetching block from memory -> Set state to E
  - Processor write Request (PrWr):
    - posts a BusRdX request on the bus
    - Other caches will invalidate their cached copies, possibly flush to mem
    - Fetching block from memory -> Set state to M
    - Processor can update the block

V1.0 ACA 40

Figure 5.13: MESI Protocol with Write Back Caches - Snooper FSM (1)

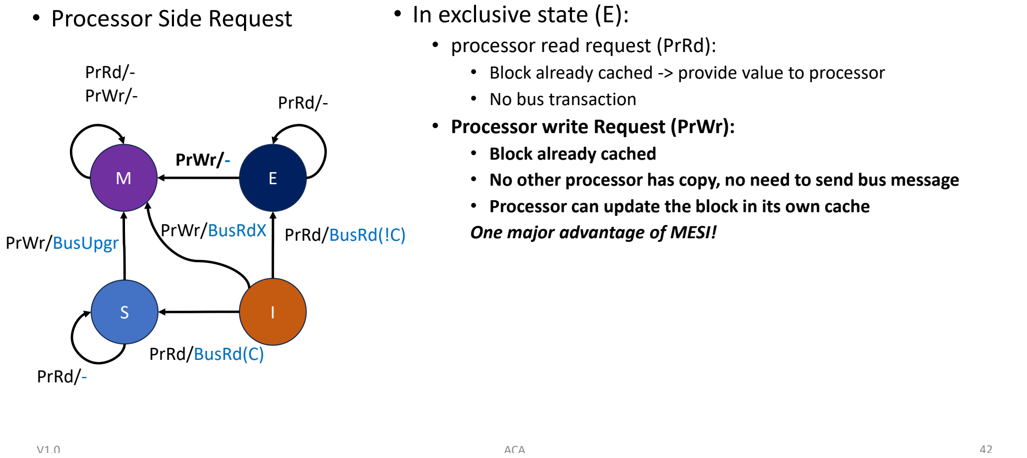


- Processor Side Request

- In shared state (S):
  - processor read request (PrRd):
    - Block already cached -> provide value to processor
    - No bus transaction
  - Processor write Request (PrWr):
    - Block already cached
    - posts a BusUpgr request on the bus
    - Other caches will invalidate their cached copies
    - Processor can update the block in its own cache
- In modified state (M):
  - processor read request (PrRd) & Processor write Request (PrWr)
    - No change in state

V1.0 ACA 41

Figure 5.14: MESI Protocol with Write Back Caches - Snooper FSM (2)



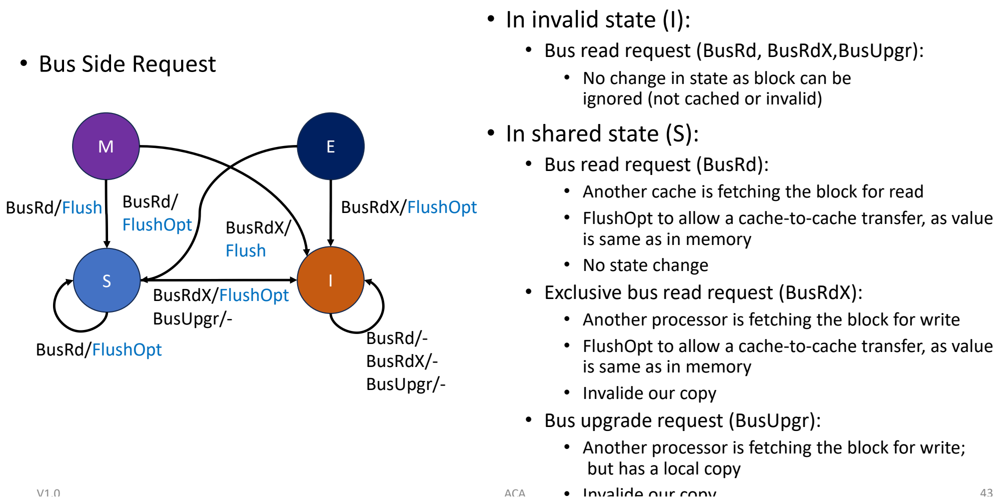
- Processor Side Request

- In exclusive state (E):
  - processor read request (PrRd):
    - Block already cached -> provide value to processor
    - No bus transaction
  - **Processor write Request (PrWr):**
    - **Block already cached**
    - **No other processor has copy, no need to send bus message**
    - **Processor can update the block in its own cache**

    One major advantage of MESI!

V1.0 ACA 42

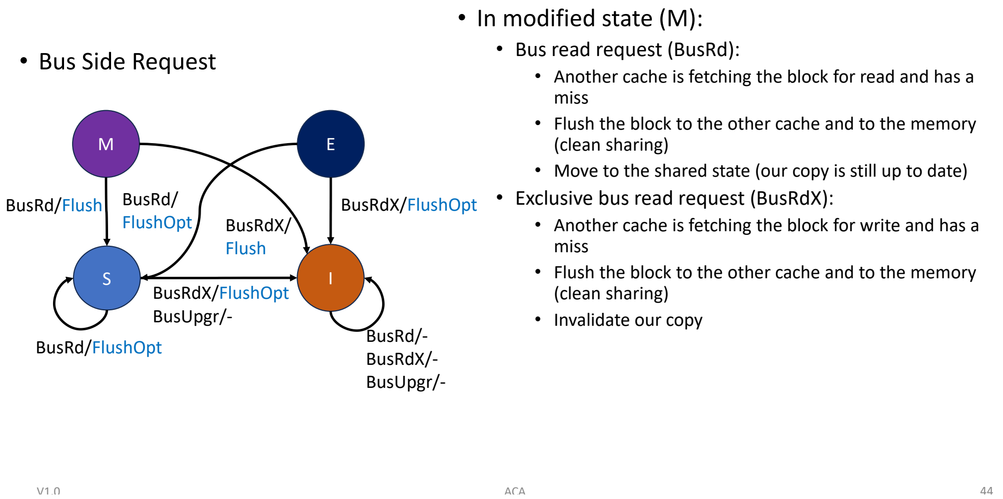
Figure 5.15: MESI Protocol with Write Back Caches - Snooper FSM (3)



- Bus Side Request

- In invalid state (I):
  - Bus read request (BusRd, BusRdX,BusUpgr):
    - No change in state as block can be ignored (not cached or invalid)
- In shared state (S):
  - Bus read request (BusRd):
    - Another cache is fetching the block for read
    - FlushOpt to allow a cache-to-cache transfer, as value is same as in memory
    - No state change
  - Exclusive bus read request (BusRdX):
    - Another processor is fetching the block for write
    - FlushOpt to allow a cache-to-cache transfer, as value is same as in memory
    - Invalide our copy
  - Bus upgrade request (BusUpgr):
    - Another processor is fetching the block for write; but has a local copy
    - Invalide our copy

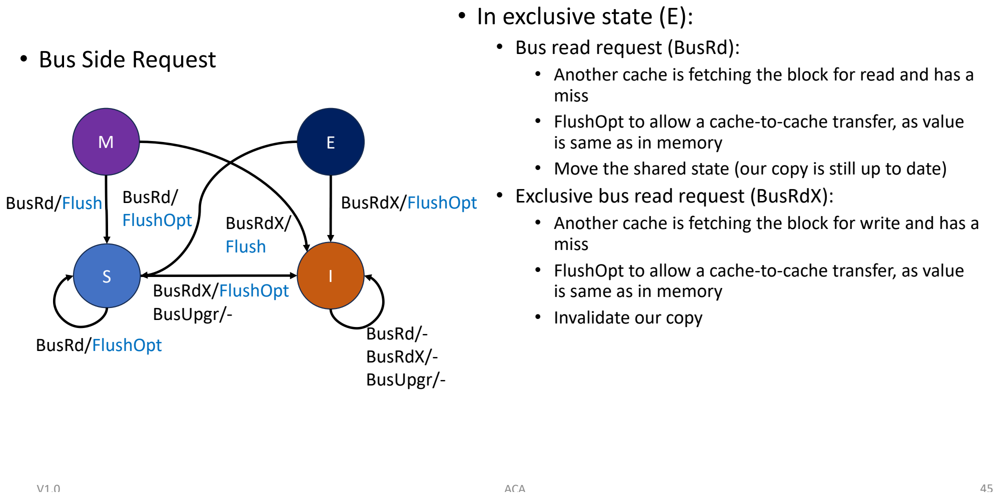
Figure 5.16: MESI Protocol with Write Back Caches - Snooper FSM (4)



- Bus Side Request

- In modified state (M):
  - Bus read request (BusRd):
    - Another cache is fetching the block for read and has a miss
    - Flush the block to the other cache and to the memory (clean sharing)
    - Move to the shared state (our copy is still up to date)
  - Exclusive bus read request (BusRdX):
    - Another cache is fetching the block for write and has a miss
    - Flush the block to the other cache and to the memory (clean sharing)
    - Invalidate our copy

Figure 5.17: MESI Protocol with Write Back Caches - Snooper FSM (5)



- Bus Side Request

- In exclusive state (E):
  - Bus read request (BusRd):
    - Another cache is fetching the block for read and has a miss
    - FlushOpt to allow a cache-to-cache transfer, as value is same as in memory
    - Move the shared state (our copy is still up to date)
  - Exclusive bus read request (BusRdX):
    - Another cache is fetching the block for write and has a miss
    - FlushOpt to allow a cache-to-cache transfer, as value is same as in memory
    - Invalidate our copy

Figure 5.18: MESI Protocol with Write Back Caches - Snooper FSM (6)

Comparison MSI vs. MESI

- Compared to the MSI protocol, the MESI protocol does not reduce bandwidth usage on the bus, but it does reduce bandwidth use to the main memory due to cache-to-cache transfers (FlushOpt).
- Bandwidth to the main memory is often a bottleneck when there are a lot of processors connected to the same memory (known as the Memory wall!).

- Additionally, MESI keeps track of data that is exclusive to the thread (threads often operate on private data; not all data is shared). No bus signaling is required for this private data.

5.2.5 MOESI Protocol with Write-Back Caches

Request In the MOESI protocol, processor requests to the cache include:
1. PrRd: processor-side request to read a cache block.
2. PrWr: processor-side request to write to a cache block.

Bus-side requests include:
1. BusRd: snooped request that indicates there is a read request to a cache block made by another processor.
2. BusRdX: snooped request that indicates there is a read exclusive (write) request to a cache block made by another processor which does not already have the block.
3. BusUpgr: snooped request that indicates that there is a write request to a cache block that another processor already has in its cache.
4. Flush: snooped request that indicates an entire cache block is placed on the bus by a processor to facilitate a transfer to another processor's cache. (Different from MESI! Not to memory, closer to FlushOpt in MESI!)
5. FlushOpt: snooped request that indicates an entire cache block is posted on the bus in order to supply it to another processor. (We refer to it as FlushOpt because, unlike Flush which is needed for write propagation correctness, FlushOpt is implemented as a performance-enhancing feature that can be removed without impacting correctness.)
6. FlushWB: snooped request that indicates that an entire cache block is written back to the main memory by another processor, and it is not meant as a transfer from one cache to another.

Cache Block States Each cache block has an associated state which can have one of the following values:
1. Modified (M): the cache block is valid in only one cache, and the value is (likely) different from the one in the main memory. This state has the same meaning as the dirty state in a write-back cache for a single-processor system, except that now it also implies exclusive ownership.
2. Owned (O): the cache block is valid, possibly dirty, and may reside in multiple caches. However, when there are multiple cached copies, there can only be one cache that has the block in the Owned state; other caches should have the block in the Shared state.
3. Exclusive (E): the cache block is valid, clean, and only resides in one cache.
4. Shared (S): the cache block is valid, possibly dirty, and may reside in multiple caches.
5. Invalid (I): the cache block is invalid.

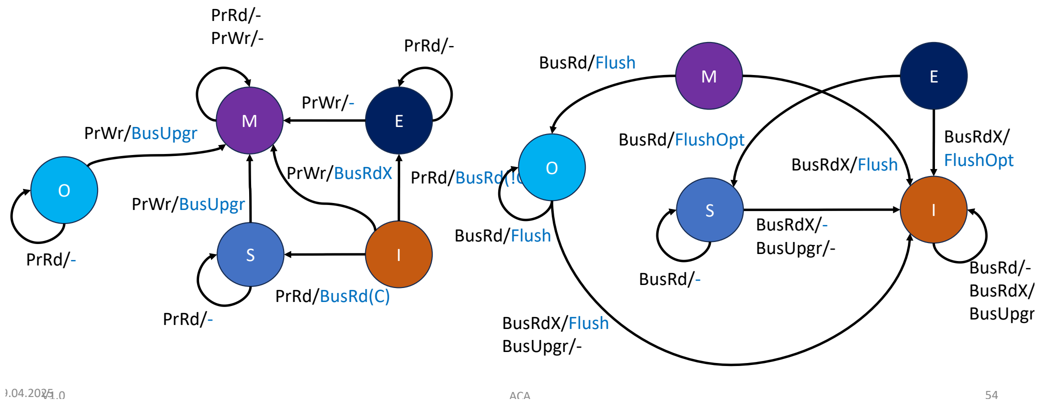


Figure 5.19: MOESI - Snooper FSM

FlushWB
- The owner (O) keeps track of the latest version on each block and supplies it.
- Dirty sharing: The memory may not have an up-to-date copy.

- FlushWB's role:
- If the owner evicts the cache block, then it needs to be written back to the main memory (this is the FlushWB), it is not in the FSM, as it is not caused by a read/write of this cache block, but by another block causing the eviction.
- There is no owner after that but other caches may still have block in shared state (transfer of owner can be implemented)

FlushOpt

- FlushOpt occurs when downgrading from Exclusive (E) to Shared (S) or Invalid (I)
- As a key characteristic, MOESI fetches blocks from the owner
- If the block is in the E state, it is not marked as "owned."
- Yet, as an optimization feature, FlushOpt indicates that the block is supplied by the cache having it in the "E" state and not by the memory in a clean-sharing cache-to-cache transfer
- This is not needed for correctness (write propagation) as the block could also be supplied by the memory (clean sharing; memory has a valid copy)

MOESI vs. MESI

- MOESI allows for dirty sharing:
  - Less memory traffic, faster transfers (cache-to-cache).
  - But with an L2 cache, the effect may be less important, as L2 to L1 transfers may still be fast.
- MOESI needs 3 bits per cache line to store state; MESI only 2 bits
- MESI, MOESI:
  - Open question: When several blocks have a clean cache block in the Shared state – who supplies the block?

5.2.6 Future Protocols

- MESIF (by Intel): MESI with a forwarding state (used as a designated supplier when several caches share a clean block), but no dirty sharing, such as MOESI.
- MSI, MESI; MOESI: Invalidation-based protocols

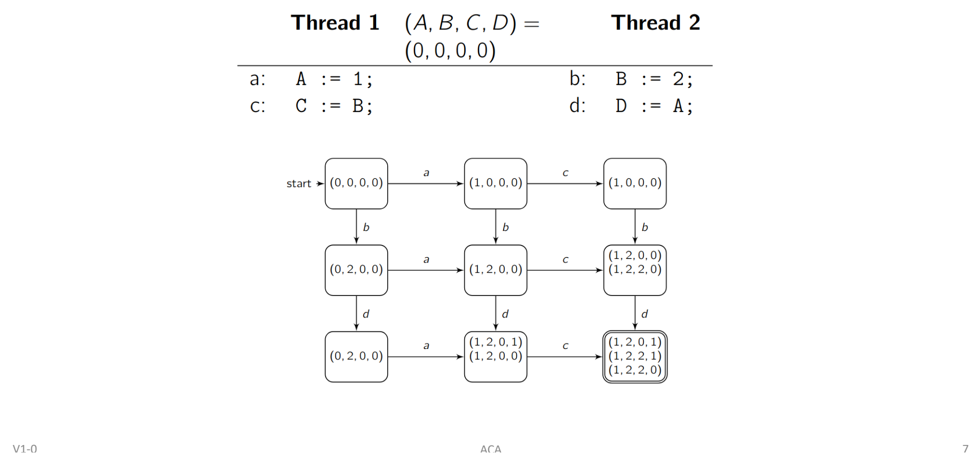
5.3 Memory Models

5.3.1 Sequential Consistency (SC) & Synchronization Problem

- Memory Model: Mechanisms are needed to ensure that accesses of one processor appear to execute in program order to all others, at least partly.
- Atomic Operations: Hardware support for synchronization

5.3.2 Abstract View on Interleaving Threads

- Interleavings are all possible intertwinings of sequences of statements from threads.
- An interleavings graph is a representation of interleavings in the form of a graph.
  - Each path from the start node to the end node of the graph corresponds to an interleaving.
  - The set of all such paths corresponds to the set of all possible interleavings
  - Due to different runtimes, different scheduling strategies, different hardware architectures, the actual execution sequence can match any arbitrary interleaving.
- a **Race Condition** is a situation in which the result of an operation depends on the temporally intertwined execution of certain other operations.



Thread 1	$(A, B, C, D) =$ $(0, 0, 0, 0)$	Thread 2
a: `A := 1;`		b: `B := 2;`
c: `C := B;`		d: `D := A;`

V1-0 ACA 7

Figure 5.20: Interleavings Graph

Sequential Consistency from the Programmer's Perspective

- a single global memory
- each core generates memory operations in program order

5.3.3 Atomic Instructions and Variables

- Assumption: The assignment of a 16-bit word occurs non-atomically, by copying the two 8-bit halves separately.
- When multiple threads access common memory cells (variables), it may be necessary to guarantee that operations on variables are executed atomically, i.e., indivisibly.
- This can only be guaranteed by the hardware (CPU).
- All common CPUs offer such atomic operations as instructions.

Volatile	Atomic
all other threads see all accesses to variables (not optimized by compiler)	Additionally, operations on these variables are atomic.

Table 5.6: Difference between volatile and atomic

5.3.4 Synchronization with Atomic Variables

Producer - Consumer

- A piece of data should be safely transferred from one thread to another thread.
  - thread T_1 writes to variable D, thread T_2 should read from D
- With the help of an atomic variable (flag F), data can be transferred "safely" from one thread to another.

5.3.5 Program vs. Execution Order in the Relaxed Memory Model

- In addition to atomic variables, executing the instructions in program order was recognized as a prerequisite for the Sequentially Consistent (SC) memory model.
- Modern computer architectures do not guarantee that instructions are executed in program order
- The compiler and OoO processor apply optimizations with reordering of instructions as for single-threaded execution

Level	(Re-)Ordering
Source Code	Program Order
Compiler	optimizing of the code (moving and removing instructions)
CPU	instruction scheduling, OoO-Execution
Memory	write buffers, caches, …
Execution	execution order

Table 5.7: Levels of Reordering in the Relaxed Memory Model

5.3.6 Release/Acquire Memory Model

- How can we integrate the new memory models so that sensible work is possible? We need additional hardware tools.
- Modern computer architectures offer so-called
- Memory-Fences (Memory Barriers).
- Moving instructions across memory fences is prohibited.

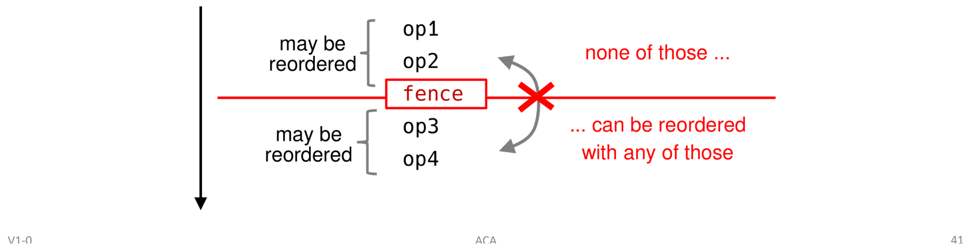


Figure 5.21: Release/Acquire Fence

- Programming languages must offer adequate language features so that memory fences can be utilized.
- Release and Acquire operations for atomic variables.
- Temporal relativity can be ensured for atomic variables, but not for conventional variables.

> **i Note**
>
> Automatic placement of Memory-Fences is not possible (undecidable problem)! (Equivalent to the Halting Problem)

Sequential Consistency on Modern Computers

Hardware tools can also ensure SC.

👍 Programmers do not have to worry.
👍 Programs are easier to write and debug.
👍 The correctness of such programs is easier to prove.
👎 The hardware instructions for SC are very expensive (slow).
👎 The performance advantages of modern architectures are not utilized.

Release/Acquire Memory Model

- Release-Operation: Sets a memory fence so that no load and store operations that stand in program order before the Release operation can be moved behind the Release operation.
- Acquire-Operation: Sets a memory fence so that no load and store operations that stand in program order after the Acquire operation can be moved before the Acquire operation.
- Relaxed-Operation: Sets no memory fences.

5.3.7 Blocking Wait

- Disadvantage of the previous type of communication/synchronization: a thread is in a loop until data can be read.
- Wastes unnecessary computing time and energy.

Semaphore

```
Lock(S);
Z_local := Z;
Z_local := Z_local +  ...;
Z := Z_local;
Unlock(S);
```

Code 5.8: Easy Semaphore Example

👍 easy to understand
👍 no waste of computing time and energy
👍 sequential consistent
👎 if a thread crashes between `lock` and `unlock`, no other thread can make progress
👎 thread dispatching takes a lot of time.
👎 Read-Modify-Write operations are slow

5.3.8 Non-Blocking Wait

- Instead of synchronization via semaphore or similar, direct use of Read-Modify-Write operations.
- Optimistic approach.
- Function `RMW(V, old_value, new_value)`
- Returns `true` if the atomic variable `V` still has the old value; `V` receives the new value simultaneously.
- Returns `false` otherwise. Implicitly, `old_value` is set to `new_value`.

👍 If few threads access the central variable simultaneously, it is very efficient.
👍 If a thread crashes, other threads can still make progress.
👍 Sequentially consistent and Release/Acquire memory model are possible.
👎 Difficult to understand for more complex algorithms.
👎 Even more difficult to understand for more complex algorithms when combined with Release/Acquire memory order.
👎 Suffers from the ABA problem (to be explained on the next slides).

The ABA Problem

1. Process P1 reads value A from a shared memory location.
2. P1 is preempted, allowing process P2 to run.
3. P2 writes value B to the shared memory location.
4. P2 writes value A to the shared memory location.
5. P2 is preempted, allowing process P1 to run.
6. P1 reads value A from the shared memory location.
7. P1 determines that the shared memory value has not changed and continues.
   - → Thus an RMW operation may succeed, although it actually should not.

- The ABA problem can be solved via CAS operations by counting the number of accesses to shared data.

Load-Link/Store-Conditional (LL/SC) Operation

- Function `LL(address)` loads the value stored at the address
- Function `SC(address, value)` stores the value at the address, provided that there was no interfering store to the address. It returns true if successful, and false otherwise.
- LL: stores the address at the cache line

- Any modification to any portion of the cache line (via conditional or ordinary store) causes the store-conditional (SC) to fail
- LL/SC operations are supported by DEC Alpha, PowerPC, MIPS, ARM, RISC-V, etc.

👍 Not sensitive to the ABA problem

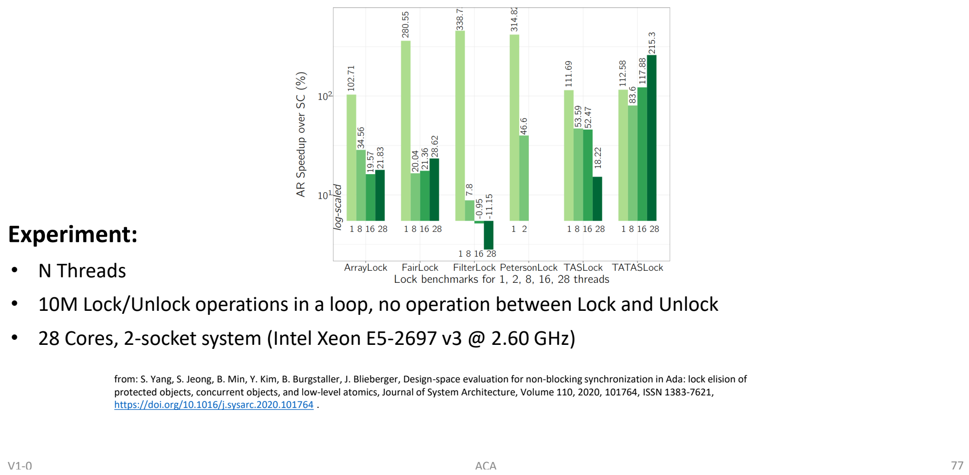
👍 Instruction set: needs two words instead of three, which are needed by CAS

👎 Sometimes fails if a context switch occurs between LL and SC operations

👎 Sometimes fails if a second LL/SC operation occurs

👎 No nesting of LL/SC operations

5.3.9 Performance Comparison



Experiment:

- N Threads
- 10M Lock/Unlock operations in a loop, no operation between Lock and Unlock
- 28 Cores, 2-socket system (Intel Xeon E5-2697 v3 @ 2.60 GHz)

from: S. Yang, S. Jeong, B. Min, Y. Kim, B. Burgstaller, J. Blieberger, Design-space evaluation for non-blocking synchronization in Ada: lock elision of protected objects, concurrent objects, and low-level atomics, Journal of System Architecture, Volume 110, 2020, 101764, ISSN 1383-7621, https://doi.org/10.1016/j.sysarc.2020.101764 .

V1-0 ACA 77

Figure 5.22: Performance Gain Intel X86 through SC-AR-Relaxation

6 Block F

6.1 On-Chip Buses

Motivation

- Most chips feature a range of processing elements (PEs) / multi-cores
- PEs need to communicate with each other
- On-chip Interconnect architecture and type play crucial role in performance.
- Chips and devices are connected via different types of interconnects

6.1.1 Interconnect Types

- **On-Chip**: Connects modules that are integrated into the same chip (IC: integrated circuit)
- **PCB-level**: Connects different ASICs + connectors and other component all mounted on one Printed Circuit Board (PCB).
- Many other interconnects (board to board, rack to rack): PCIe, Ethernet, CAN, UART, I2C, SPI, GPIO

6.1.2 On-chip Buses

6.1.2.1 Memory-mapped Buses

- Purpose
  - Read or write a value from or to a certain address
  - Value can be data or peripheral control information
- Memory-mapped Bus has several (sub-)buses (group of signals) and a defined bus protocol
  - Address bus
  - Data bus for reading data
  - Data bus for writing data
  - Control signals: Indicate if access is read or write, bus length, ID, bus grant, …
- Modules on the bus can either act as initiators or targets
  - Typical initiators: CPUs, DSPs, DMAs, bus bridges, …
  - Typical targets: Memory, accelerators, interface peripheral, bus bridge

Classes of Memory-mapped Buses

- **Single-initiator bus**
  - One initiator component can address different target components, which are mapped to different addresses
- **Shared bus**
  - There are several initiators on the bus
  - An arbiter decides which initiator module is granted access to the bus
  - Only one initiator can access one slave via the bus at a time
- **Layered bus**
  - There is more than one arbiter such that more than one initiator is granted access on the bus
  - Only one target component on each layer can be accessed at a time
- **Crossbar/ bus matrix**
  - Each target component has its own arbiter
  - Each target component can be accessed by one initiator at a time

6.1.2.2 Single Initiator

- Target knows
  - if it is addressed by observing the address bus ADDR

  - or decoder generates SEL signal for targets based on address bus ADDR
- Target can receive data on write data bus WDATA
- Decoder forwards the data from the addressed target by multiplexing it to the read data bus RDATA
- Additional control bus CTRL for signals related to bus protocol (e.g. WR, SEL, RDY)

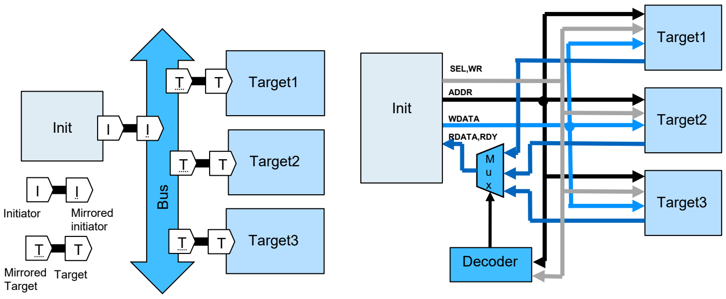


Figure 6.1: Single Initiator

Simple Write Access

1. Initiator places address and data on the ADDR and WDATA bus
   - Initiator indicates write by setting signal WR to high
   - Initiator indicates that access is started by setting SEL signal to high
2. Target acknowledges write access by RDY signal

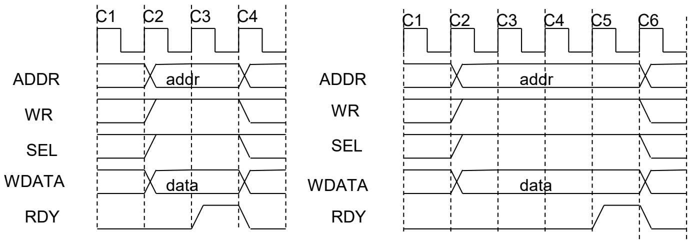


Figure 6.2: Single Initiator - Write

Simple Read Access

1. Initiator places address on the ADDR bus
   - Initiator indicates read access by setting signal WR to low
   - Initiator indicates that access is started by setting SEL signal to high
2. Target places data on RDATA bus
   - Target acknowledges read access by RDY signal

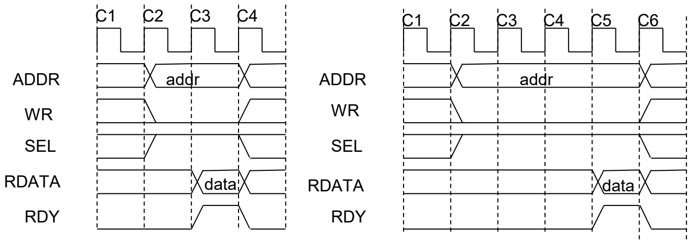


Figure 6.3: Single Initiator - Read

Performance

- each access takes a minimum of two cycles
- maximum bus bandwidth is: $\mathrm{BW}_{\mathrm{bus}} = 0.5 \cdot \mathrm{buswidth} \cdot f_{\mathrm{bus}}$

Pipelined Access

- The next address can be placed on the bus while the data is read
  - Additional control signals and logic required to support pipelined accesses
- maximum bus bandwidth is: $\text{BW}_{\text{bus}} = \text{buswidth} \cdot f_{\text{bus}}$

Burst Accesses

- A burst accesses consecutive addresses
- Version 1: the addresses for all accesses must be given and a control signal that indicates that this is a burst access of a certain size
- Version 2: Only the start address and a control signal indicating the burst size must be provided

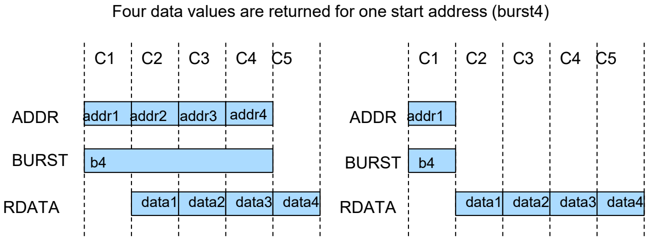


Figure 6.4: Single Initiator - Burst

6.1.2.3 Multiple Initiators

Shared Bus

- The arbiter grants access to the initiator
- only the address and data of the initiator are forwarded to the targets

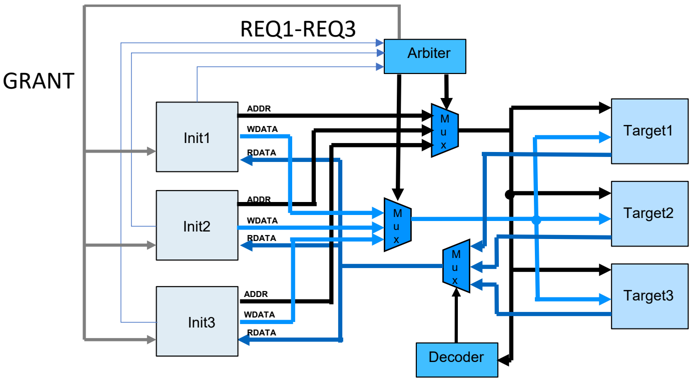


Figure 6.5: Multiple Initiators - Arbiter

- Round-robin: Access granted to initiators in pre-defined order that is repeated
- FIFO: First initiator requesting the bus is granted access
- Priority: Initiator with highest priority is granted access to the bus

Split Accesses

- A slave can allow an access to be split if it involves many wait cycles
- Access for initiator I_1 is split by the slave issuing a start of split
- I_2 is granted the bus, and the access for initiator I_2 is performed
  - Then, the access for initiator I_1 is finished by issuing an end of split

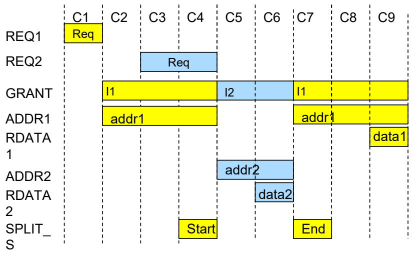


Figure 6.6: Multiple Initiators - Arbiter

Crossbar / Bus Matrix

- all targets can be accessed individually
- conflicts only arise when two initiators access the same target
- `GRANT/REQ` omitted

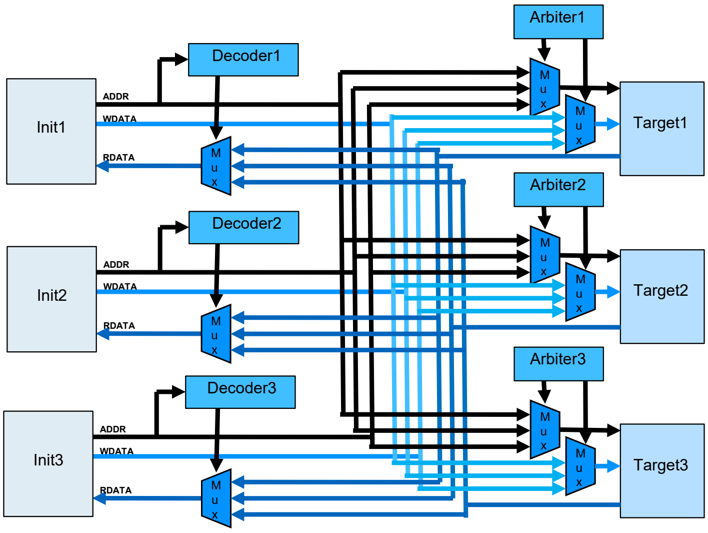


Figure 6.7: Multiple Initiators - Bus Matrix

Layered Bus

- Targets are on different layers
- Initiator can connect to targets on different layers simultaneously

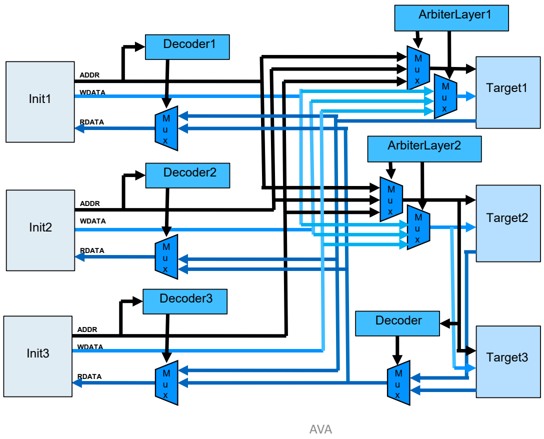


Figure 6.8: Multiple Initiators - Layered Bus

Some Bus Standards

- AMBA Bus (ARM)

    - AHB: Advanced High Performance Bus
    - APB: Advanced Peripheral Bus
    - AXI: Advanced eXtensible Interface
- Wishbone (Open)
- TileLink (Open)

6.2 Network-on-Chip (NoC)

6.2.1 Introduction to NoCs

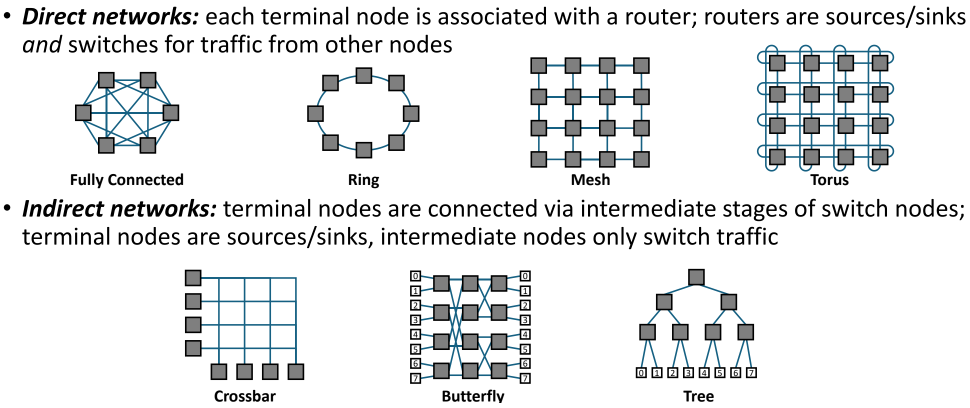
- Need for scalability and reduced cost
    - Avoid long interconnects and delays caused by increased system complexity
    - Reduce wiring overhead due to an increasing number of system components
- Performance demands
    - Goal: high bandwidth and low latency
    - Concurrent communication required due to increased traffic
- Solution: Network-on-Chip (NoC)
    - Move from bus to network (small-scale networks at the chip/system level)
        - Larger-scale networks will be discussed in later lectures
    - Broadcast can be avoided, but is still possible via multiple messages (when required)
    - Serialization is achievable, e.g., by forcing the same path or via sequence numbers

Basics

- Objective: To connect nodes with each other via routers and wires, enabling messages to be sent from source to destination
- Building blocks:
    - **Node**: any component, e.g., processor, memory, or a combination of them
    - **Network interface**: module connecting a node to the network
    - **Router**: forwards data from inputs to outputs (network interfaces or other routers)
    - **Link**: physical set of wires, e.g., connecting two routers
    - **Channel**: logical connection between routers
    - **Message**: unit of transfer for the nodes
    - **Packet**: unit of transfer for the network

6.2.2 NoC Topologies

- Topology: arrangement of nodes and channels
    - Determines e.g., number of hops, number of alternative paths, cost
- Properties for comparison
    - **Degree**: number of links at each node
    - **Distance**: number of links in the shortest route
    - **Diameter**: maximum distance between any two nodes
    - **Bisection bandwidth**: available bandwidth from one partition to the other, when cutting the network into two equal parts (minimum for multiple possible cuts)



- ***Direct networks:*** each terminal node is associated with a router; routers are sources/sinks *and* switches for traffic from other nodes
- ***Indirect networks:*** terminal nodes are connected via intermediate stages of switch nodes; terminal nodes are sources/sinks, intermediate nodes only switch traffic

Figure 6.9: Network Topologies

	Fully Connected	Ring	Mesh	Torus	Crossbar	Butterfly	Tree
Description	every node is connected to every other node with a direct link	each node is connected to two other nodes	k nodes per dimension	k nodes per dimension	connects n inputs to m outputs		
Nodes	N	N	$N = 2^k$	$N = 2^k$	$N = n \cdot m$		
Links	$(N \cdot (N - 1))/2$	N	$2k \cdot (k - 1)$	$2N$	$n \cdot m$		
Degree	N	N	4	4	-		
Diameter	1	$\lfloor N/2 \rfloor$	$2k - 2$	k	1		
Bisection Width	$\lfloor N/2 \rfloor \cdot \lceil N/2 \rceil$	2	k	$2k$	-		
Pros	high fault tolerance, low contention, low latency	simple, low link costs	path diversity, regular and equal-length links	avoids asymmetry and improves path diversity	non-blocking, latency	lower const compared to crossbar	simple, cheap
Cons	high costs, limited scalability	high latency, limited path diversity	large diameter, asymmetric (higher demand for central links	unequal link lengths, higher cost compared to mesh	high cost, limited scalability	blocking, lack of path diversity, locality not exploitable	bottleneck towards root

Table 6.8: Comparison of Network Topologies

6.2.3 NoC Messages

- Message: a logically continuous group of bits, may be arbitrarily long
- Packet: basic unit of routing and sequencing, with a restricted maximum length
  - Consists of a header and a segment of a message

- Flit (flow control digit): basic unit of bandwidth and storage allocation
  - Contains no separate routing/sequencing information and therefore follow the same path in-order
  - Subdivision allows for low overhead (large packets) and fine-grained resource utilization (small flits)
- Phit (physical transfer digit): information transferred over a channel in a single clock cycle

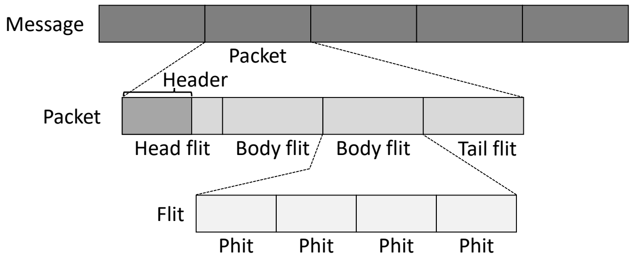


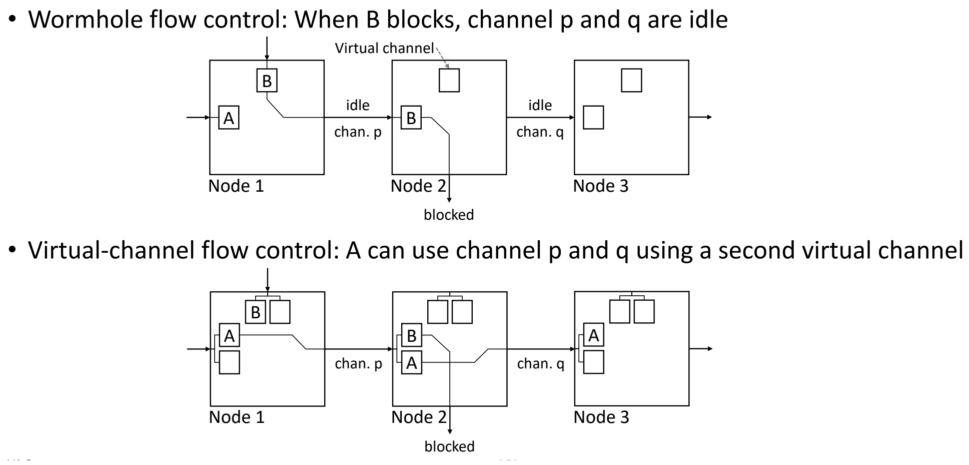
Figure 6.10: Message

Flow Control vs. Routing

- Flow control: Allocates resources (channels, control state, buffers) to packets
  - Alternative view: resolve contention during packet transmission
  - Contention: What happens if two packets want to use the same channel at the same time?
- Routing: Selects the path a packet takes from source to destination
  - Determines how well the potential of the given topology is exploited
  - Should balance load across network channels

6.2.4 NoC Flow Control

- Bufferless
  - Dropping
  - Misrouting
    - No buffers available, therefore misroute “losing” packets, “winning” packet gets the requested channel
  - Circuit switching
    - First allocate channels to build a circuit from source to destination, then send packets along the circuit, deallocate circuit after packets are sent
- Buffered
  - Store-and-forward
    - Each node waits until packet is received completely before transmission to the next node
  - Cut-through
    - Flits are forwarded as soon as they are received, and the subsequent channel and buffer space is acquired (allocation still at packet granularity)
  - Wormhole
  - Virtual channel



- Wormhole flow control: When B blocks, channel p and q are idle
- Virtual-channel flow control: A can use channel p and q using a second virtual channel

Figure 6.11: Wormhole & Virtual-channel

6.2.5 NoC Routing

- Selects the path a packet takes from source to destination in a given topology
- Determines how well the potential of the given topology is exploited
- Balance load across the network channels to avoid hotspots and contention
  - Difficult, particularly with non-uniform traffic patterns causing load misbalances

6.2.5.1 Dimension-order Routing

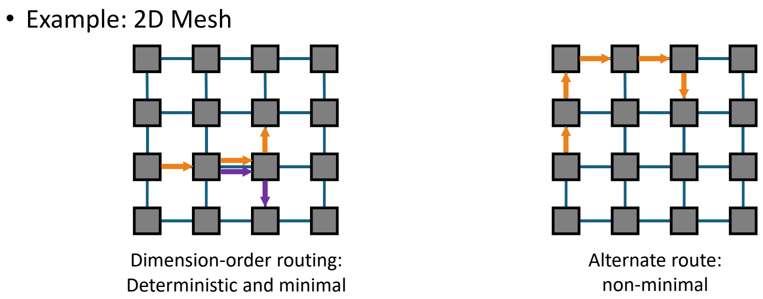
- First move towards x-dimension, then move towards y-dimension (XY)

👍 simple

👍 minimal

👎 can cause load imbalance

👎 doesn't exploit path diversity



- Example: 2D Mesh

Figure 6.12: Dimension-order Routing

6.2.5.2 Valiant's Algorithm

- A packet from source s to destination d is routed via an intermediate node d'
  - Randomly select intermediate node d'
  - Phase I: Route packet from s to d'
  - Phase II: Route packet from d' to d
  - Use arbitrary routing algorithm for Phase I+II, e.g., dimension order routing for tori and meshes

👍 randomizes traffic

👍 balances network load

👎 non-minimal

👎 doesn't exploit locality

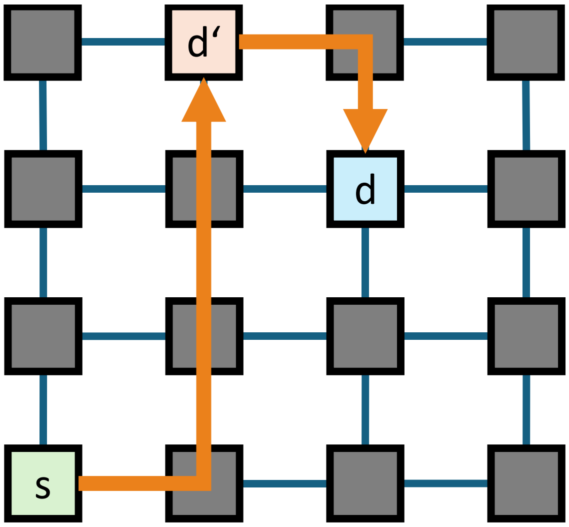


Figure 6.13: Valiant’s Routing

- Improvement
  - restrict d' to one in a minimal quadrant
  - preserves locality, compromises load balancing

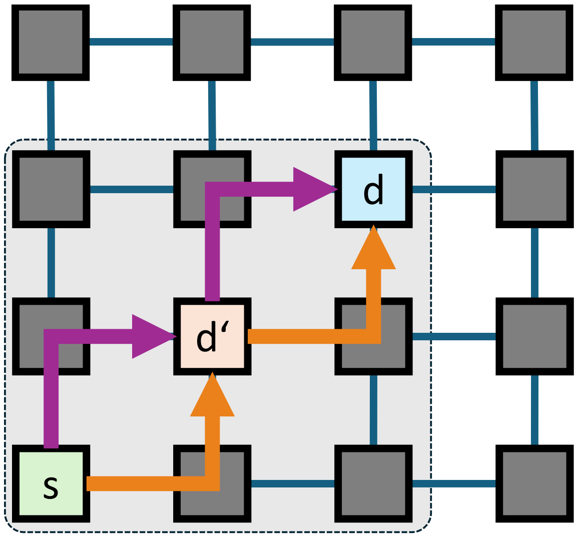
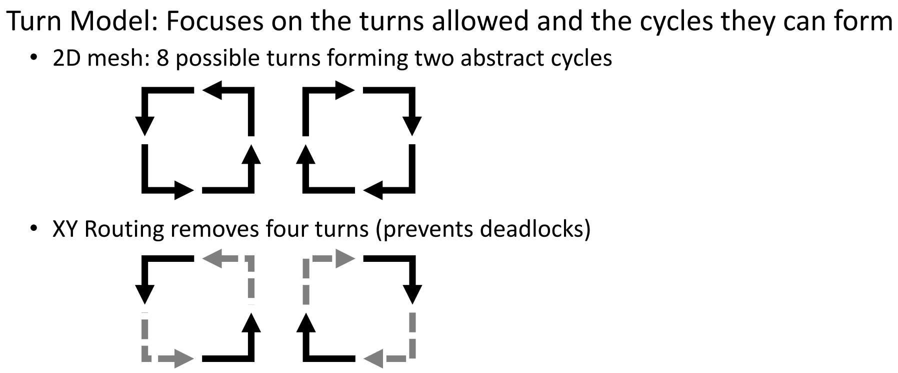


Figure 6.14: Valiant’s Improved Routing

- Deadlocks
  - situation where packets cannot make progress as they are waiting on each other to release resources



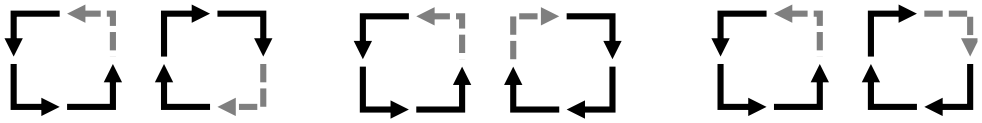
Turn Model: Focuses on the turns allowed and the cycles they can form

- 2D mesh: 8 possible turns forming two abstract cycles
- XY Routing removes four turns (prevents deadlocks)

Figure 6.15: Restrict Routing (1)

Turn Model: Focuses on the turns allowed and the cycles they can form

- Removing one (carefully selected) turn from each abstract cycle also prevents deadlocks



west-first: traveling west only allowed at the start

north-last: traveling north only allowed as last direction

negative-first: traveling first west and south, then east and north

- Removing any two turns does not prevent deadlocks

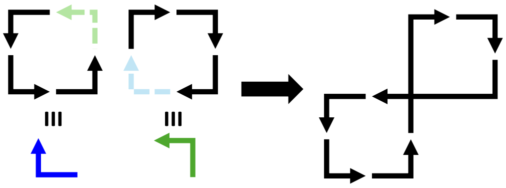
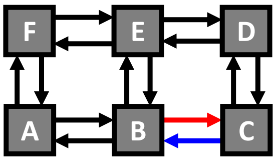


Figure 6.16: Restrict Routing (2)

6.2.5.3 Channel Dependence Graph (CDG)

Network topology:



Channel Dependence Graph:

- One vertex for each channel
- Edges denote dependences
    - Dependence exists if it is possible for channel *i* to wait for channel *i+1*
    - 180° turns not allowed (e.g., AB → BA)

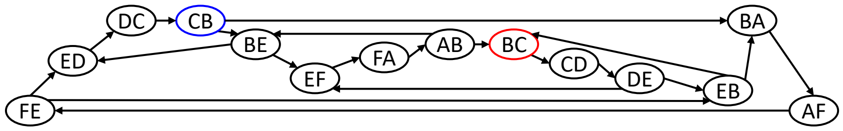
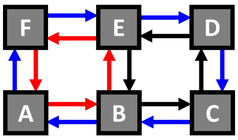
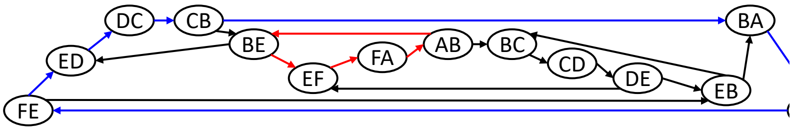


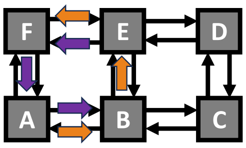
Figure 6.17: CDG

Channel Dependence Graph may contain cycles





Route through AB, BE, EF and route through EF, FA, AB → Deadlock



→ Remove selected edges in the CDG

Figure 6.18: Cycles in CDG

Example: Remove Edges in the CDG (West-first turn model)

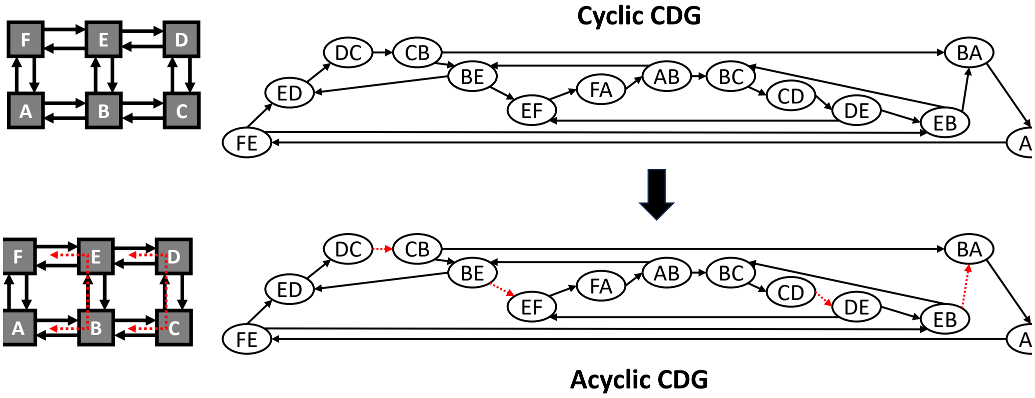
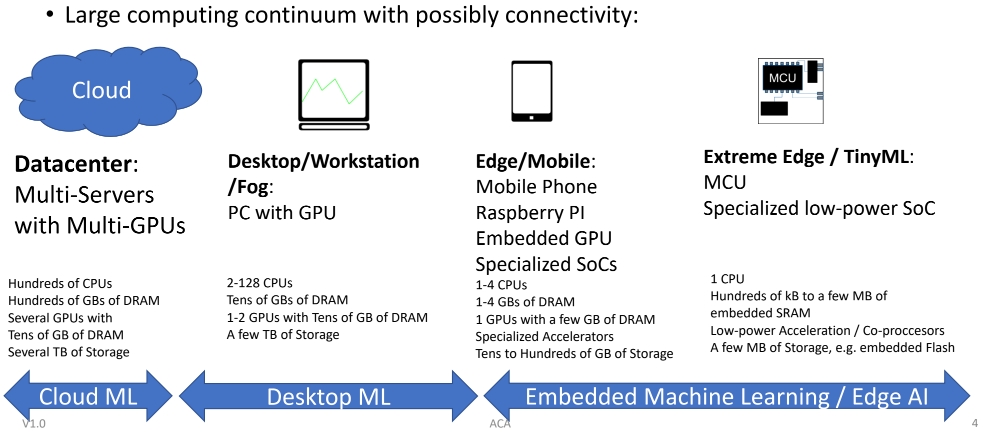


Figure 6.19: Acyclic CDG

7 Block G

7.1 GP-GPUs, TPUs, NPUs



- Large computing continuum with possibly connectivity:

Datacenter:
Multi-Servers
with Multi-GPUs

Hundreds of CPUs
Hundreds of GBs of DRAM
Several GPUs with
Tens of GB of DRAM
Several TB of Storage

Desktop/Workstation /Fog:
PC with GPU

2-128 CPUs
Tens of GBs of DRAM
1-2 GPUs with Tens of GB of DRAM
A few TB of Storage

Edge/Mobile:
Mobile Phone
Raspberry PI
Embedded GPU
Specialized SoCs

1-4 CPUs
1-4 GBs of DRAM
1 GPUs with a few GB of DRAM
Specialized Accelerators
Tens to Hundreds of GB of Storage

Extreme Edge / TinyML:
MCU
Specialized low-power SoC

1 CPU
Hundreds of kB to a few MB of
embedded SRAM
Low-power Acceleration / Co-proccesors
A few MB of Storage, e.g. embedded Flash

Cloud ML
Desktop ML
Embedded Machine Learning / Edge AI

Figure 7.1: ML Platforms

- Types include: Deep Neural Networks, Convolutional Neural Networks, Transformers, Graph Neural Networks, and Recursive Neural Networks.
- Computing demand is often measured in MAC operations
- Size is often measured in the number of parameters
- Examples
- Large Language Models (LLMs) - produces human-like text
  - GPT-4: 170 trillion parameters (10e12)
  - GPT-3: 175 billion parameters (10e9)
  - ResNet18: 11 million parameters (10e6) for image classification (e.g., autonomous driving)
  - Keyword Spotting (KWS): 16k-300k parameters (10e3) to detect keywords in an audio stream (e.g., for audio wakeup in TinyML)

Embedded ML Applications

- Data is generated at the edge by various sensors.
- ML applications are executed on embedded devices “close to” the sensors.
- Examples
  - Autonomous driving based on HD cameras, Lidar, and radar
  - Wearable human activity tracking using gyroscopes and accelerometers
  - Visual wake-up from camera
  - Audio wake-up (keyword spotting) from microphone
  - Gesture recognition from radar sensors

ANN Architectures

- Layered computation: $a^l = f^{l(a^{l-1})}$
  - $a^L = f^L(f^{L-1}(...f^1(x)...))$
- $a^l = f^l\left(W \cdot a^{l-1} + b^{l-1}\right)$
  - W: weights
  - a: activations
  - f^l: activation functions (`ReLU`, `tanh`, `softmax`)
- The design of the ANN model architecture must consider the target system
  - ROM/RAM memory resources (weights, activations)
  - Computational power: Operations (support for nonlinear operations)

  - Acceleration features (types of layers, layer configurations)
- Network Architecture Search (NAS)
  - Algorithms that systematically explore different ANN model architectures automatically
  - Computationally very expensive (requires training many candidates to evaluate accuracy)

Training

- Training of the ANN model is done on a powerful machine (GPU)
- Trained model is deployed on the embedded device
- The embedded device executes the trained model (inference task).
- Training
  - Selection of hyperparameters
  - Optimization of the ANN model's trainable parameters
  - Typically using a backpropagation algorithm

Quantization

- Models are usually trained with floating-point (FP) precision (float, double).
- Inference (execution of trained model on embedded device)
- Full precision (FP) computation (multiplication, addition) is expensive
- Hardware Floating Point Units are expensive (area, energy)
- For inference, the model is transferred to a quantized variant
- Integer computations are less expensive
- Many challenges: rounding, overflow, rescaling, shifting
- Simple Example (8-bit integer $[-127...128]$):

Quantization Formats

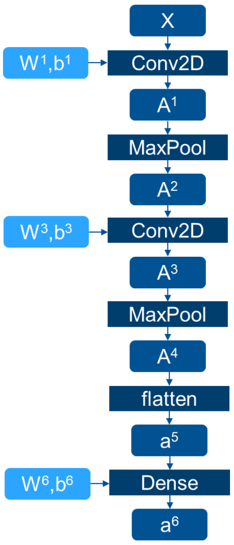
- Integer formats for weights/activations usually given by bit-width: x-bit
- On many embedded processors (byte-type quantization simplest 8bit, 16bit)
- Byte /uint8 (8bit) quantization range: [0 ... 255]
- Accumulation variables usually larger size
- Sub-byte integer quantization < 8 bit
  - Binary quantization w in {0,1}
  - Ternary quantization w in {-1,0,1}
- Also reduced-precision floating-point possible (many formats)

Pruning

- *Unstructured pruning*: Small weight values are set to zero.
  - Skip computation with zero values (may require additional logic in the program)
- *Structured pruning*: A column, row, or kernel is removed from the operator
  - The operator is modified

Consists of layers (structure reprented by data flow graph)

$$
\begin{aligned}
\mathbf{A}^1 &= \mathrm{Conv2D}(\mathbf{X}, \mathrm{W}^1, \mathbf{b}^1, \sigma_s^1, \sigma_r^1, \delta_s^1, \delta_r^1, P^1, \mathrm{ReLu}) \\
\mathbf{A}^2 &= \mathrm{maxpool}(\mathbf{A}^1 \pi_r^2, \pi_s^2) \\
\mathbf{A}^3 &= \mathrm{Conv2D}(\mathbf{A}^2, \mathrm{W}^3, \mathbf{b}^3, \sigma_s^3, \sigma_r^3, \delta_s^3, \delta_r^3, P^3, \mathrm{ReLu}) \\
\mathbf{A}^4 &= \mathrm{maxpool}(\mathbf{A}^3, \pi_r^4, \pi_s^4) \\
\mathbf{a}^5 &= \mathrm{flatten}(\mathbf{A}^4) \\
\mathbf{a}^6 &= \mathrm{Dense}(\mathbf{a}^5, \mathrm{W}^6, \mathbf{b}^6, \mathrm{Softmax})
\end{aligned}
$$



ACA

Figure 7.2: Convolutional Neural Network - Example

Image to Column (Img2Col) Transformation

- For many targets, a highly optimized implementation of matrix-matrix-multiply computation exists (e.g., for accelerators, CPUs with SIMD support, GPUs, and even single-issue CPUs).
- Img2Col transforms a convolution operation into a matrix-matrix multiply operation.
- Requires building a batch matrix that is larger than the original activation tensor because it holds duplicate values.
  - Usually, Img2Col is not applied to the full input activation tensor but rather within the convolution loop on a portion of the tensor to avoid building the entire batch matrix

7.1.1 General-Purpose Graphics Processor Units (GPGPUs)

- GPUs were initially introduced for real-time rendering, especially for video games.
- Nowadays, GPUs can be found in many devices (data centers, PCs, laptops, phones, embedded GPUs, etc.).
- General Purpose (GP-GPU): NVIDIA's CUDA programming language allowed GPUs to be used for general-purpose computing beyond rendering (now especially used for ML)
- GPUs are combined with a CPU either on a single chip or by inserting an additional card (e.g., via PCIe).
- The CPU is responsible for initiating computation on the GPU and transferring data to and from the GPU; it is often called "the host".

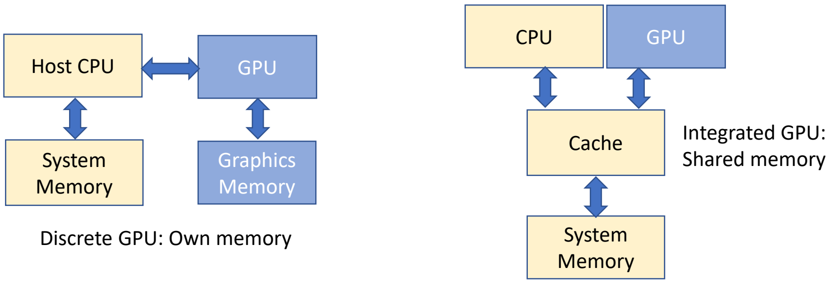


Figure 7.3: GPU: Discrete vs Integrated

Threads, Warps, Thread block

- The threads that compose a compute kernel are organized into a hierarchy consisting of a grid of thread blocks, which in turn consist of warps.
- In the CUDA programming model, individual threads execute instructions whose operands are scalar values (e.g., 32-bit floating-point).

- To improve efficiency, typical GPU hardware executes groups of threads together in lock-step (SIMD). These groups, called warps, consist of 32 threads
- Warps are grouped into a larger unit, called a thread block by NVIDIA.
- GPUs use the Single Instruction, Multiple Data (SIMD) model
- Scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware
- Loads and stores are scatter-gather, as threads perform scalar loads and stores.

8 Laboratories

8.1 Lab 1: Vector Processors

8.1.1 Vector Configuration

- Vector processor utilizes SIMD

Design Time Parameters:

- VLEN
  - in `vector_config.cmake` called `VREG_W`
- `VMEM_W`
  - size of the memory ports in bits
  - has to be equal to the pipeline containing the VLSU

Run Time Parameters:

- VL
  - if $\mathrm{vl} > \mathrm{VLMAX} \Rightarrow \mathrm{vl} = \mathrm{VLMAX}$ is returned
  - VLMAX can be requested with `__riscv_vsetvlmax_e{}m{}`
- SEW
- - LMUL

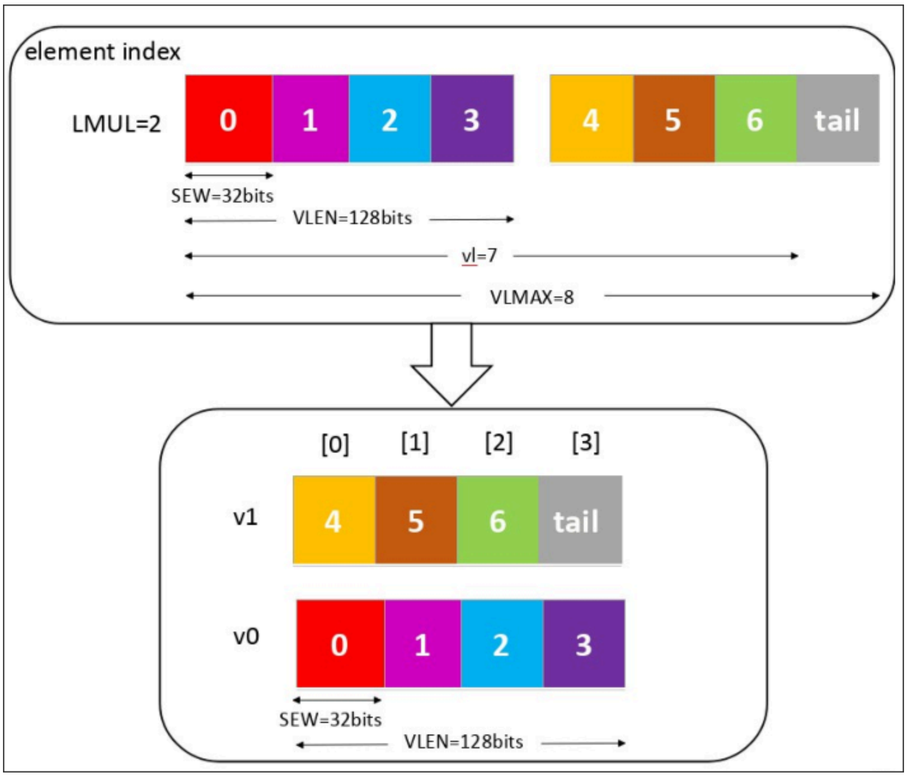


Figure 8.1: Graphical Representation of the RISC-V Design-Time and Runtime Parameters

Restrictions:

- all 6 functional units must be present
- only one unit of each type is allowed
- $2 \leq \#$ number of pipelines ≤ 6

Timing Behavior:

- wider lanes allows for functional units to precess faster through more parallel operations
- microarchitecture always passes entive vector register groups through the pipeline
  - ⇒ latency of instructions depends on `VLMAX` independet on the current set `vl`
- Only one vector operation can be processed in a vector pipeline at once.
  - However, if two vector operations use different pipelines with no data dependencies, they can be processed simultaneously
- 3 Read Ports, 1 Write Port
- vector-chaining is performed at the whole-vector register level
  - assuming each operation uses a different pipeline

8.1.2 Functional Units

Functional Unit	Description & Tasks	Commands	Latency Cycles	Init Interval
VLSU	Laod-Store Unit	`vle`, `vse`, `vlse`, `vsse`	4	1
VMUL	Multiply-Unit	`vmul`, `vwmul`	5	1
VALU	Arithmetic Unit	`vadd`, `vsub`, `vwadd`, `vwsub`	3	1
SLD	Slide Unit		variable	variable
VDIV	Division Unit		34	32
VLEM	Elementwise Unit		variable (but very slow)	variable

8.1.3 How to Start

8.1.3.1 (optional) Find a good algorithm

- mostly hard

8.1.3.2 Implement Basic Vectorized Code

❗ first version -> keep - LMUL at a minimum
  - 1 before every sign extension

□ get it working according to test cases

□ *chunk addressing*
  - fit as much data into the variables but keep track how much data was processed already

```
size_t rem_colsB = numColsB;
while (rem_colsB > 0U) {
    ...
    rem_colsB -= vl;
}
```

□ pointer access important and **dangerours**

8.1.3.3 Maximize - LMUL

□ just so within a certain scope (e.g. a while loop) not more than 32 physical registers are used
  - → but already seperated into respective pipelines to utilize vector-chaining

8.1.3.4 optimize `vector_config.cmake`

□ identify which unit is used for which operation and then seperate in respective pipelines
  - no overhead, minimal seperation

□ try to “play around” with `REG_W`, but very likely it shouldn’t be too high
  - either 64 or 128 bits

  ❗ can either reduce/boost performance

□ minimize pipeline width to get to LUT target (but maximize within LUT target)
  - → do everything within the LUT target (minimize everything to achieve LUT target, but maximize within LUT target)

8.1.3.5 Loop Unrolling

□ try to process multiple (2x, 4x, 8x) vector operations within a single loop cycle

- use the following constructs for it

```
while (2* i < numRowsB){...}
i = 2*i;
while (i < numRowsB){...} // tail
```

or

```
size_t vl_max = __riscv_vsetvlmax_e16m4();
while (len >= 2* vl_max) {...} // in here only load vl_max elements
while (len > 0U){...} // tail
```

❗ do not forget the tail handling

8.1.3.6 Play Around

❗ use as little vector registers as possible

❗ use combined wideining instruction

  - instead of multiplication and `sext` seperatly

□ play around with `vector_config.cmake` in order to hit targets

8.1.4 RISC-V V Extension Assembly Reference

8.1.4.1 Data types and configuration instructions

```
vsetvli rd, rs1, vtypei   # rd = new vl,  rs1 = AVL, vtypei = new vtype setting
vsetivli rd, uimm, vtypei # rd = new vl, uimm = AVL, vtypei = new vtype setting
vsetvl rd, rs1, rs2       # rd = new vl,  rs1 = AVL, rs2 = new vtype value

# Accepted SEW: e8/e16/e32/e64
# Accepted LMUL: mf8/mf4/mf2/m1/m2/m4/m8

# Examples: Request vector length stored in a0, store the provided length in t0
vsetvli t0, a0, e8        # SEW= 8, LMUL=1
vsetvli t0, a0, e8, m2    # SEW= 8, LMUL=2
vsetvli t0, a0, e32, mf2  # SEW=32, LMUL=1/2
```

Intricacies of the `vsetvli` instruction:

rd	rs1	AVL value	Effect on vl
-	!x0	Value in rs1	Gives requested vl or VLMAX, whichever is greater
!x0	x0	INT_MAX	Sets vl to VLMAX
x0	x0	Old vl	Keeps old vl, but can change data type (SEW, LMUL)

8.1.4.2 Load/Store instructions

Vector Load/Store

```
# Vector loads and stores

# vd destination, rs1 base address, vm is mask encoding (v0.t or <missing>)
 vle8.v vd, (rs1), vm  # 8-bit unit-stride load
vle16.v vd, (rs1), vm  # 16-bit unit-stride load
vle32.v vd, (rs1), vm  # 32-bit unit-stride load
vle64.v vd, (rs1), vm  # 64-bit unit-stride load

# vs3 store data, rs1 base address, vm is mask encoding (v0.t or <missing>)
 vse8.v vs3, (rs1), vm # 8-bit unit-stride store
vse16.v vs3, (rs1), vm # 16-bit unit-stride store
vse32.v vs3, (rs1), vm # 32-bit unit-stride store
vse64.v vs3, (rs1), vm # 64-bit unit-stride store
```

Vector strided Load/Store

```
# Vector strided loads and stores

# vd destination, rs1 base address, rs2 byte stride
 vlse8.v vd, (rs1), rs2, vm  # 8-bit strided load
vlse16.v vd, (rs1), rs2, vm  # 16-bit strided load
vlse32.v vd, (rs1), rs2, vm  # 32-bit strided load
vlse64.v vd, (rs1), rs2, vm  # 64-bit strided load

# vs3 store data, rs1 base address, rs2 byte stride
 vsse8.v vs3, (rs1), rs2, vm # 8-bit strided store
vsse16.v vs3, (rs1), rs2, vm # 16-bit strided store
vsse32.v vs3, (rs1), rs2, vm # 32-bit strided store
vsse64.v vs3, (rs1), rs2, vm # 64-bit strided store
```

8.1.4.3 Arithmetic, logical and move instructions

General vector binary arithmetic operation syntax

```
# Assembly syntax pattern for vector binary arithmetic instructions

# Operations returning vector results
vop.vv vd, vs2, vs1 # vector-vector     vd[i] = vs2[i] op vs1[i]
vop.vx vd, vs2, rs1 # vector-scalar     vd[i] = vs2[i] op x[rs1]
vop.vi vd, vs2, imm # vector-immediate  vd[i] = vs2[i] op imm
```

```
# Assembly syntax pattern for vector widening arithmetic instructions

# Double-width result, two single-width sources: 2*SEW = SEW op SEW
vwop.vv vd, vs2, vs1 # vector-vector  vd[i] = vs2[i] op vs1[i]
vwop.vx vd, vs2, rs1 # vector-scalar  vd[i] = vs2[i] op x[rs1]

# Double-width result, first source double-width, second source single-width: 2*SEW =
2*SEW op SEW
vwop.wv vd, vs2, vs1 # vector-vector  vd[i] = vs2[i] op vs1[i]
vwop.wx vd, vs2, rs1 # vector-scalar  vd[i] = vs2[i] op x[rs1]
```

Vector Single-Width Integer Add, Subtract and Multiply

```
# Integer adds
vadd.vv vd, vs2, vs1  # Vector-vector
vadd.vx vd, vs2, rs1  # vector-scalar
vadd.vi vd, vs2, imm  # vector-immediate

# Integer subtract
vsub.vv vd, vs2, vs1  # Vector-vector
vsub.vx vd, vs2, rs1  # vector-scalar

# Signed multiply, returning low bits of product
vmul.vv vd, vs2, vs1  # Vector-vector
vmul.vx vd, vs2, rs1  # vector-scalar

# Signed multiply, returning high bits of product
vmulh.vv vd, vs2, vs1 # Vector-vector
vmulh.vx vd, vs2, rs1 # vector-scalar

# Unsigned multiply, returning high bits of product
vmulhu.vv vd, vs2, vs1  # Vector-vector
vmulhu.vx vd, vs2, rs1  # vector-scalar

# Signed(vs2)-Unsigned multiply, returning high bits of product
vmulhsu.vv vd, vs2, vs1 # Vector-vector
vmulhsu.vx vd, vs2, rs1 # vector-scalar
```

Vector Widening Integer Add, Subtract and Multiply

```
# Widening unsigned
vwaddu.vv vd, vs2, vs1  # vector-vector, all other variants possible

# Widening signed
vwadd.vv vd, vs2, vs1   # vector-vector, all other variants possible

# Widening signed-integer multiply
vwmul.vv vd, vs2, vs1, vm  # vector-vector
vwmul.vx vd, vs2, rs1, vm  # vector-scalar

# Widening unsigned-integer multiply
vwmulu.vv vd, vs2, vs1, vm # vector-vector
vwmulu.vx vd, vs2, rs1, vm # vector-scalar
```

Vector Multiply-Add (multiply-accumulate)

```
# Integer multiply-add, overwrite addend
vmacc.vv vd, vs1, vs2  # vd[i] = +(vs1[i] * vs2[i]) + vd[i]
vmacc.vx vd, rs1, vs2  # vd[i] = +(x[rs1] * vs2[i]) + vd[i]

# Integer multiply-add, overwrite multiplicand
vmadd.vv vd, vs1, vs2  # vd[i] = (vs1[i] * vd[i]) + vs2[i]
vmadd.vx vd, rs1, vs2  # vd[i] = (x[rs1] * vd[i]) + vs2[i]
```

Vector Sign-extend/Zero-extend, Reduce and Move

```
# Scalar -> Vector
vmv.v.v vd, vs1  # vd[i] = vs1[i]
vmv.v.x vd, rs1  # vd[i] = x[rs1]
vmv.v.i vd, imm  # vd[i] = imm

# Vector -> Scalar
vmv.x.s rd, vs1  # rd = vs1[0]

# Extension:
vzext.vf2 vd, vs2  # Zero-extend SEW/2 source to SEW destination
vsext.vf2 vd, vs2  # Sign-extend SEW/2 source to SEW destination
vzext.vf4 vd, vs2  # Zero-extend SEW/4 source to SEW destination
vsext.vf4 vd, vs2  # Sign-extend SEW/4 source to SEW destination

# Simple reductions, where [*] denotes all active elements:
vredsum.vs vd, vs2, vs1    # vd[0] = sum( vs1[0] , vs2[*] )

# Unsigned sum reduction into double-width accumulator
vwredsumu.vs vd, vs2, vs1  # 2*SEW = 2*SEW + sum(zero-extend(SEW))

# Signed sum reduction into double-width accumulator
vwredsum.vs vd, vs2, vs1   # 2*SEW = 2*SEW + sum(sign-extend(SEW))
```

8.1.4.4 Code examples

```
# Example: Load 16-bit values, widen multiply to 32b, shift 32b result right by 3,
store 32b values.
# On entry:
# a0 holds the total number of elements to process
# a1 holds the address of the source array
# a2 holds the address of the destination array
loop:
    vsetvli a3, a0, e16, m4, ta, ma # vtype = 16-bit integer vectors; a3 = vl (number
of elements this iteration)
    vle16.v v4, (a1)        # Get 16b vector
    slli t1, a3, 1          # Multiply # elements by 2 bytes (sizeof 16-bit int)
    add a1, a1, t1          # Bump pointer
    vwmul.vx v8, v4, x10    # Widening multiply into 32b in <v8--v15>

    vsetvli x0, x0, e32, m8, ta, ma # Operate on 32b values LMUL 8
    vsrl.vi v8, v8, 3       # Shift by 3
    vse32.v v8, (a2)        # Store vector of 32b elements
    slli t1, a3, 2          # Multiply # elements  by 4 (sizeof int)
    add a2, a2, t1          # Bump pointer
    sub a0, a0, a3          # Decrement count by vl
    bnez a0, loop           # Any more elements to process?
```

8.2 Lab 2: High Level Synthesis

8.2.1 Theoretical Questions (for practice only)

8.2.1.1 Lab 1

What are memory-mapped peripherals?

Communication with the peripherals is done over memory regions. Hardware registers are mapped into the address space. It is handled like *load*/*store* (done by the LSU (LoadStore Unit)).

What is the reason for declaring register references as `volatile`?

So the compiler does not optimize them away (when only only read, not written, they're deleted). Also order of memory access is retained.

What are the contents of `Uart/uart_t` struct in the UART driver library?

- Pointer to all control status registers
- And a custom callback function.

What is *busy waiting*

Waiting in a `while` loop, where no operation is performed just waiting/checking the value.

Is polling, in general, a bad design choice?

No
- If you have a predictable system where everything happens in known cycles.
- If the system needs to be cycle-accurate.
  - interrupts have an overhead

Why is there still a system tick reported, while the processing system is in busy wait state? (Bare-metal system)

Because timer interrupts still get triggered by the timer.

What is cross compilation?

If the host operating system/architecture is not the same as the target operating system/architecture.

What is C-code validation and C-code synthesis in Vivado-HLS?

Validation is the process where the C-code gets tested on a testbench to see if the output is the same as expected. Checks that the C algorithms performs the correct operation.
Synthesis is building a hardware component which implements the function defined in the C-code.

What is *Local* vs *Global* optimizations?

local:

- Limited to one *Basic Block* or loop iteration.
- Techniques:
  - local scheduling
  - local resource sharing
  - peephole optimization

global:

- Works on the entire *Control Data Flow Graph (CDFG)*.
- Focuses on parallelism and reducing overall latency.
- Techniques:
  - loop unrolling
  - loop pipelining
  - function inlining
  - array access optimization

What are *Directives* in Vivado-HLS= List the available directies and provide a one-line descripton of each.

List is in Section 8.2.3.

How does *loop-unrolling* works in optimizing the design?

Allows loop iterations to be executed in parallel, reducing the overall cycles needed.

What is the difference between `DATAFLOW` and functional-level `PIPELINE` directives?

`DATAFLOW` does additional optimizations, not only pipelining.

What is the difference between `LOOP_FLATTEN` and `UNROLL` directives?

`FLATTEN` changes the loop hierachy (in code and RTL) but does not influence the number of executed iterations, unrolling minimizes the iterations.

What were the indicators that led to conclusion that the inital design was not pipelined in *Step 3* of the tutorial?

1. Latency and initialization interval of the top block where equal.
2. For all loops it was reported that they were not pipelined.

Why didn't we apply `PIPELINE`-directive directly to the `dict_1d` in *Step 4* of the tutorial?

Then the `dct_coeff` table would need to be duplicated for every pipeline stage.

What were the indications that led to the conlcusion that an imperfect loop nest was blocking the loop-pipelining in *Step 5* of the tutorial?

There was still a loop hierachy despite loop flattening was specified.

What were the inidcator that led to the conlcusion that data-dependency was *"bottlenecking"* the design in *Step 6*?

1. There was a warning about exactly that.
2. DCT loop was taking many more cycles than all other loops.

Describe briefly how an HDL design from Vivado-HLS can be exported to Vivado's IP-Catalog?

Solution → *Export RTL*

8.2.2 Definitoins

8.2.2.1 Modul Hierachy

- **Latency:** the total time it takes for the function to finish one execution (start to finish)
- **Interval:** time you must wait before you can call the function again
  - if not pipelined:

    $$\text{latency} = \text{interval}$$

  - if pipelined: $\text{interval} = 1$
- **Look-Up Table (LUT):** the fundamental programmable logic gate used for boolean operations and small distributed memory (LUTRAM)
- **Flip-Flop (FF):** a 1-bit storage register used to hold state and sychronize data between pipeline stages
- **Digital Signal Prcessor (DSP):** a hardened solicon block optimized for high-speed, efficient arithmetic without using genal logic

8.2.2.2 Performance Profile

The following metrices describe the behaviour of a specific loop (inside the function)

- **Iteration Latency:** the time it takes to complete one single iteration of the loop
- **Initiation Interval:** the time between two loop iterations
  - (if sequential: = iteration latency, if pipelined = 1).
- **Trip Count:** number of times the loop is executed

8.2.2.3 Total Time

- **Sequential:** $\text{latency} = \text{trip count} * \text{iteration latency}$
- **Pipelined:** $\text{latency} = (\text{trip count} * \text{initiation iterval}) + \text{iteration latency}$

8.2.3 Directives

⚠ Not all of them are important

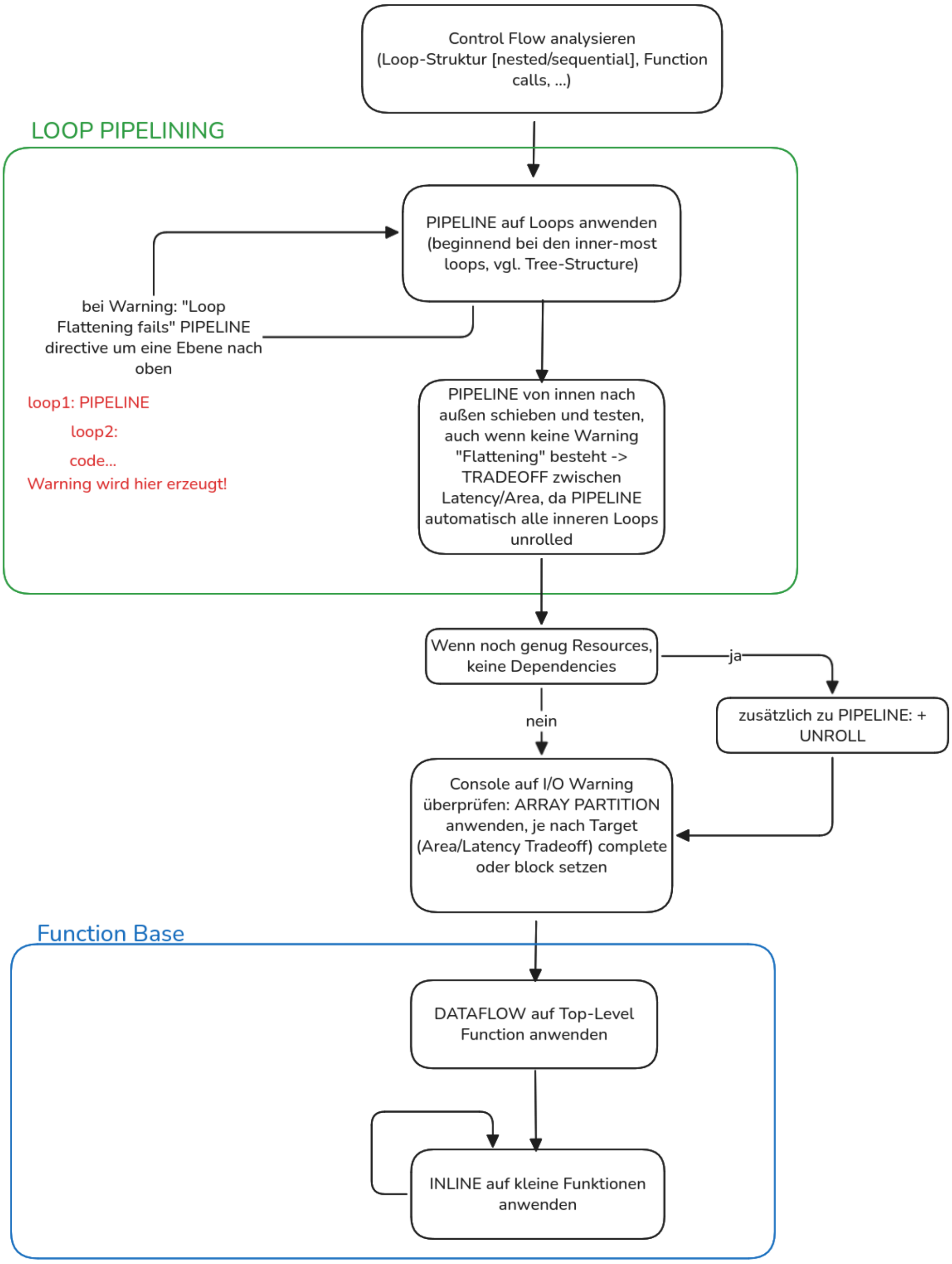

Control Flow analysieren (Loop-Struktur [nested/sequential], Function calls, ...)
LOOP PIPELINING
PIPELINE auf Loops anwenden (beginnend bei den inner-most loops, vgl. Tree-Structure)
bei Warning: "Loop Flattening fails" PIPELINE directive um eine Ebene nach oben
loop1: PIPELINE
loop2:
code...
Warning wird hier erzeugt!
PIPELINE von innen nach außen schieben und testen, auch wenn keine Warning "Flattening" besteht -> TRADEOFF zwischen Latency/Area, da PIPELINE automatisch alle inneren Loops unrolled
Wenn noch genug Resources, keine Dependencies
ja
zusätzlich zu PIPELINE: + UNROLL
nein
Console auf I/O Warning überprüfen: ARRAY PARTITION anwenden, je nach Target (Area/Latency Tradeoff) complete oder block setzen
Function Base
DATAFLOW auf Top-Level Function anwenden
INLINE auf kleine Funktionen anwenden

Figure 8.2: Flowgraph how to work towards targets

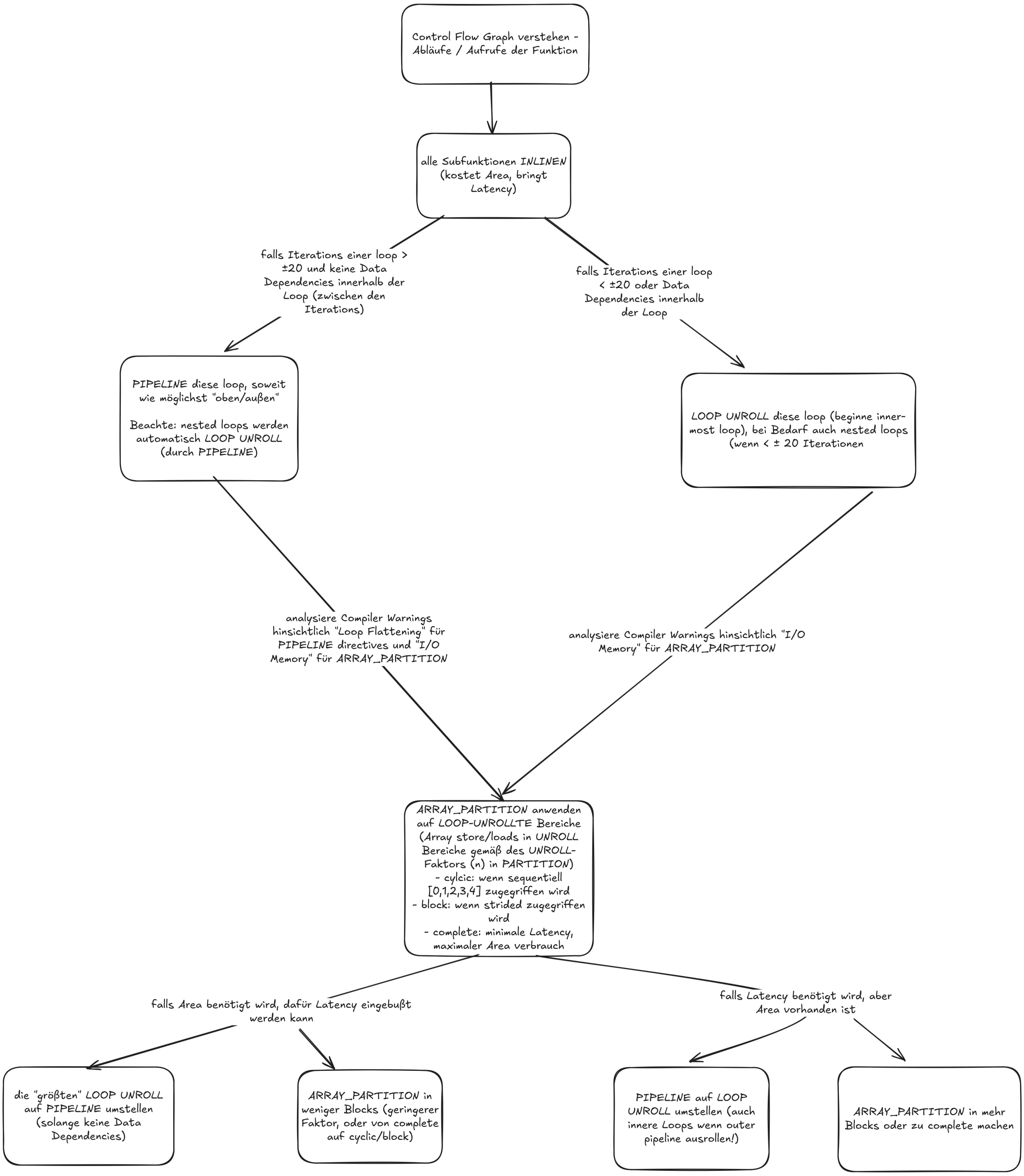

Control Flow Graph verstehen - Abläufe / Aufrufe der Funktion
alle Subfunktionen INLINEN (kostet Area, bringt Latency)
falls Iterations einer loop > ±20 und keine Data Dependencies innerhalb der Loop (zwischen den Iterations)
falls Iterations einer loop < ±20 oder Data Dependencies innerhalb der Loop
PIPELINE diese loop, soweit wie möglichst "oben/außen"
Beachte: nested loops werden automatisch LOOP UNROLL (durch PIPELINE)
LOOP UNROLL diese loop (beginne inner-most loop), bei Bedarf auch nested loops (wenn < ± 20 Iterationen
analysiere Compiler Warnings hinsichtlich "Loop Flattening" für PIPELINE directives und "I/O Memory" für ARRAY_PARTITION
analysiere Compiler Warnings hinsichtlich "I/O Memory" für ARRAY_PARTITION
ARRAY_PARTITION anwenden auf LOOP-UNROLLTE Bereiche (Array store/loads in UNROLL Bereiche gemäß des UNROLL-Faktors (n) in PARTITION)
- cylcic: wenn sequentiell [0,1,2,3,4] zugegriffen wird
- block: wenn strided zugegriffen wird
- complete: minimale Latency, maximaler Area verbrauch
falls Area benötigt wird, dafür Latency eingebußt werden kann
falls Latency benötigt wird, aber Area vorhanden ist
die "größten" LOOP UNROLL auf PIPELINE umstellen (solange keine Data Dependencies)
ARRAY_PARTITION in weniger Blocks (geringerer Faktor, oder von complete auf cyclic/block)
PIPELINE auf LOOP UNROLL umstellen (auch innere Loops wenn outer pipeline ausrollen!)
ARRAY_PARTITION in mehr Blocks oder zu complete machen

Figure 8.3: Flowgraph how to work towards targets (Extended)

8.2.3.1 `ALLOCATION`

Limits the number of RTL instances of specific functions and loop iterations.

- 👍 Reduces area
- 👎 Also reduces performance

8.2.3.2 `ARRAY_PARTITION`

Partition an array into multiple smaller arrays, use many smaller memories instead of a single large one.

- 👍 More read/write ports ⇒ better throughput
- 👎 More instances ⇒ more area
- 💡 small arrays ⇒ use `complete` partition

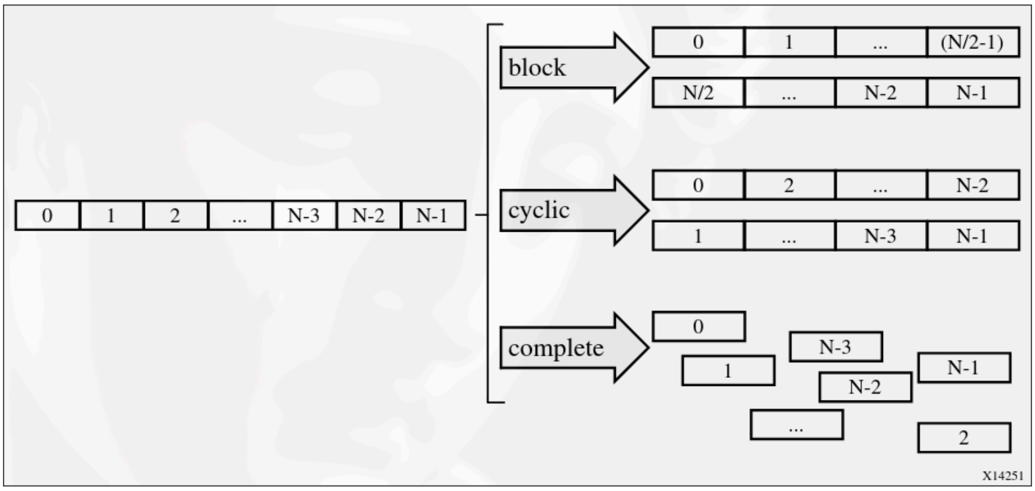


Figure 8.4: How different partition modes work

- *Block*
  - use if you have strided array accesses
- *Cyclic*
  - ⚠️ if you have successiv array accesses you should use that
    - because then successive elements are in different lanes and are non-blocking
- *Complete*

8.2.3.3 DATAFLOW

The HLS tool seeks to minimize latency and improve concurrency as much as data-dependencies allow. This way, functions or loop iterations can operate in parallel (interleaving)

- ⚠️ can only be applied to the **top-level function**
- 👍 Better throughput
- 👍 Lower latency
- 👎 More area usage

8.2.3.4 DEPENDENCE

Explicitly define dependencies between loop iterations since sometimes the automatic analysis is too conservative.

- 👍 Potentially more optimal solution
- 👎 Potentially wrong solution due to eliminated *true dependencies*

8.2.3.5 EXPRESSION_BALANCE

Rearrange assotiative and commuatative operations; *on* by default for integers; *off* by default for floats.

8.2.3.6 FUNCTION_INSTANTIATE

Use different RTL instances for different calls to the same function.

8.2.3.7 INLINE

inlines a function, so it's not a separate RTL entity.

- 👍 Better global optimization
- 👎 RTL block cannot be reused for other calls to the function.

8.2.3.8 LATENCY

Specifies minimum and/or maximum latency.

8.2.3.9 LOOP_MERGE

Merge consequtive loops into one but only if they have the same bounds / same number of iterations and no data dependency between them.

8.2.3.10 `LOOP_FLATTEN`

Turn nested loops into one single loop.

- 👍 Saves a clock cycle per outer loop iteration

8.2.3.11 `PIPELINE`

Converts a code block into a pipelined instance with a given instantiation interval.

- 👍 Better throughput
- 👎 More hardware needed

8.2.3.12 `STABLE`

Specifies that the input argument does not change and does not need to be saved at the start of the function.

8.2.3.13 `STREAM`

Implements an array as a FIFO instead of RAM. More efficient in some cases.

8.2.3.14 `UNROLL`

Specification for loop unrolling (unroll factor).

8.3 Lab 3: Multi-Core Programming Basics

8.3.1 Open MP & Atomics

8.3.1.1 Directives & Functions

8.3.1.1.1 Useful `omp` Functions

```c
int id = omp_get_thread_num();
int numthreads = omp_get_num_threads();
```

8.3.1.1.2 General

```c
#pragma omp parallel
{
    ...
}
```

- general directive to advise the processor to run it on multiple threads
- will run on all threads if not specified

```c
#pragma omp single
{
    ...
}
```

- the following section will only be done by one thread, and skipped by all others
- synchronization after this block
- only makes sense if this block is **inside** a `omp parallel` block.

8.3.1.1.3 Loops

```
#pragma omp parallel for
for(...)
{
    ...
}


#pragma omp collapse(k)
for(...)
{
    ...
}

#pragma omp reduction(+: sum)
for(...)
{
    ...
}
```

- `omp parallel for`
  - distributes loop iterations among threads inside a parallel region
- `omp collapse(k)`
  - where k is the number of nested loops which will be collapsed
- `omp reduction(op: var)`
  - Performs a reduction operation (sum, max, min, etc.) across all threads transparently
  - `#pragma omp reduction(+: sum)`

8.3.1.1.4 Atomic Operations

```
std::atomic<std::string*> ptr{nullptr};
std::atomic<bool> ready(false);
```

- to use variable as *atomics* they must be defined that way

8.3.1.1.4.1 Memory Orders

As the name suggests, memory orders are used to control the order of memory operations. They are used to prevent reordering of memory operations by the compiler or CPU.

Memory Order	Description	Typical Usage
`relaxed`	instruction reordering by the compiler/cpu is allowed in both directions	
`release`	no reads or writes in the current thread can be reordered **before** this load.	*producer* publishes with release
`acquire`	no reads or writes in the current thread can be reordered **after** this store	*consumer* reads it with acquire

Table 8.9: Memory Orders in C++

Despite the `memory_order` all operations in these variables are atomic, it affects only the order of memory operations.

8.3.1.1.4.2 Storing

```
atomic_store_explicit(&ptr, msg, std::memory_order_relaxed);
atomic_store_explicit(&ready, true, std::memory_order_release);

// same as
ptr.store(msg, std::memory_order_relaxed);
ready.store(true, std::memory_order_release);
```

Code 8.27: Atomic Store

8.3.1.1.4.3 Loading

```
while (!atomic_load_explicit(&ready, std::memory_order_acquire)) {
    // busy-wait
}

p = atomic_load_explicit(&ptr, std::memory_order_acquire);

// same as
p = ptr.load(std::memory_order_acquire);
```

Code 8.28: Atomic Load

8.3.1.2 Threads

```
std::this_thread::yield();
```

- *halts* the current thread for a small time, to give the others chance to run

8.3.2 Tipps

8.3.2.1 Busy Waiting

```
if (ready.load(memory_order_acquire))
{
    ...
}
```

- *busy waiting* on a ready flag is usually done with the ordering `memory_order_acquire`

8.3.2.1.1 Not So Busy Waiting

```
// wait for producer
while (!atomic_load_explicit(&ready, std::memory_order_acquire))
    std::this_thread::yield();
```

- `yield` gice the other thread the chance to run
  - signalizes that this thread has nothing to do for now
- it has to be executed on `this_thread` which is a namespace
  - is similar to `std::this_thread::sleep_for(100ms);` which suspends the current thread for a fixed time

8.3.2.2 False Sharing

False Sharing is when independent variables lay in the memory next to each other and get cached all at once. Now if one of them gets modified, the whole line gets invalidated. This means that other threads have to read from memory again, if they want to use "their" vairable.

A fix for this would be **alignment** with **padding**. This means that between every variable there are put variables which are needed by the same thread. If the threads does not need any other variables, typically it is filled with not-used variables. A example for that would be a struct which holds at the end an array which fills it up.

```
typedef struct {
    uint8_t ready;
    uint8_t padding[CACHE_LINE_SIZE-sizeof(uint8)];
} Thread_Var_t
```

In modern C++ standards there is another builtin function to align, for example entries of an array

```
#define CACHE_LINE_SIZE 64

alignas(CACHE_LINE_SIZE) double partial_sum[MAX_THREADS];
```

Now every entry of `partial_sum` can be modified without invalidating the other entries.

8.3.3 Code examples

8.3.3.1 Producer & Consumer

```
#include <atomic>
#include <thread>
#include <iostream>
#include <string>

std::atomic<std::string*> ptr{nullptr};
std::atomic<bool> ready(false);

void producer() {

    std::string* msg = new std::string("Hello from Producer!");
    atomic_store_explicit(&ptr, msg, std::memory_order_relaxed);
    atomic_store_explicit(&ready, true, std::memory_order_release);
}

void consumer() {
    std::string* p = nullptr;
    while (!atomic_load_explicit(&ready, std::memory_order_acquire)) {
        // busy-wait
    }
    // p = ptr.load(std::memory_order_acquire);
    p = atomic_load_explicit(&ptr, std::memory_order_acquire);
    std::cout << "Consumer received message: " << *p << std::endl;
    delete p;
}

int main() {
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join();
    t2.join();
    return 0;
}
```

Each for every (independent) exchanged message, you have one variable holding the value and one *ready* signal, to signalize the reader, that the value is now valid. Both of them should be atomic. And their memory access according to Table 8.9.

- `std::atomic<std::string*> ptr{nullptr};`
- `std::atomic<bool> ready(false);`
- Producer wrtites with: `memory_order_release`
- Consumer reads with: `memory_order_acquire`

9 Outlines

List of Images

List of Tables

List of Listings

10 Glossarium

ABI – **Application Binary Interface** 9

ALU – **Arithemtic Logic Unit** 17, 50, 50, 56, 56, 57

ASIC – **Application-specific integrated circuit** 56

BB – **Basic Block**: Block which cannot be split up anymore 34

BTB – **Branch Target Buffer** 19, 19, 19, 20, 20

CDFG – **Control Data Flow Graph**

CDG – **Channel Dependence Graph** 102, 102, 102, 103, 128, 128, 128

CC – **Clock Cycles** 18, 18, 19, 19

CFG – **Control Flow Graph** 12, 13

CPI – **Cycles per Instructions** 18

DAG – **Directed Acyclic Grpah** 59

DMA – **Direct Memory Access**

DSP – **Digital Signal Processor**: sometimes meaning *digital signal processing* 56

DFG – **Data Flow Graph** 13, 13

FSM – **Finite State Machine** 3, 54, 56, 69, 69, 70, 71, 72, 73, 73, 74, 74, 74, 74, 74, 82, 82, 83, 83, 84, 84, 127, 127, 127, 127, 127, 127

FU – **Functional Unit** 21, 21, 21, 24, 26, 35, 37, 45, 46, 69, 70, 70, 70, 70, 70, 71, 71, 71, 73, 126, 127

HA – **Hardware Accelerator**: moving parts of the software to the hardware, to be faster and more efficient

HLS – **High Level Synthesis** 3, 3, 52, 52, 53, 54, 54, 55, 55, 55, 55, 55, 56, 57, 58, 59, 60, 126

II – **Initiation Interval**

ISA – **Instruction Set Architecture**: specifies the set of instructions that a processor can execute 8, 27, 27, 47

IR – **Intermediate Representation** 14, 14, 14

LMUL: how many physical registers are bundled into one logical 107, 109, 109

NoC – **Network-on-Chip** 4, 4, 4, 4, 4, 4, 97, 97, 97, 97, 98, 98, 99, 99, 100, 100, 101, 102, 103

PC – **Program Counter** 17, 17, 18, 18, 19

RAW – **Read after Write**: somethimes also called *True Dependency* 17, 17, 25, 25, 25, 25, 26, 32, 33, 33, 33, 125, 125, 125

RTL – **Register Transfer Level** 3, 69, 69, 70, 71, 72, 73, 73, 74

SEW: selected element width 107

SGU – **Sequencing Graph Unit** 59, 59, 59, 59, 59, 61, 73, 77, 77, 126, 126, 127

SIMT – **Single Instruction, Multiple Threads**

SIMD – **Single Instruction, Multiple Data** 3, 45, 50, 50, 50, 50, 50, 107, 107, 107

SoC – **System-on-Chip** 44, 44, 44

SSA – **Static Single Assignment** 12, 12, 13

TRP – **Time-Resource-Plane** 57, 57, 58, 58, 60, 60, 126, 126, 126, 126, 126

TLP – **Thread-Level Parallelsim** 37, 37

VL: actual Vector Length 107

VLEN: Vector Length 107

VLIW – **Very Long Instruction Word** 2, 33, 33, 34, 34, 35, 35

WAR – **Write after Read** 25, 25, 25, 25, 26, 27, 27, 33, 34, 125, 125, 125

WAW – **Write after Write** 25, 25, 25, 25, 26, 27, 27, 33, 34, 125, 125, 125

Dennard Scaling – **Dennard Scaling**: As transistors get smaller, their power density stays constant, so that the power use stays in proportion with area.

DSA – **Domain-Specific Architecture**: A computer architecture specialized for a particular domain or application.

GPU – **Graphics Processing Unit**: A specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device.

Moore's Law – **Moore's Law**: The number of transistors on a microchip doubles about every two years.

NPU – **Neural Processing Unit**: A class of specialized hardware accelerator or microprocessor that is optimized for the execution of artificial intelligence (AI) algorithms.

RISC – **Reduced Instruction Set Computer**

RISC-V – **RISC-V**: An open standard instruction set architecture (ISA) based on reduced instruction set computer (RISC) principles. 2, 8, 9, 9, 9, 81, 92, 127, 128, 128

SMP – **Symmetric Multi-Processor** 79

TPU – **Tensor Processing Unit**: An AI accelerator application-specific integrated circuit (ASIC) developed by Google for neural network machine learning.

vector-chaining: generate interim results (without LSU) so they can be used and forwarded to other units 108, 109