Computer Systems

Advanced Processor Pipelines 1

Daniel Mueller-Gritschneder

15.04.2024

Persons

Prof. Daniel Müller-Gritschneder

Embedded Computing Systems

Florian Kriebel

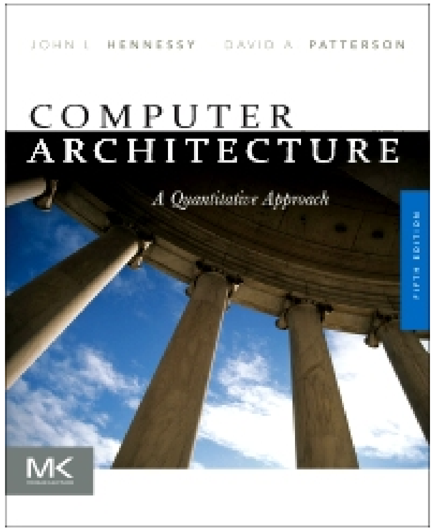
Embedded Computing Systems

Sources

This book covers the basics of how to design a simple in-order scalar processor pipeline in detail in hardware.

- Literature: „Digital Design and Computer Architecture: RISC-V Edition“, by Sarah L. Harris and David Harris
  - https://shop.elsevier.com/books/digital-design-and-computer-architecture-risc-v-edition/harris/978-0-12-820064-3
  - https://pages.hmc.edu/harris/ddca/ddcarv.html (Includes resources for students!)
  - They also provide slideshows – the basis for ours! You can investigate extended version at their website.
- Available at TU's library: https://catalogplus.tuwien.at/permalink/f/qknpf/UTW_alma21139903990003336

Sources



So-called application processors have many additional features:
Branch prediction, Out of order execute, Scoreboard, Superpipelining, Multi-issue, Superscalar, VLIW, Multi-threading, …

Disclaimer: The book provides advanced concepts from real complex processor designs. We only study the concepts at a high level. For simplicity, the used pipeline models in this lecture are reduced strongly in complexity.

But: We will have a look at some current RISC-V processor designs

Literature: „**Computer Architecture A Quantitative Approach”** 5th Edition - September 16, 2011
Authors: John L. Hennessy, David A. Patterson eBook ISBN: 9780123838735
- https://shop.elsevier.com/books/computer-architecture/hennessy/978-0-12-383872-8
- Available at TU's library: https://catalogplus.tuwien.at/permalink/f/8agg25/TN_cdi_askewsholts_vlebooks_9780123838735

Content – Session 1

- Short Recap: RISC-V Assembly
- Five-Stage In-order Scalar Processor Pipeline
    - Pipelined Execution & Stages
    - Data Hazards & Forwarding Paths
    - Control Hazards
- Branch Prediction
    - Static Predictors: Taken / Not taken /BTFNT
    - Branch Target Buffer
    - Dynamic Predictors: 1 bit / 2 bit

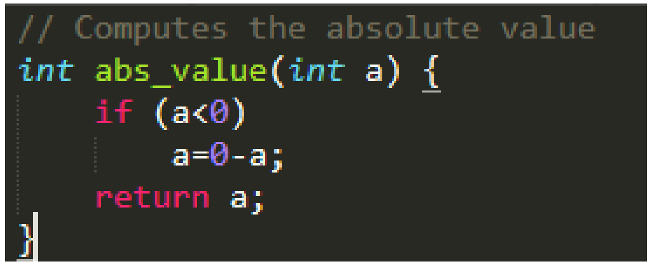
- A look at a real RISC-V processor – CVA6
- A look at a real RISC-V processor – ESP32- C3
- Trap Handling

Optional, not relevant for exam

Short Recap: RISC-V Assembly

Writing a small assembly function: abs_value

- Example C-Code 1



```c
// Computes the absolute value
int abs_value(int a) {
    if (a<0)
        a=0-a;
    return a;
}
```

JALR rd,rs,imm
Behavior:
rd=PC+4
PC=rs+sign_extend(imm)

JALR x0,ra,0
PC=ra

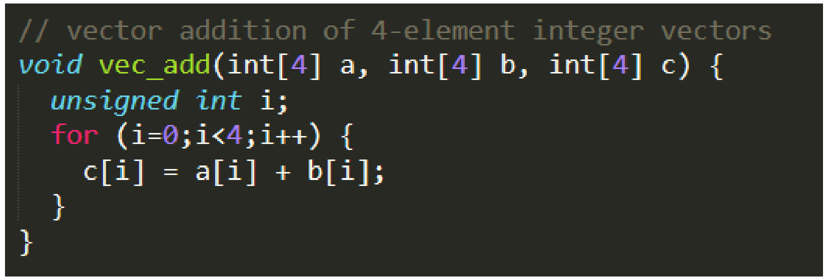
- RISC-V Code
  - According to ABI a is given to the function in register a0
  - The function should also return a in register a0

```
abs_value:
        BGE a0,zero,abs_value_return # if a>=0
        SUB a0,zero,a0                # a=0-a
abs_value_return:
        RET                          # JR ra
```

function return which is a pseudo instruction for
JR ra
which is a pseudo instruction for
JALR x0,ra,0

Writing a small assembly function: vec_add

- Example C-Code 3



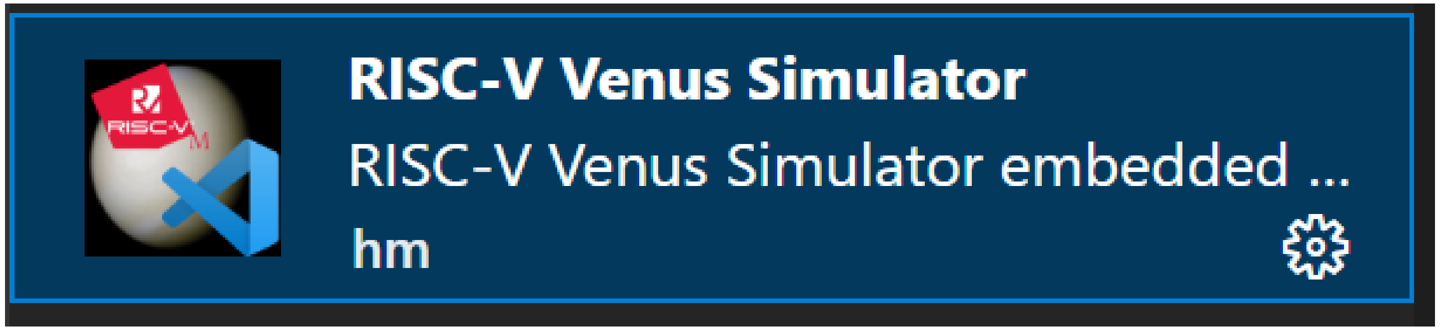
```c
// vector addition of 4-element integer vectors
void vec_add(int[4] a, int[4] b, int[4] c) {
  unsigned int i;
  for (i=0;i<4;i++) {
    c[i] = a[i] + b[i];
  }
}
```

RISC-V Code

```
# base address of a: a0,
# base address of b: a1,
# base address of c: a2,
# i: t0,  constant 4: t3
vec_add:
  LI t0,0       # i=0
  LI t3,4       # t3=4
vec_add_for:
  LW t1,0(a0)    # t1 = a[i]
  LW t2,0(a1)    # t2 = b[i]
  ADD t1,t1,t2   # t1 = a[i] + b[i]
  SW t1,0(a2)    # c[i] = t1
  ADDI a0,a0,4   #next element is base address + 4
  ADDI a1,a1,4   #next element is base address + 4
  ADDI a2,a2,4   #next element is base address + 4
  ADDI t0,t0,1   # i++
  BLTU t0,t3,vec_add_for # for (i < 4)
  RET   # void return
```

RISC-V Simulator

- Visual Studio Code

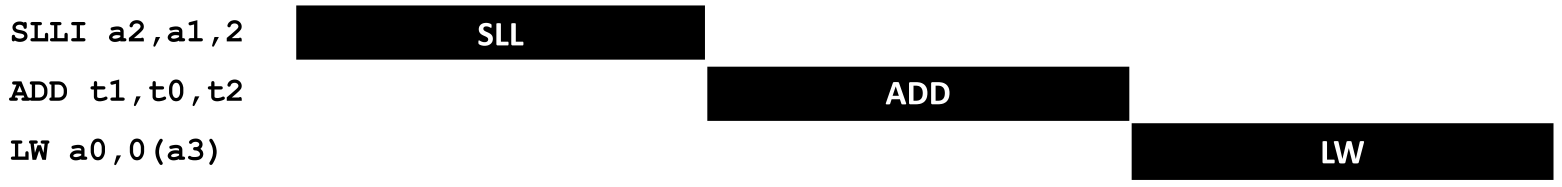

RISC-V Venus Simulator
RISC-V Venus Simulator embedded ...
hm

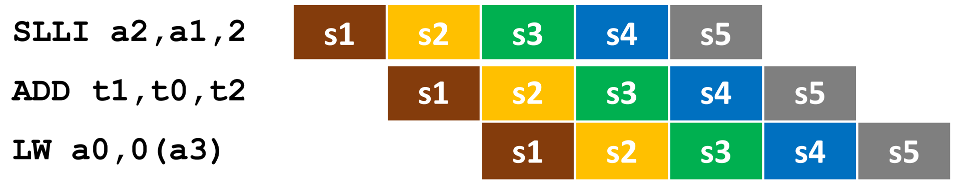
Extensions -> Venus Simulator for RISC-V Assembly

Five-Stage Scalar In-order Processor Pipeline

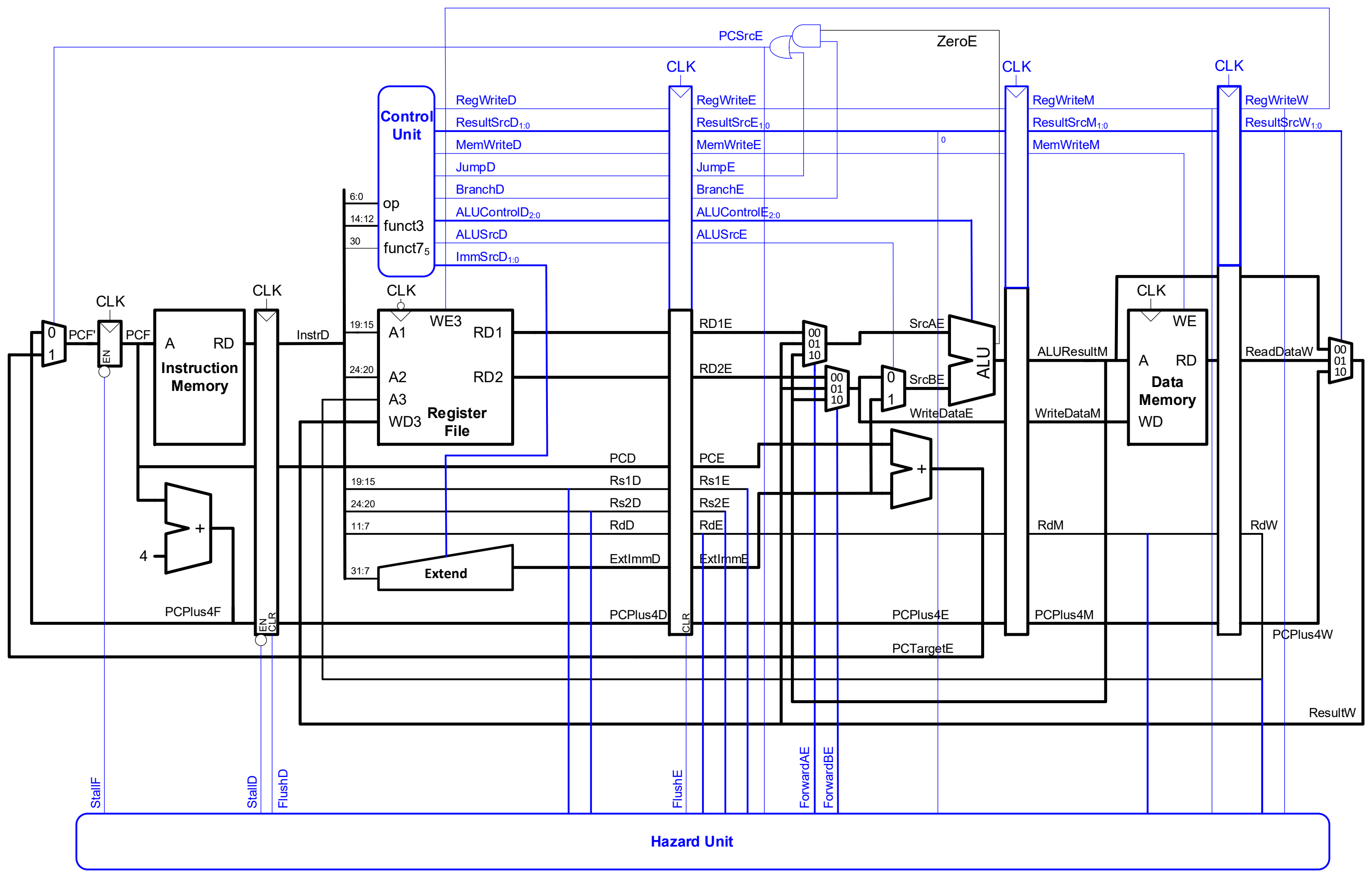
Pipelined execution

- We break down instructions in sub-computations and place them into stages (s)
- We execute the instructions in a pipelined fashion („Fließband“)



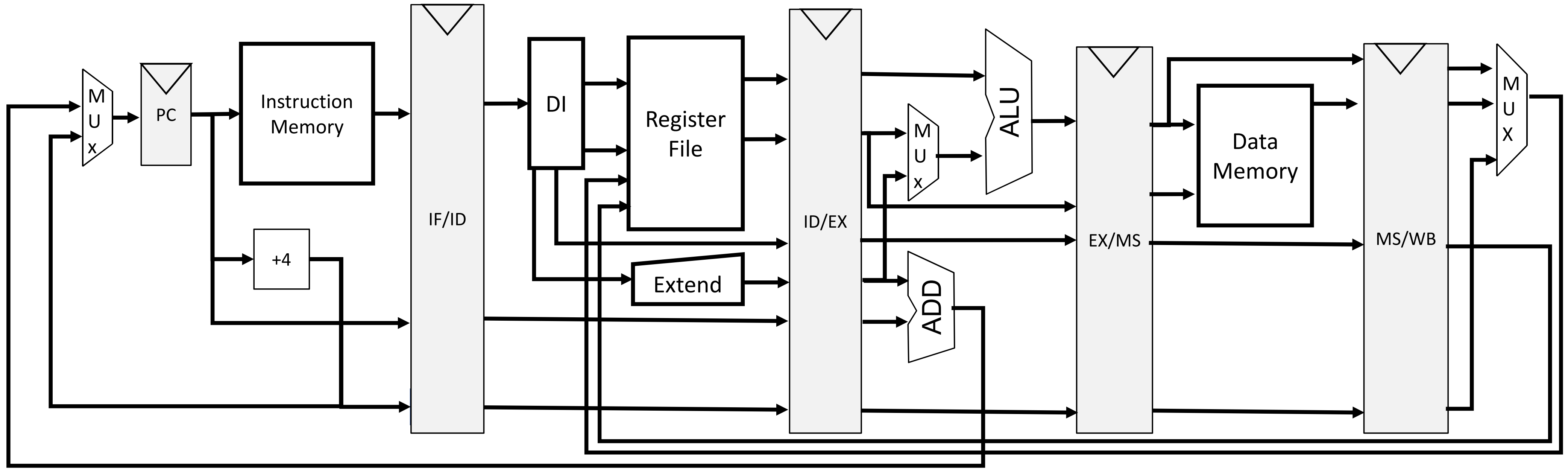


Recap: Five-Stage In-order Scalar Processor Pipeline (Harris & Harris)



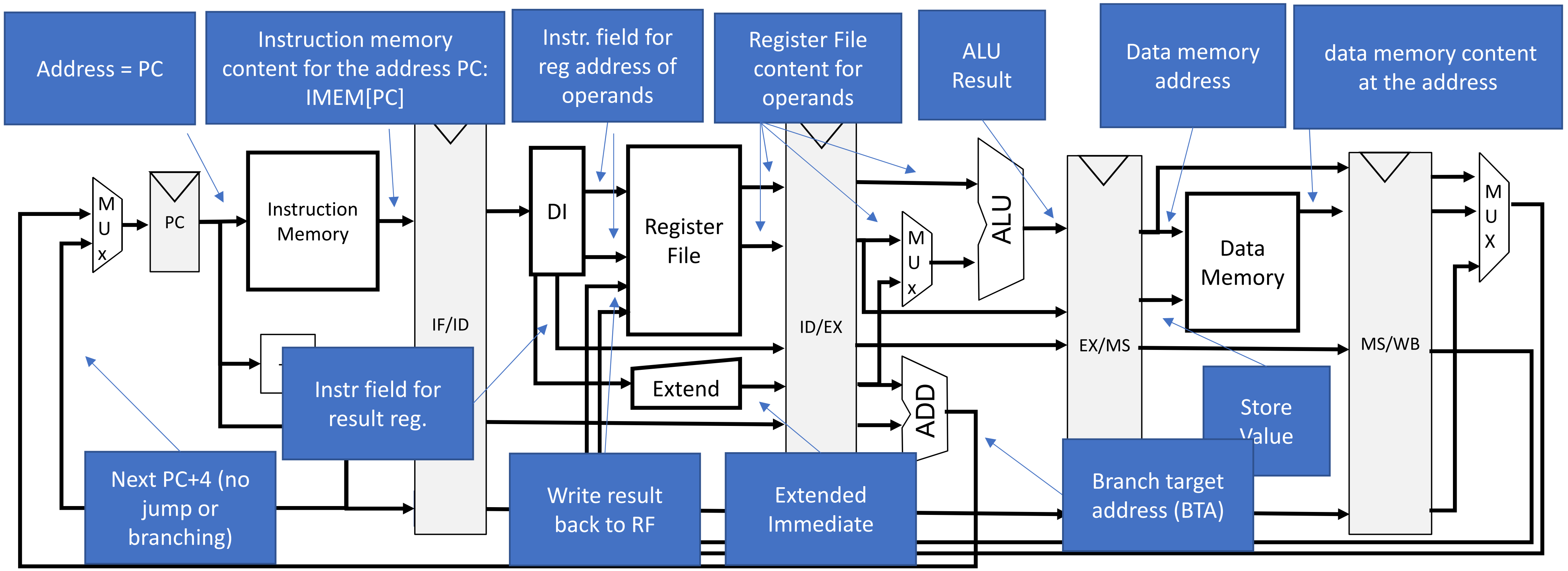
Five-stage Pipeline - Data Signal Busses

- Data path scheme of the pipeline:
  - We omit all control signals.
  - We are only interested how instructions can „flow“ through the pipeline (data signal busses)



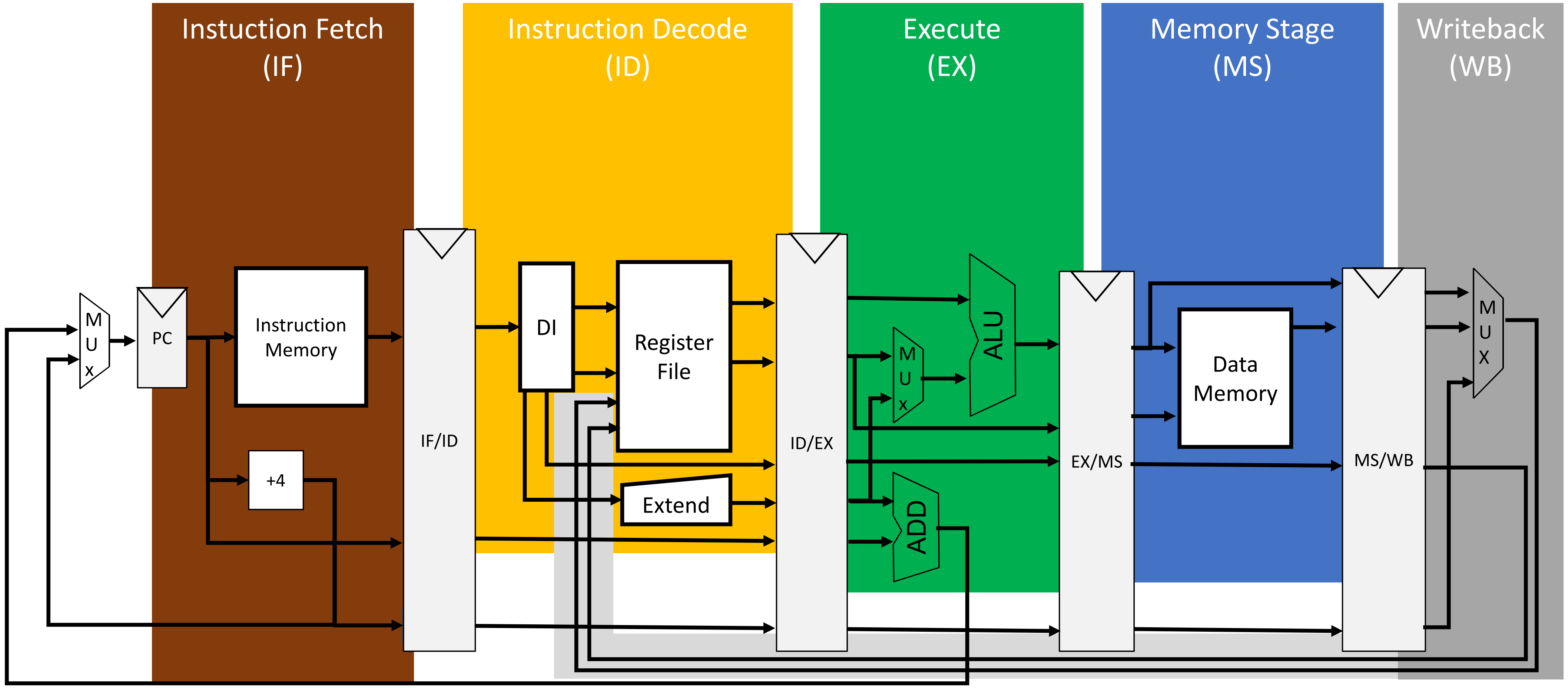
Five-stage Pipeline - Data Signal Busses

- Data path scheme of the pipeline:
  - We omit all control signals.
  - We are only interested how instructions can „flow“ through the pipeline (data signal busses)


Address = PC
Instruction memory content for the address PC: IMEM[PC]
Instr. field for reg address of operands
Register File content for operands
ALU Result
Data memory address
data memory content at the address
M U x
PC
Instruction Memory
IF/ID
DI
Register File
Extend
ID/EX
M U x
ALU
ADD
EX/MS
Data Memory
MS/WB
M U X
Instr field for result reg.
Next PC+4 (no jump or branching)
Write result back to RF
Extended Immediate
Branch target address (BTA)
Store Value

Five-stage Pipeline - Stages

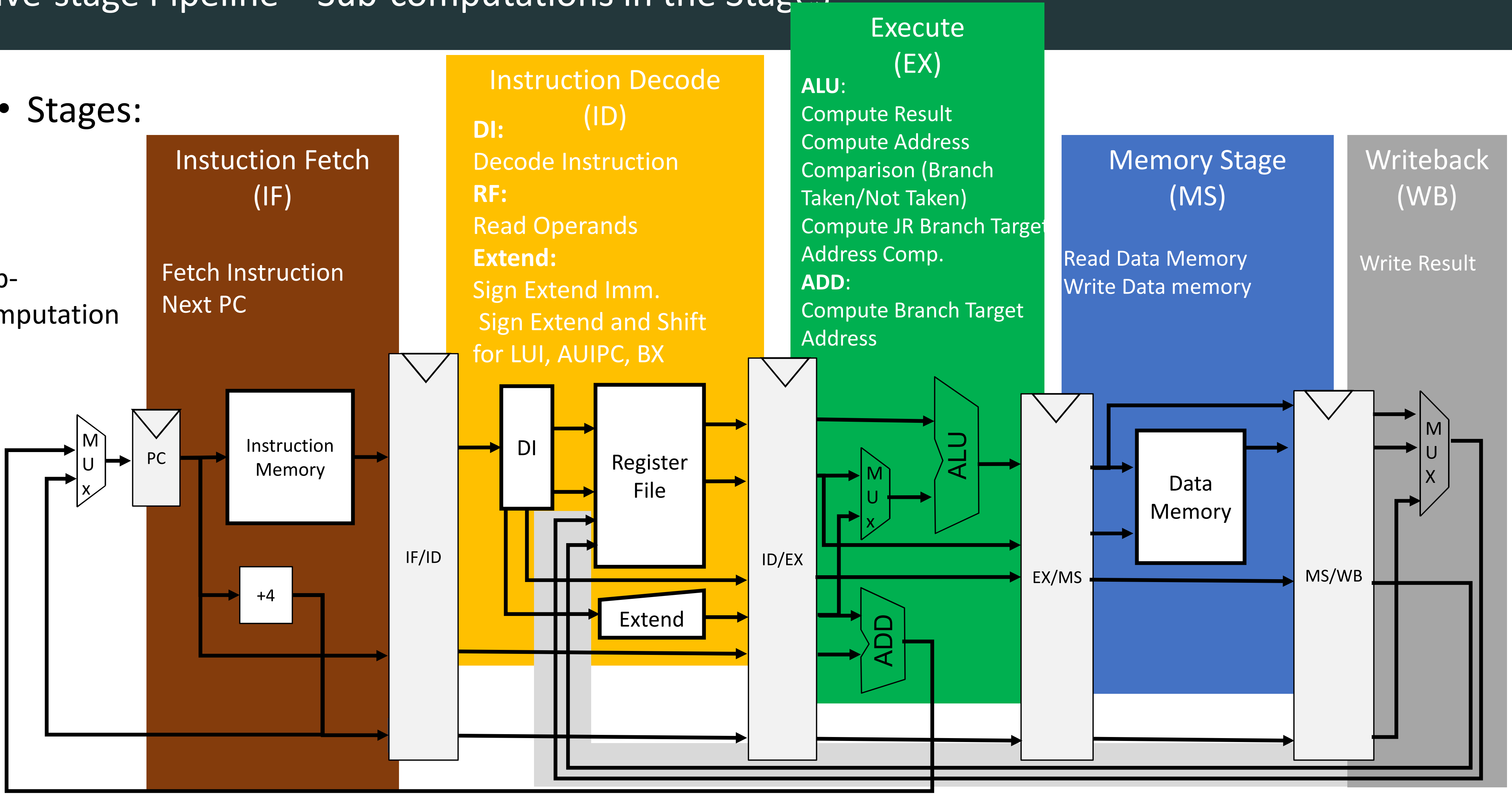
- Stages:



Five-stage Pipeline – Sub-computations in the Stages

- Stages:

Sub-computation



Instuction Fetch (IF)

Fetch Instruction
Next PC

Instruction Decode (ID)

DI:
Decode Instruction
RF:
Read Operands
Extend:
Sign Extend Imm.
Sign Extend and Shift for LUI, AUIPC, BX

Execute (EX)

ALU:
Compute Result
Compute Address
Comparison (Branch Taken/Not Taken)
Compute JR Branch Target
Address Comp.
ADD:
Compute Branch Target Address

Memory Stage (MS)

Read Data Memory
Write Data memory

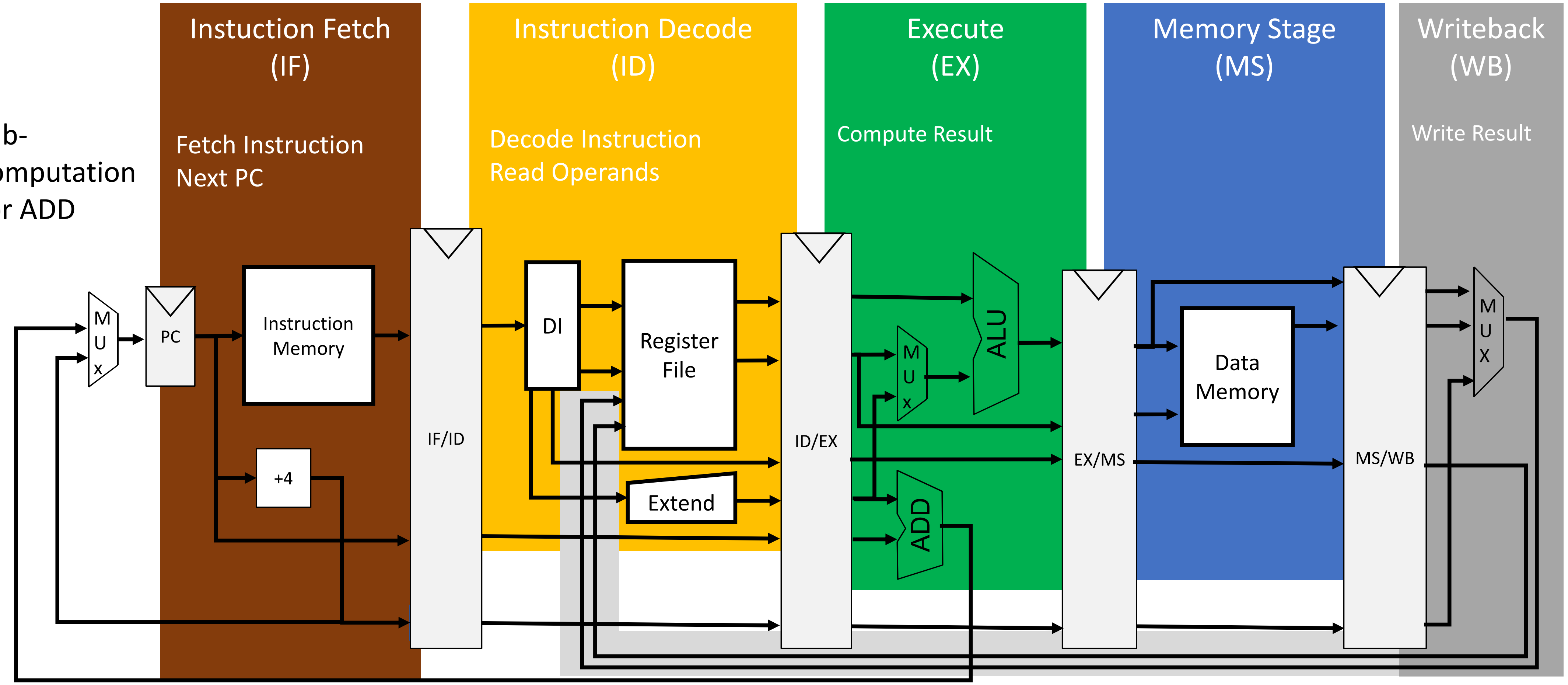
Writeback (WB)

Write Result

Five-stage Pipeline – Sub-computations in the Stages (Example ADD)

- Instructions do not require all subcomputations, e.g. ADD

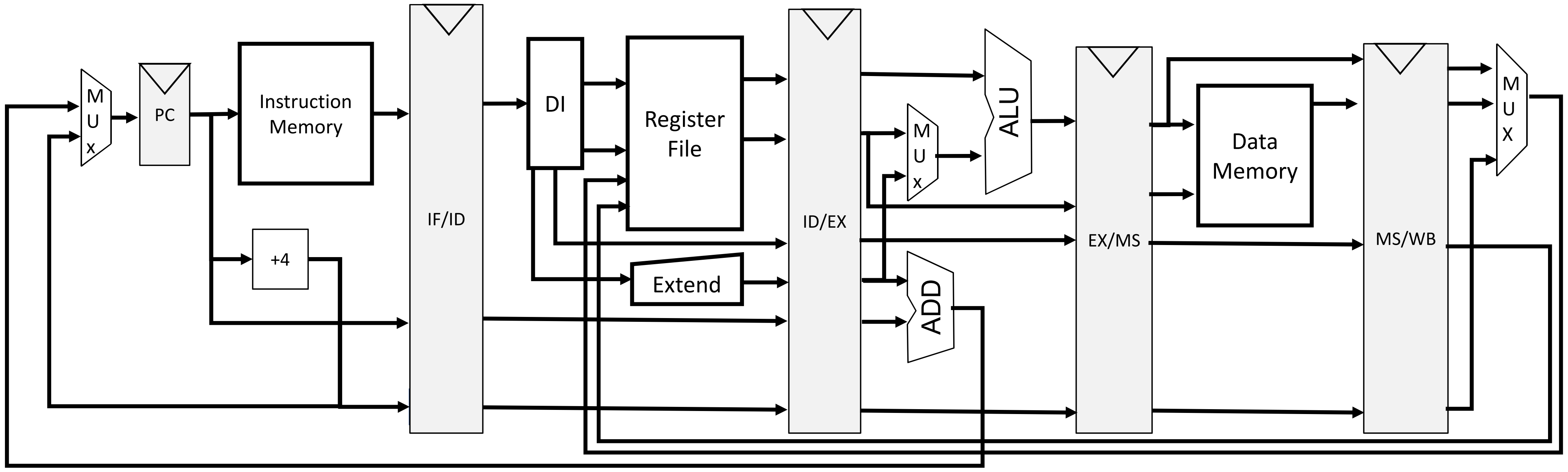
Sub-Computation For ADD



Five-stage Pipeline – Example Program

- Example program

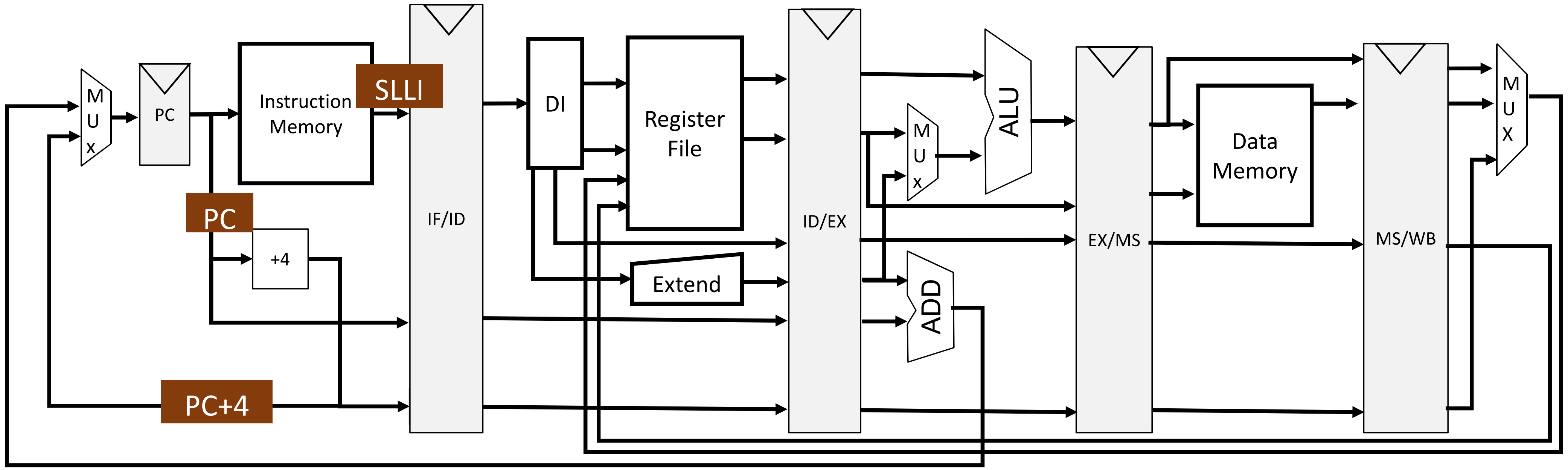
```
#int test1(int *x, int i) {return x[i]+i;}
test1:
  SLLI a2,a1,2  # a2=i*4
  ADD a2,a0,a2   # baseaddr+offset i*4
  LW a0,0(a2)  # a0 = x[i]
  ADD a0,a0,a1   # a0= x[i] + i
  RET
```



Five-stage Pipeline – Example Program – Cycle 1

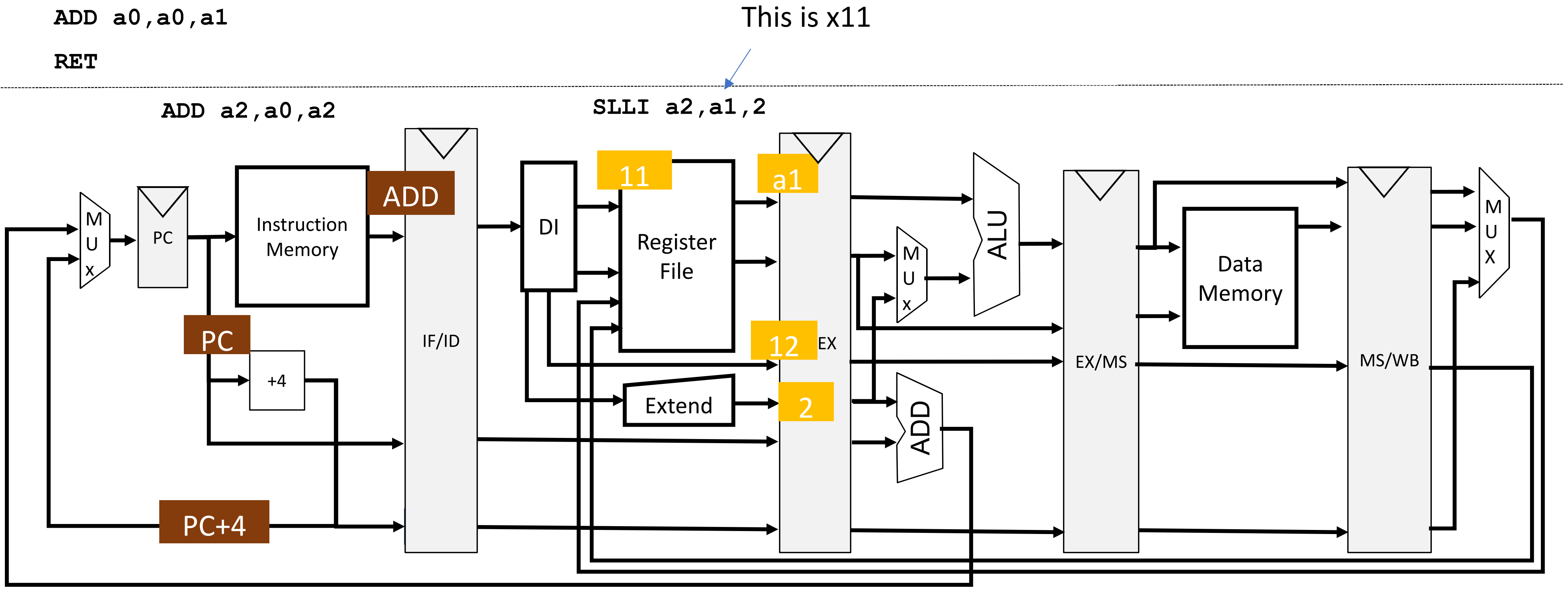
	Cycle 1
`SLLI a2,a1,2`	IF
`ADD a2,a0,a2`	
`LW a0,0(a2)`	
`ADD a0,a0,a1`	
`RET`	

`SLLI a2,a1,2`



Five-stage Pipeline – Example Program – Cycle 2

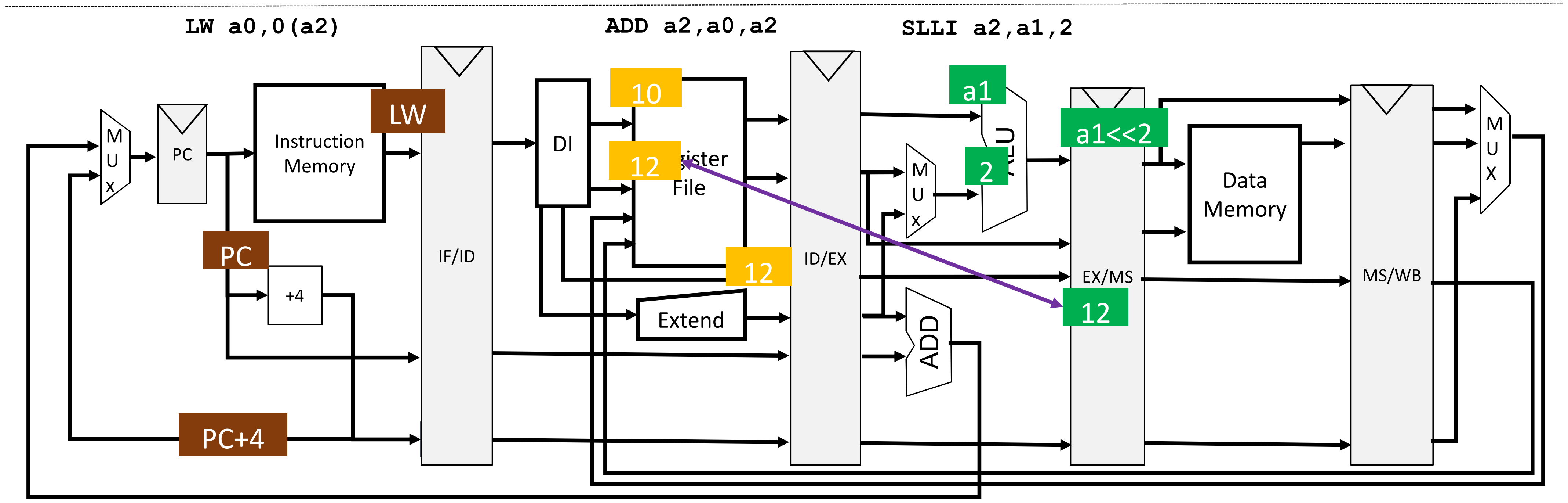
	Cycle 1	Cycle 2
`SLLI a2,a1,2`	IF	ID
`ADD a2,a0,a2`		IF
`LW a0,0(a2)`		
`ADD a0,a0,a1`		
`RET`		



Five-stage Pipeline – Example Program – Cycle 3

	Cycle 1	Cycle 2	Cycle 3
`SLLI a2,a1,2`	IF	ID	EX
`ADD a2,a0,a2`		IF	ID
`LW a0,0(a2)`			IF
`ADD a0,a0,a1`			
`RET`			

Data hazard: a2 not yet updated by SLLI -> Stall ADD because it cannot leave ID stage



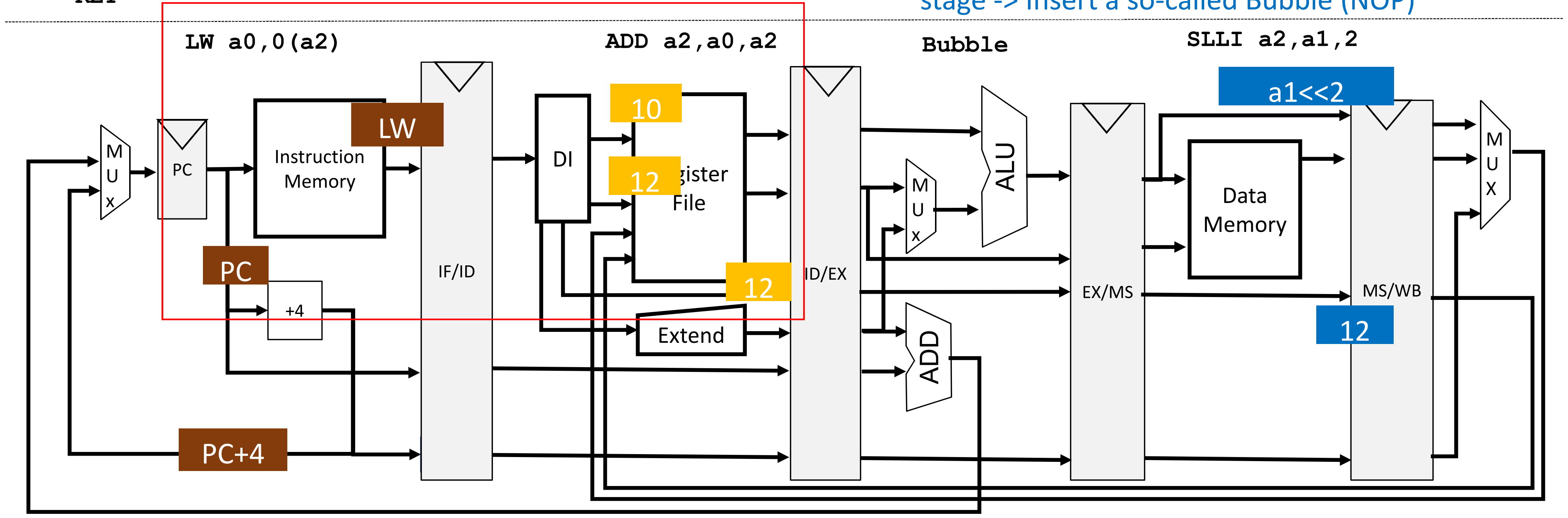
Five-stage Pipeline – Example Program – Cycle 4

	Cycle 1	Cycle 2	Cycle 3	Cycle 4
`SLLI a2,a1,2`	IF	ID	EX	MS
`ADD a2,a0,a2`		IF	ID	stall
`LW a0,0(a2)`			IF	stall
`ADD a0,a0,a1`				
`RET`				

Stalls backpropagate in the pipeline to following instructions

There is no instruction in the execute stage -> Insert a so-called Bubble (NOP)

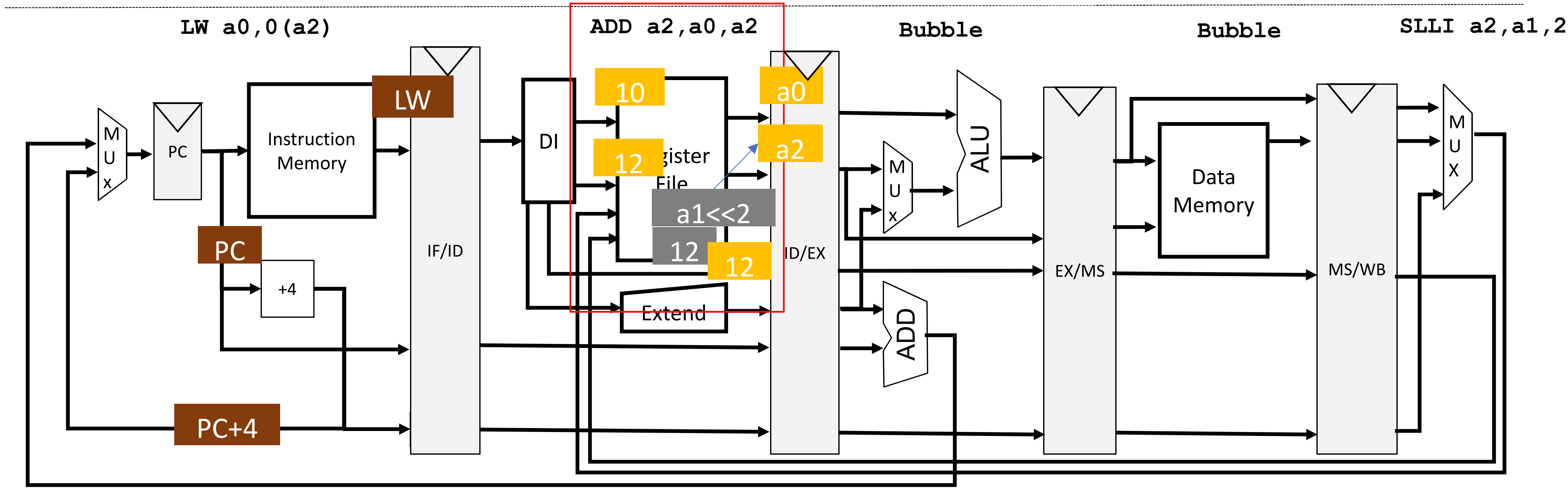
ADD and LW stall



Five-stage Pipeline – Example Program – Cycle 5

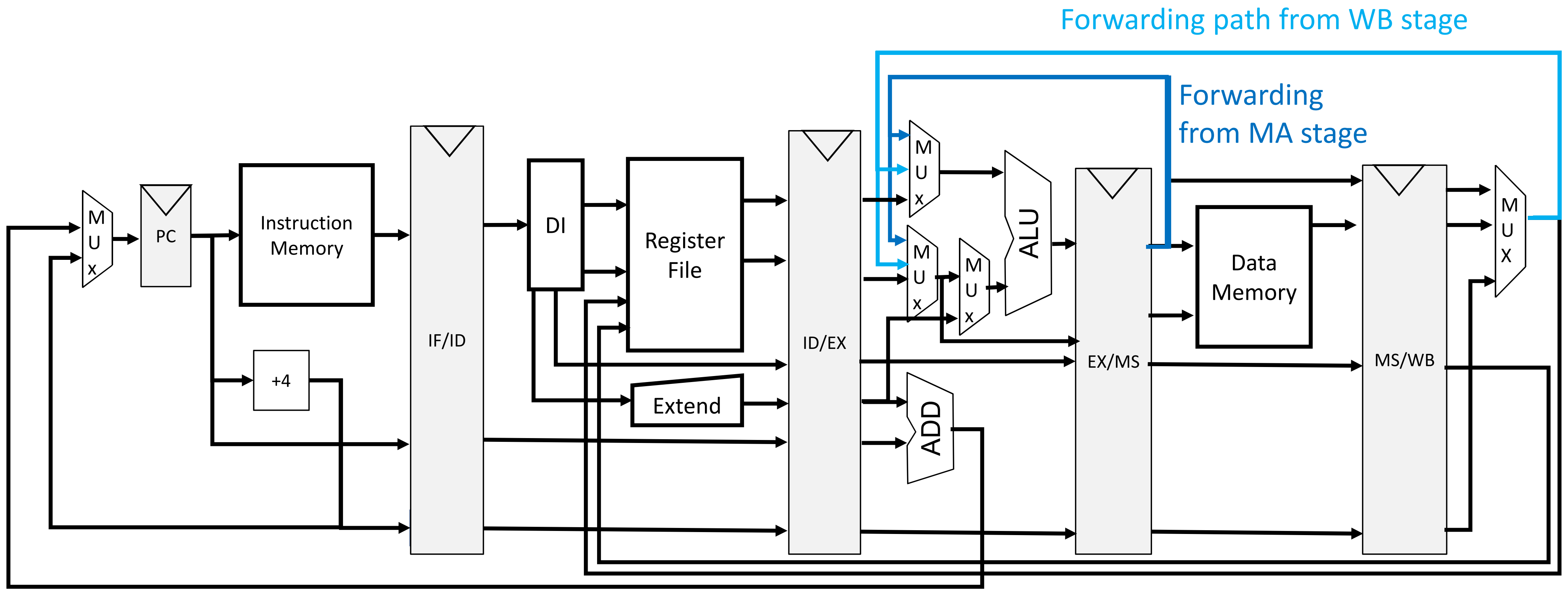
	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
`SLLI a2,a1,2`	IF	ID	EX	MS	WB
`ADD a2,a0,a2`		IF	ID	stall	stall
`LW a0,0(a2)`			IF	stall	stall
`ADD a0,a0,a1`					
`RET`					

ADD can complete ID stage -> stop stalling



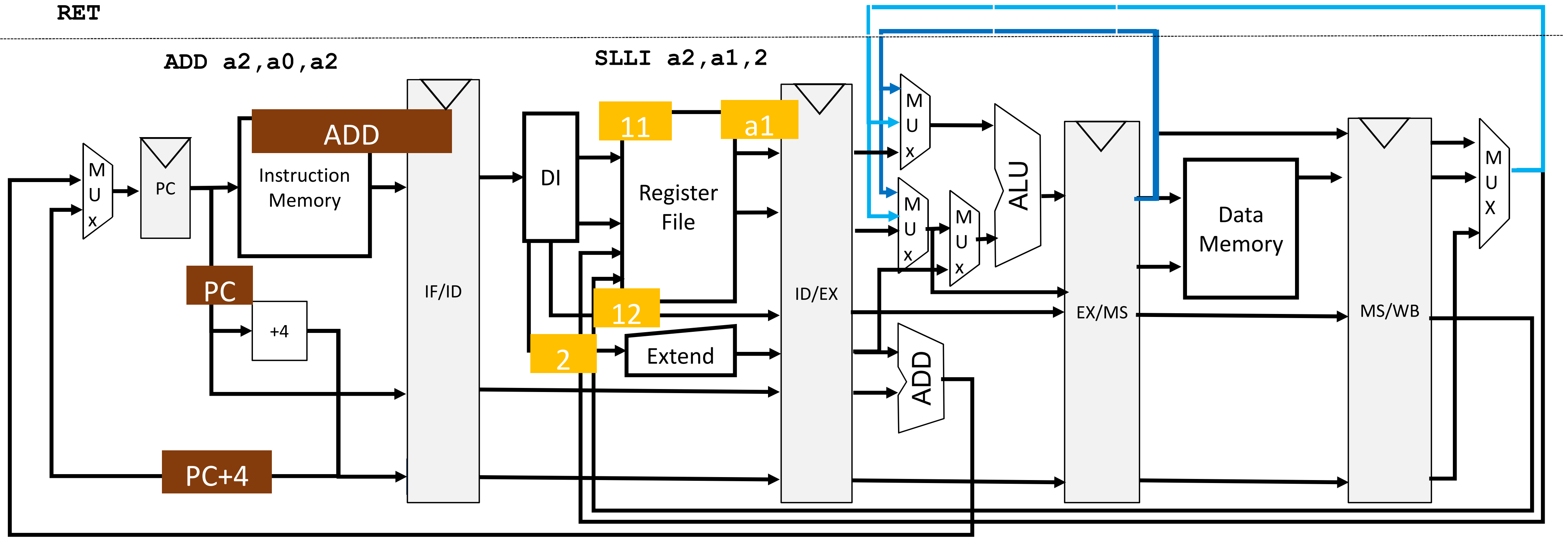
Five-stage Pipeline with Forwarding Path

- Data hazards can be effectively mitigated using a forwarding path
- While named „forwarding path“ the signal buses go „back“ in the pipeline



Five-stage Pipeline with Forwarding Path – Example Program – Cycle 2

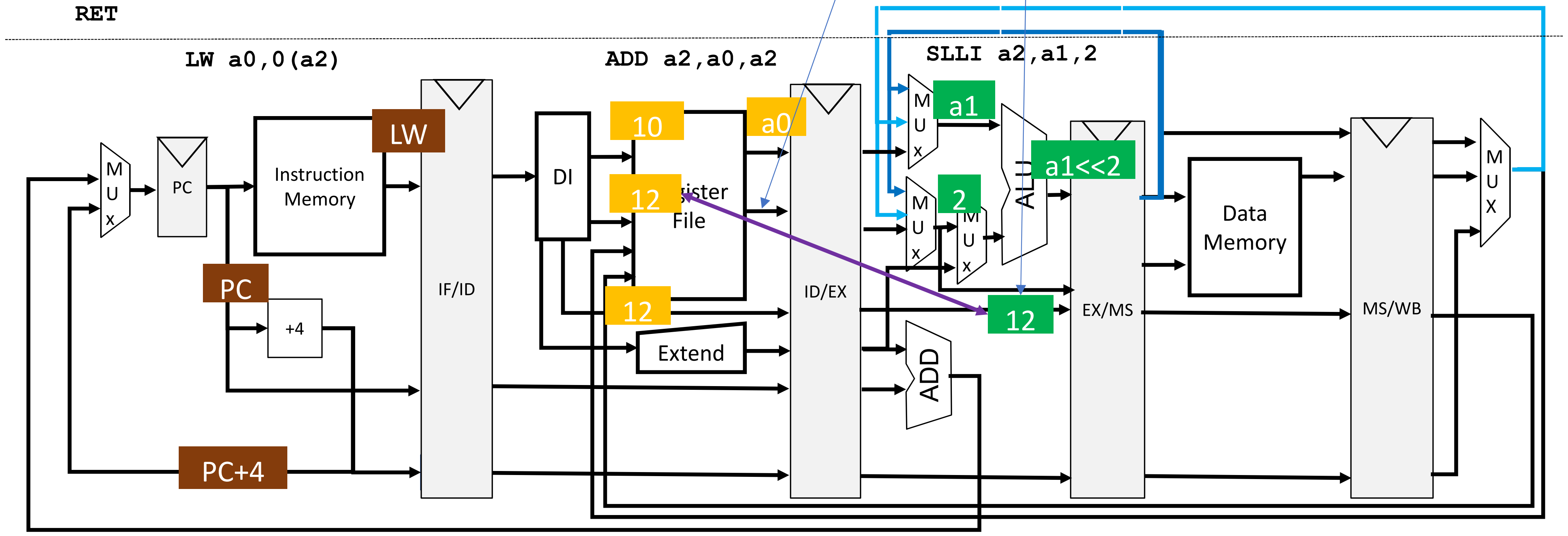
	Cycle 1	Cycle 2
`SLLI a2,a1,2`	IF	ID
`ADD a2,a0,a2`		IF
`LW a0,0(a2)`		
`ADD a0,a0,a1`		
`RET`		



Five-stage Pipeline with Forwarding Path – Example Program – Cycle 3

	Cycle 1	Cycle 2	Cycle 3
`SLLI a2,a1,2`	IF	ID	EX
`ADD a2,a0,a2`		IF	ID
`LW a0,0(a2)`			IF
`ADD a0,a0,a1`			
`RET`			

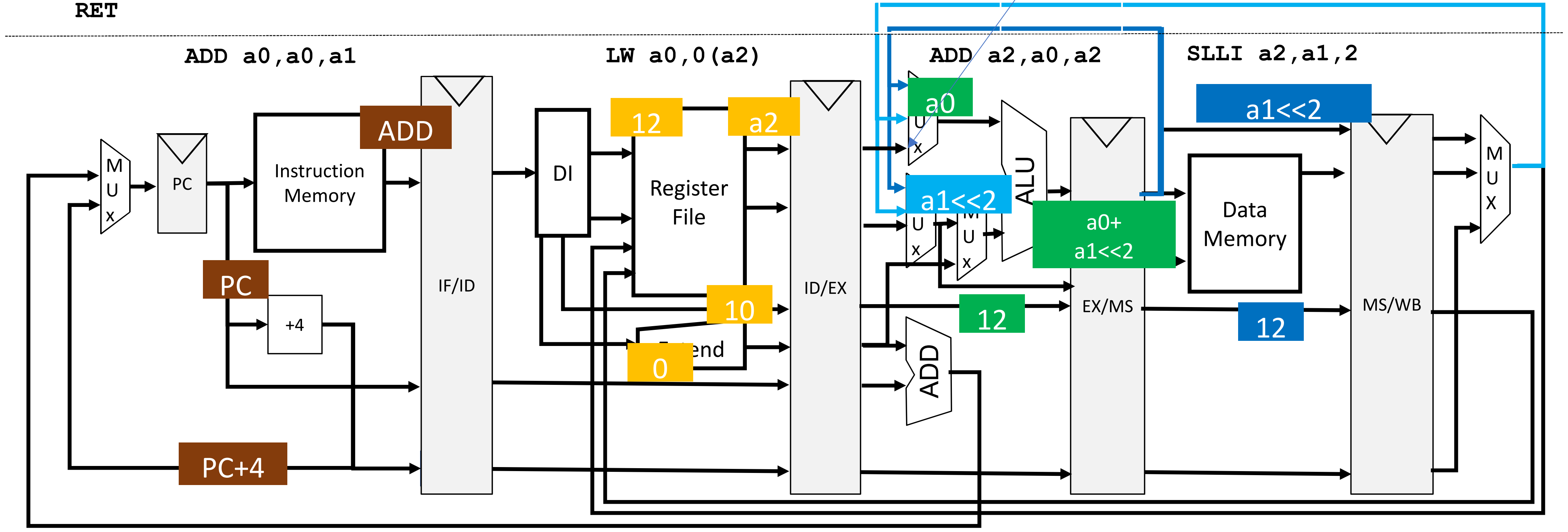
We proceed as we know we can get a2 via the forwarding path



Five-stage Pipeline with Forwarding Path – Example Program – Cycle 4

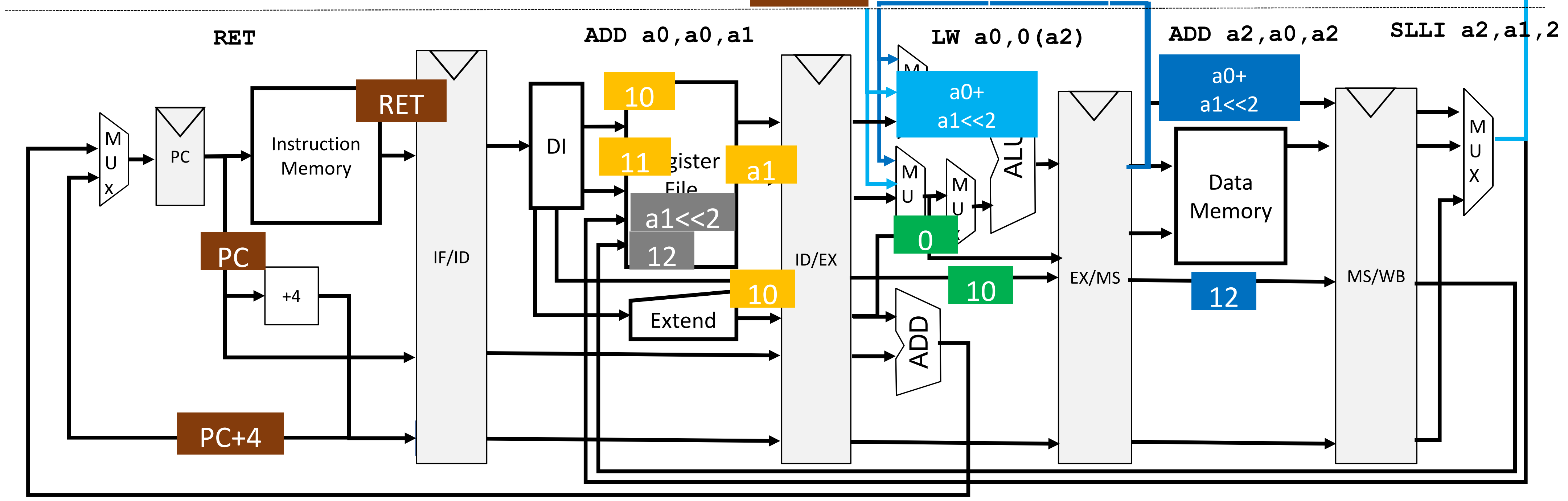
	Cycle 1	Cycle 2	Cycle 3	Cycle 4
`SLLI a2,a1,2`	IF	ID	EX	MS
`ADD a2,a0,a2`		IF	ID	EX
`LW a0,0(a2)`			IF	ID
`ADD a0,a1,a0`				IF
`RET`				

We forward a2 via the forwarding path



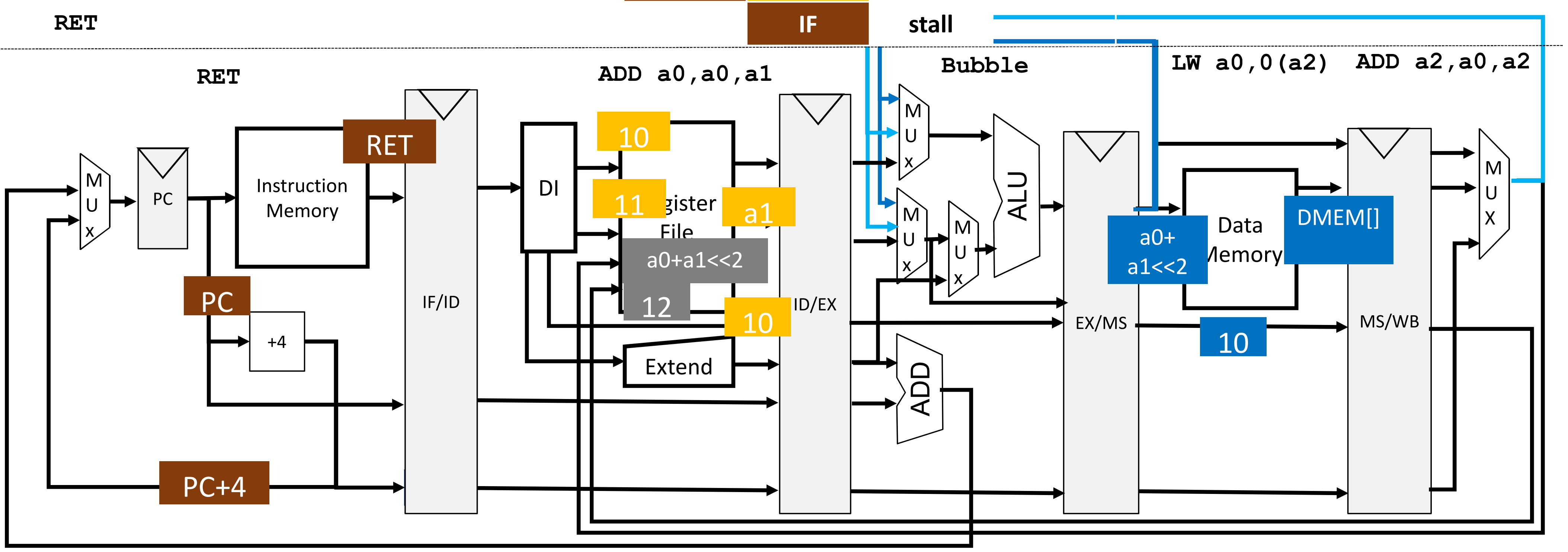
Five-stage Pipeline with Forwarding Path – Example Program – Cycle 5

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
`SLLI a2,a1,2`	IF	ID	EX	MS	WB
`ADD a2,a0,a2`		IF	ID	EX	MS
`LW a0,0(a2)`			IF	ID	EX
`ADD a0,a0,a1`				IF	ID
`RET`					IF



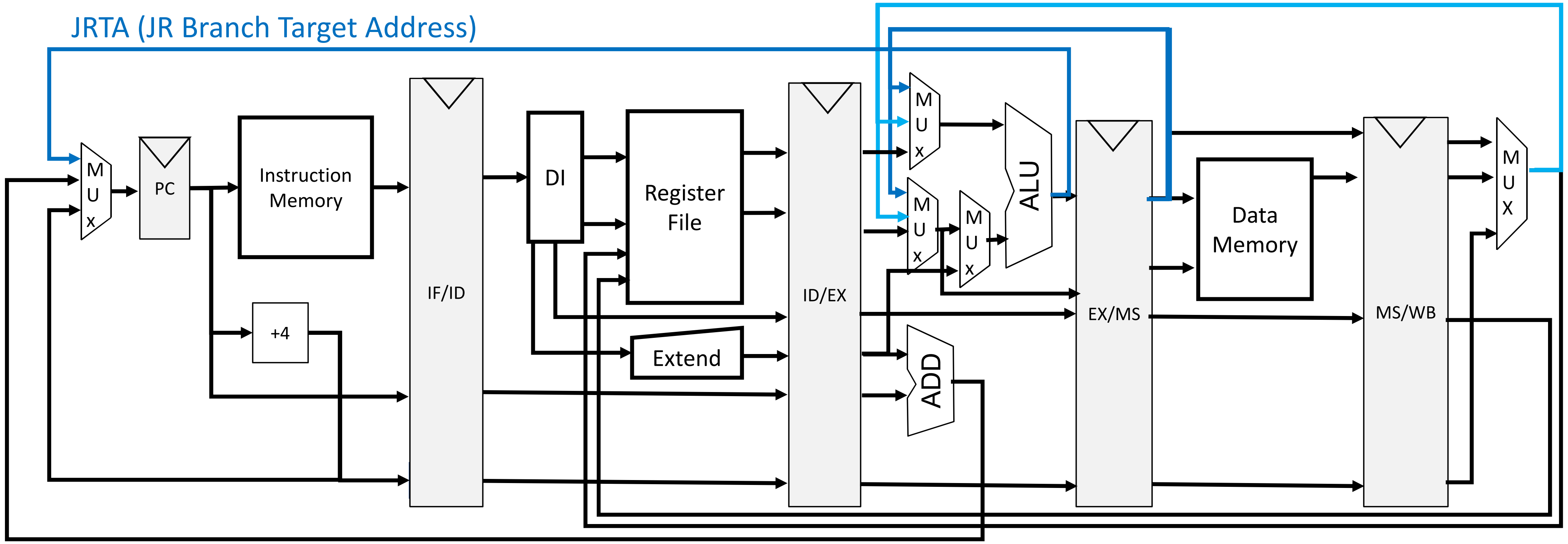
Five-stage Pipeline with Forwarding Path – Example Program – Cycle 6

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6
`SLLI a2,a1,2`	IF	ID	EX	MS	WB	
`ADD a2,a0,a2`		IF	ID	EX	MS	WB
`LW a0,0(a2)`			IF	ID	EX	MS
`ADD a0,a0,a1`				IF	ID	stall
`RET`					IF	stall



Five-stage Pipeline with Forwarding Path and JR

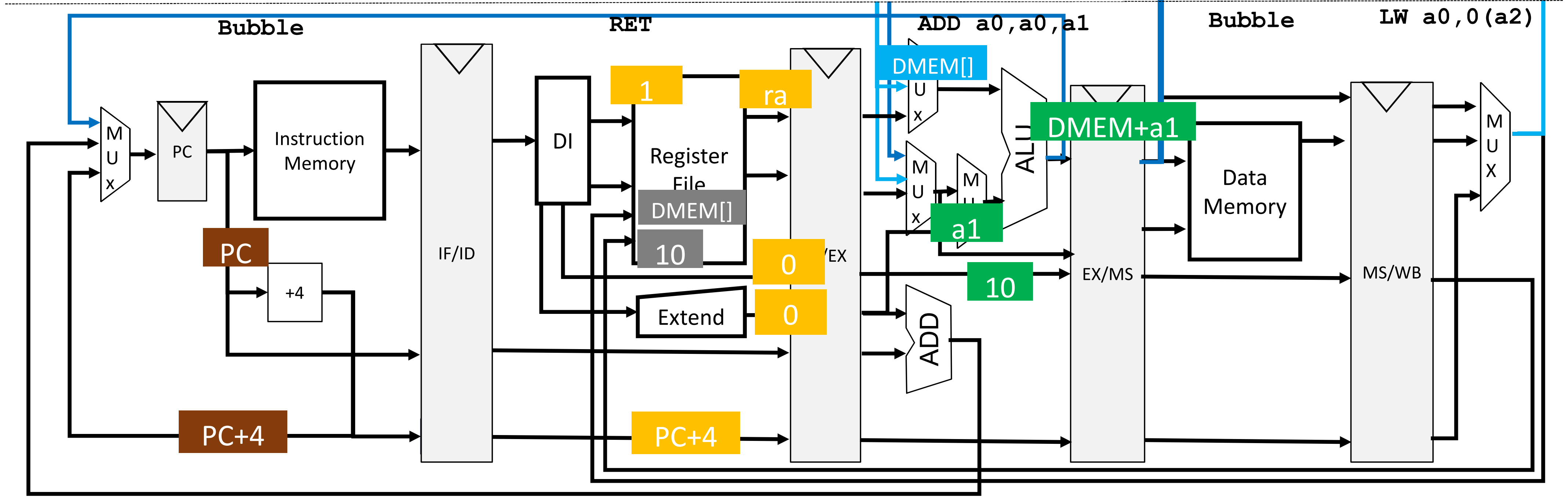
- RET is a pseudo-instruction for jump register JR ra, which is a pseudo instruction for JALR x0,ra,0
- The Harris pipeline does not support to load a register value into PC
- We need another bus for implementing the JR instruction



Five-stage Pipeline with Forwarding Path – Example Program – Cycle 7

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7
`SLLI a2,a1,2`	IF	ID	EX	MS	WB		
`ADD a2,a0,a2`		IF	ID	EX	MS	WB	
`LW a0,0(a2)`			IF	ID	EX	MS	WB
`ADD a0,a0,a1`				IF	ID	stall	EX
`RET`					IF	stall	ID

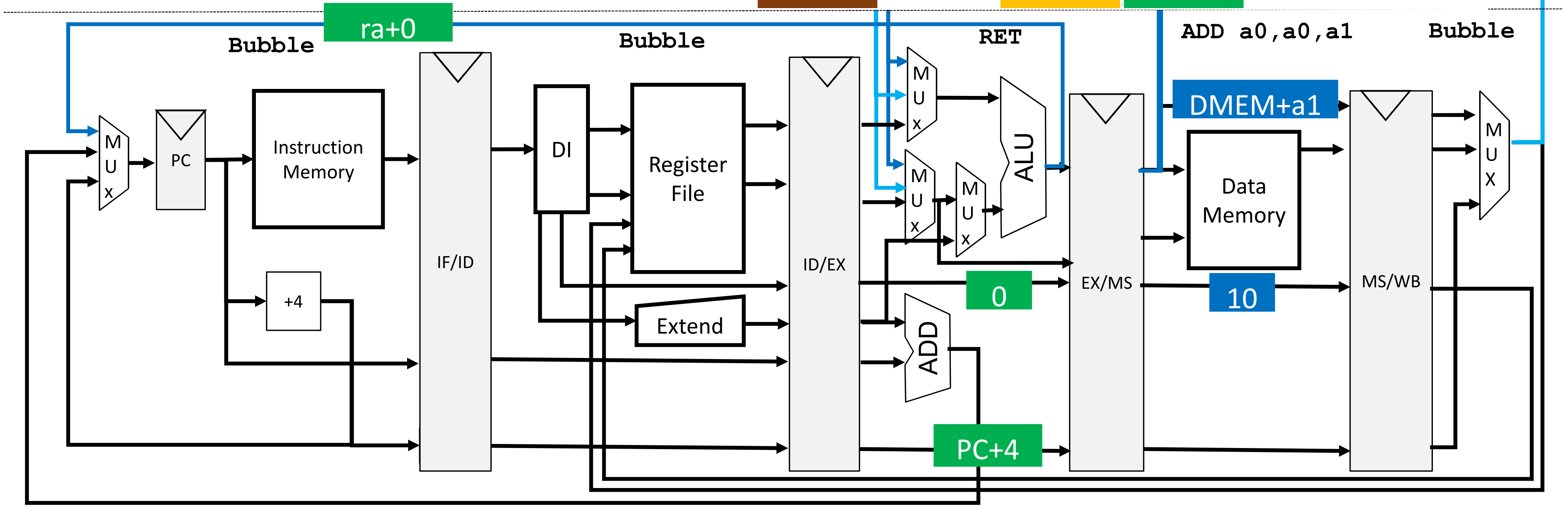
RET Pseudo instr for JALR x0,ra,0



Five-stage Pipeline with Forwarding Path and JR - Example Program – Cycle 8

RET Pseudo instr for JALR x0,ra,0

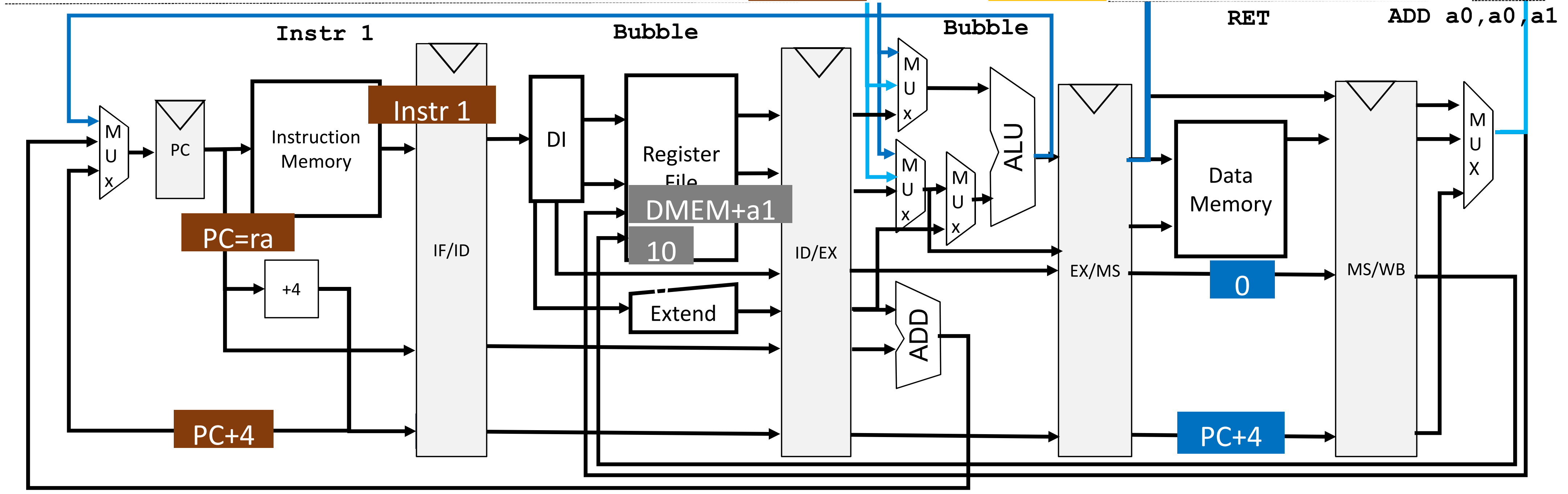
	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
`SLLI a2,a1,2`	IF	ID	EX	MS	WB			
`ADD a2,a0,a2`		IF	ID	EX	MS	WB		
`LW a0,0(a2)`			IF	ID	EX	MS	WB	
`ADD a0,a0,a1`				IF	ID	stall	EX	MS
`RET`					IF	stall	ID	EX



Five-stage Pipeline with Forwarding Path and JR - Example Program – Cycle 9

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
`SLLI a2,a1,2`	IF	ID	EX	MS	WB				
`ADD a2,a0,a2`		IF	ID	EX	MS	WB			
`LW a0,0(a2)`			IF	ID	EX	MS	WB		
`ADD a0,a0,a1`				IF	ID	stall	EX	MS	WB
`RET`					IF	stall	ID	EX	MS

RET Pseudo instr for JALR x0,ra,0

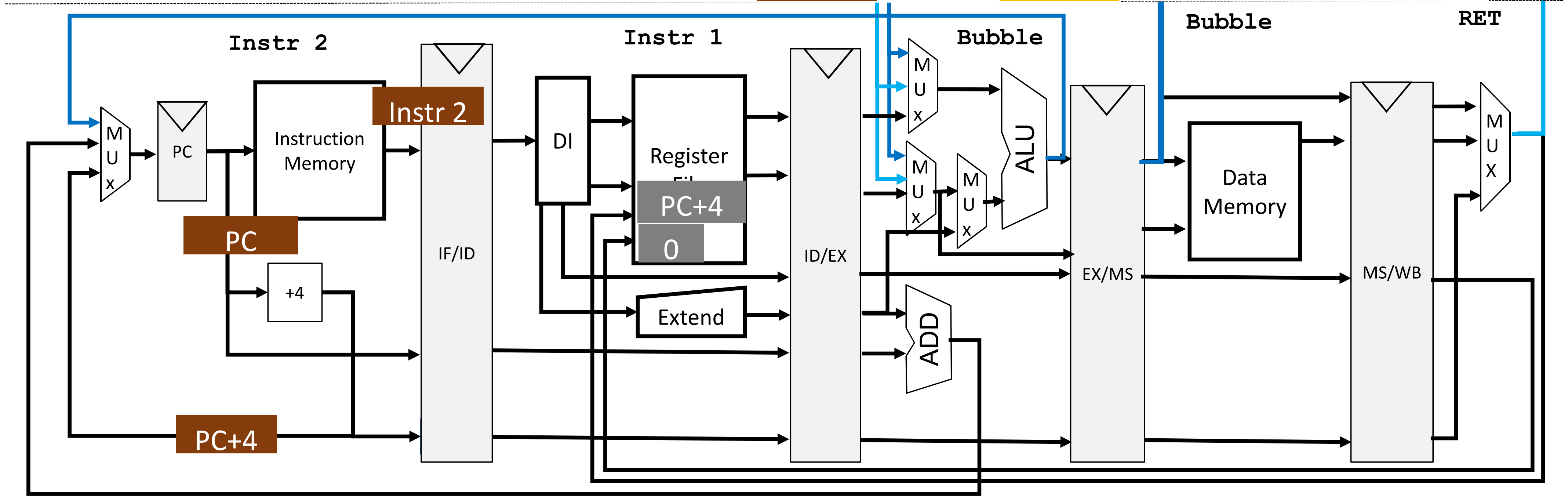


Five-stage Pipeline with Forwarding Path and JR - Example Program – Cycle 10

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
`SLLI a2,a1,2`	IF	ID	EX	MS	WB					
`ADD a2,a0,a2`		IF	ID	EX	MS	WB				
`LW a0,0(a2)`			IF	ID	EX	MS	WB			
`ADD a0,a0,a1`				IF	ID	stall	EX	MS	WB	
`RET`					IF	stall	ID	EX	MS	WB

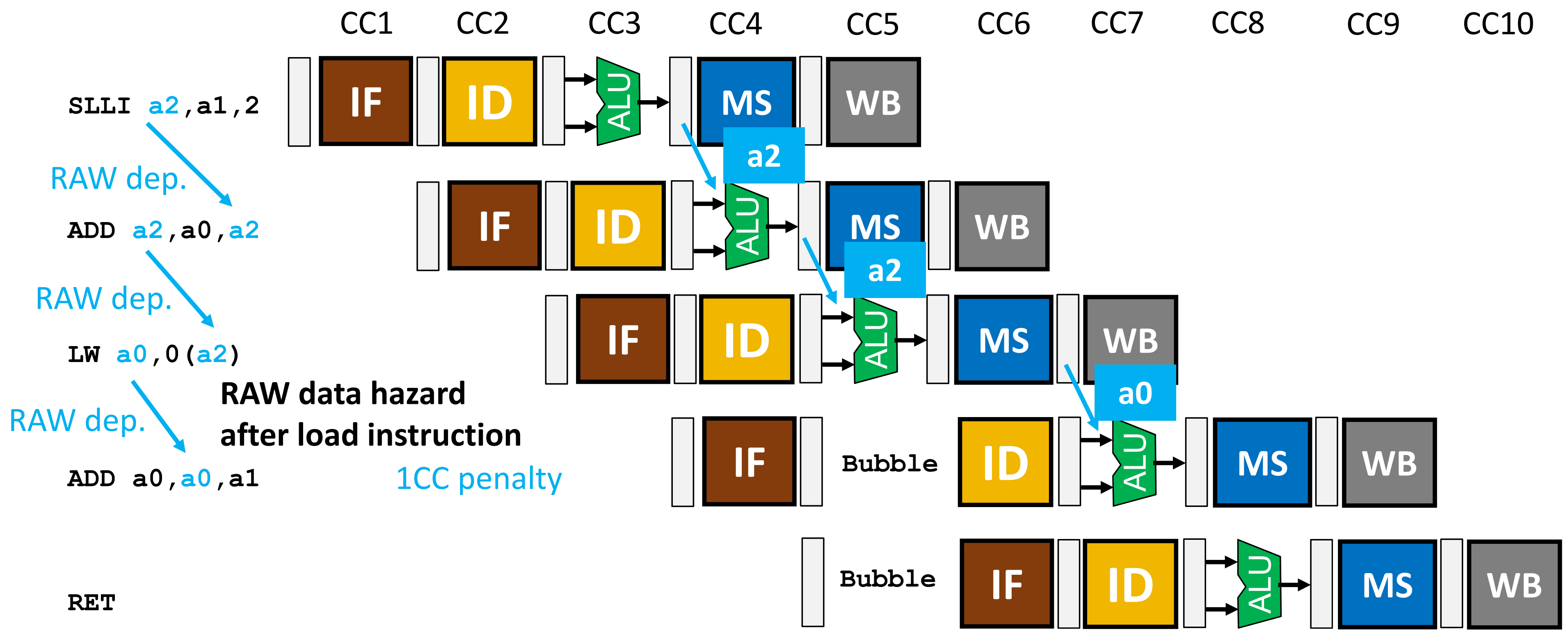
RET Pseudo instr for JALR x0,ra,0

WB on x0 has no effect, always x0=0



Five-stage Pipeline with Forwarding Path and JR - Pipeline Execution Diagram

- With forwarding path: Possible data hazard after load with penalty of 1 clock cycle (cc)



Read After Write (RAW) dependency or „**true dependency**“:
One instructions reads operand that is written as result of previous instructions.
Data hazard prevents the next instruction in the instruction stream from executing during its designated clock cycle.

Compiler Instruction Scheduling to Avoid RAW Data Hazards after Load Instructions

- Compiler often can move instructions to avoid RAW data hazards after loads
- Program order must not change (See next session)

➢ Rarely data hazard penalty observed in five-stage pipeline with forwarding paths

➢ Example:

```
vec_add_for:
  LW t1,0(a0)      # t1 = a[i]
  LW t2,0(a1)      # t2 = b[i]
   RAW        1CC penalty
  ADD t1,t1,t2     # t1 = a[i] + b[i]
  SW t1,0(a2)      # c[i] = t1
  ADDI a0,a0,4     #base address + 4
  ADDI a1,a1,4     #base address + 4
  ADDI a2,a2,4     #base address + 4
  ADDI t0,t0,1     # i++
  (…)
```

```
vec_add_for:
  LW t1,0(a0)      # t1 = a[i]
  LW t2,0(a1)      # t2 = b[i]
  ADDI t0,t0,1     # i++
  ADD t1,t1,t2     # t1 = a[i] + b[i]
  SW t1,0(a2)      # c[i] = t1
  ADDI a0,a0,4     #base address + 4
  ADDI a1,a1,4     #base address + 4
  ADDI a2,a2,4     #base address + 4
    (…)
```

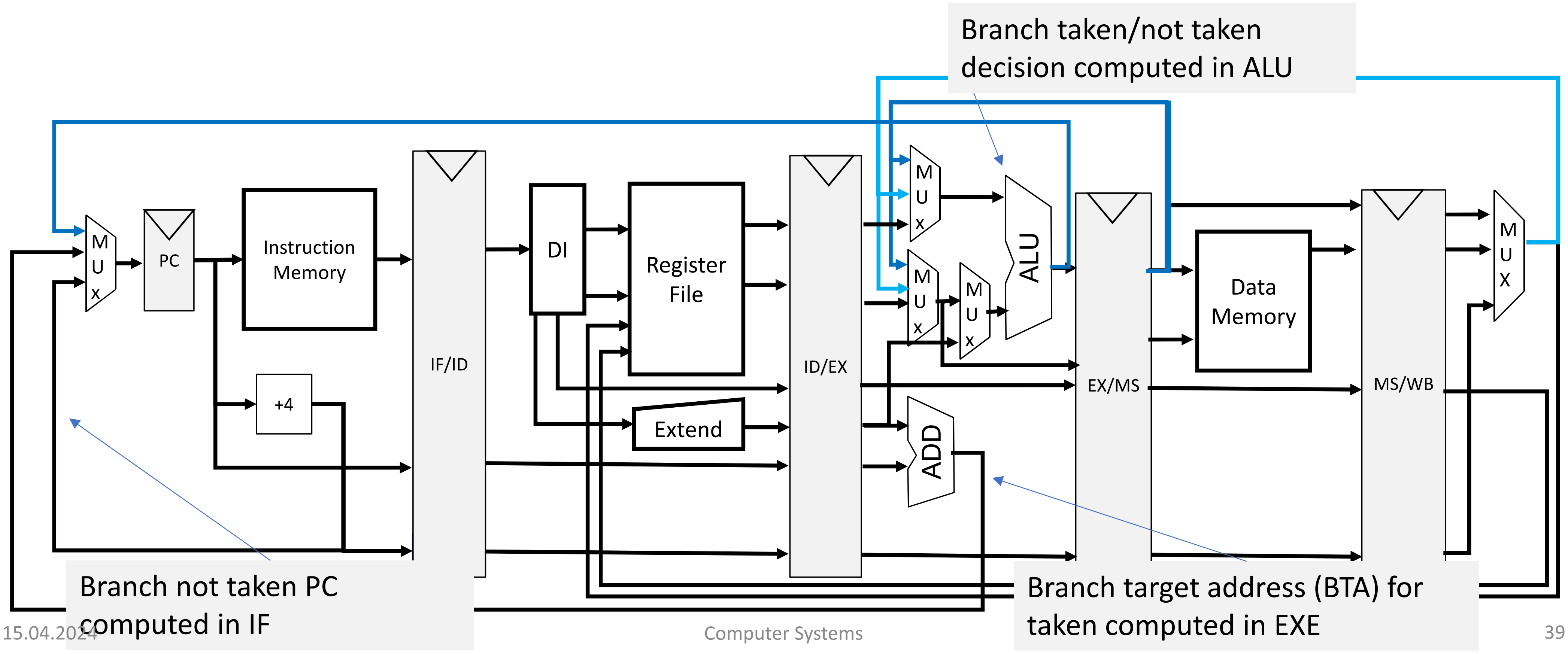
Control Hazard

Control Hazards

- **Control hazards** arise from instructions that change the PC
- **When the flow of instruction addresses is not sequential**
  - Unconditional branches (`jal, jalr`)
  - Conditional branches (`beq, bne, ...`)
  - Exceptions
- Possible approaches
  - **Stall** (impacts CPI)
  - **Move decision point** as early in the pipeline as possible (Extra HW)
  - **Predict** and hope for the best!
  - ***Delay decision*** *(requires compiler support)*
- Control hazards occur **less frequently** than data hazards,
  but there is **nothing as effective** against control hazards as forwarding is for data hazards

Control Hazards – Conditional Branches

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome (Branch taken/Not taken)
  - Next PC is either PC+4 (branch not taken) or PC+imm<<1 (branch taken)


Branch taken/not taken decision computed in ALU
MUX
PC
Instruction Memory
+4
IF/ID
DI
Register File
Extend
ID/EX
MUX
MUX
MUX
ALU
ADD
EX/MS
Data Memory
MS/WB
MUX
Branch not taken PC computed in IF
Branch target address (BTA) for taken computed in EXE

Handling Control Hazards: Stall on Branch

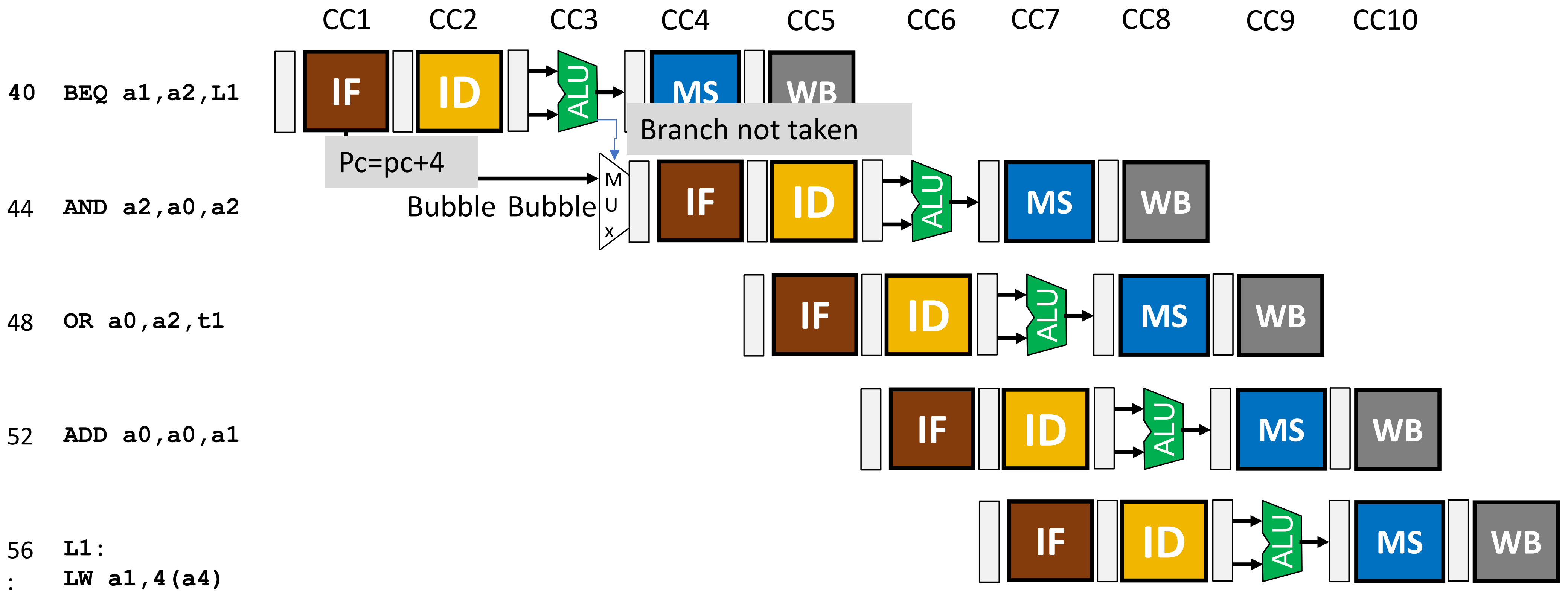
- **Conservative Approach:** **Wait** until branch outcome determined before fetching next instruction

Conservative approach: Stall immediately after fetching a branch, wait until outcome of branch is known and fetch branch address.

- **Reducing Branch Delay:**
  - **E.g. Move Branch Decision to ID Stage:** **Extra hardware** so that we can test registers, calculate the branch address, and update the PC *during the second stage of the pipeline*

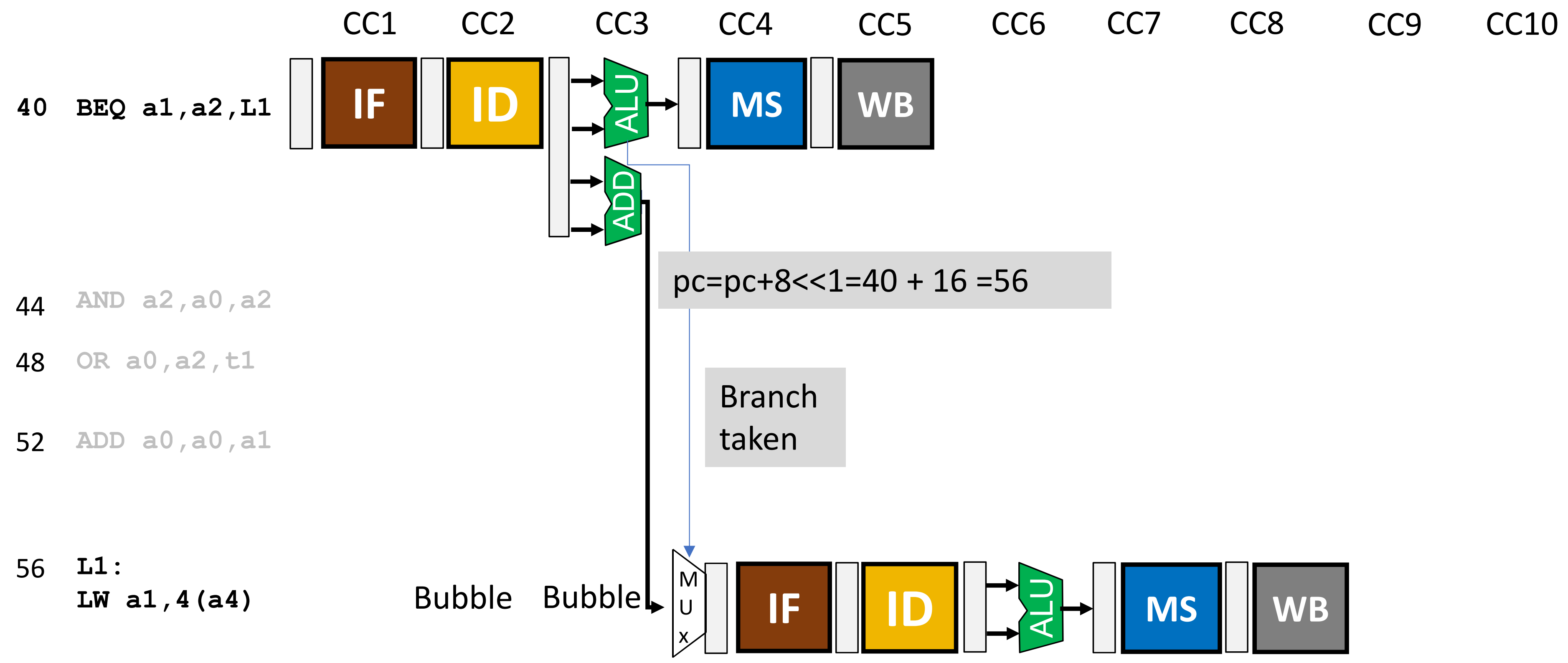
Handling Control Hazards: Conservative Approach (Branch not Taken)

- Control hazard (branch not taken): stall pipeline until decision known
- Branch penalty: 2 clock cycles



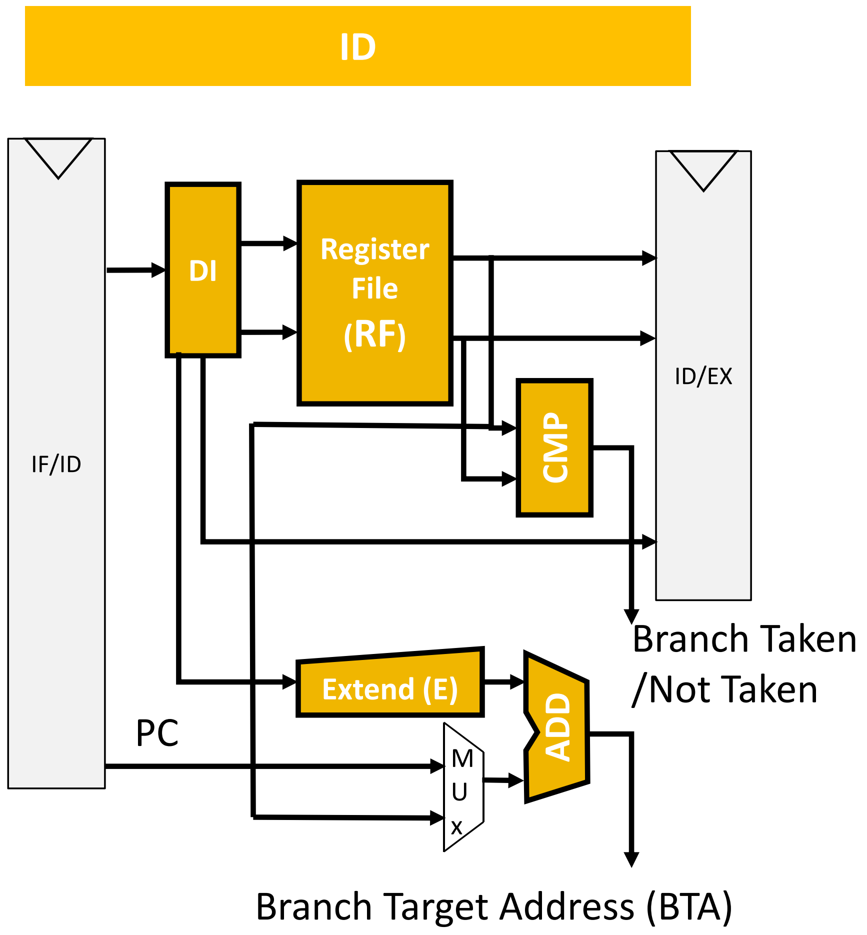
Handling Control Hazards: Conservative Approach (Branch Taken)

- Control hazard (branch taken): stall pipeline until decision and branch target known
- Branch penalty: 2 clock cycles



Reducing Branch Delay - Move Branch Decision to ID Stage

- **A lot of branches rely on simple tests (e.g., equality)**
- **Add hardware to determine outcome of branch in the ID stage**
  → Reduce cost of the taken branch
  - Subcomputation: Compute Branch Target Address in ID
    - Move target address adder from EX to ID
    - PC and immediate are already in IF/ID pipeline register
  - Subcomputation: Comparison
    - Additional register comparator (done before in EX via the ALU)
    - **Additional Forwarding and Hazard Handling**



Reducing Branch Delay - Move Branch Decision to ID Stage – Branch Taken

- Target address adder in ID, Extra comparator to get branch decision in ID
- Branch penalty: Only one clock cycle

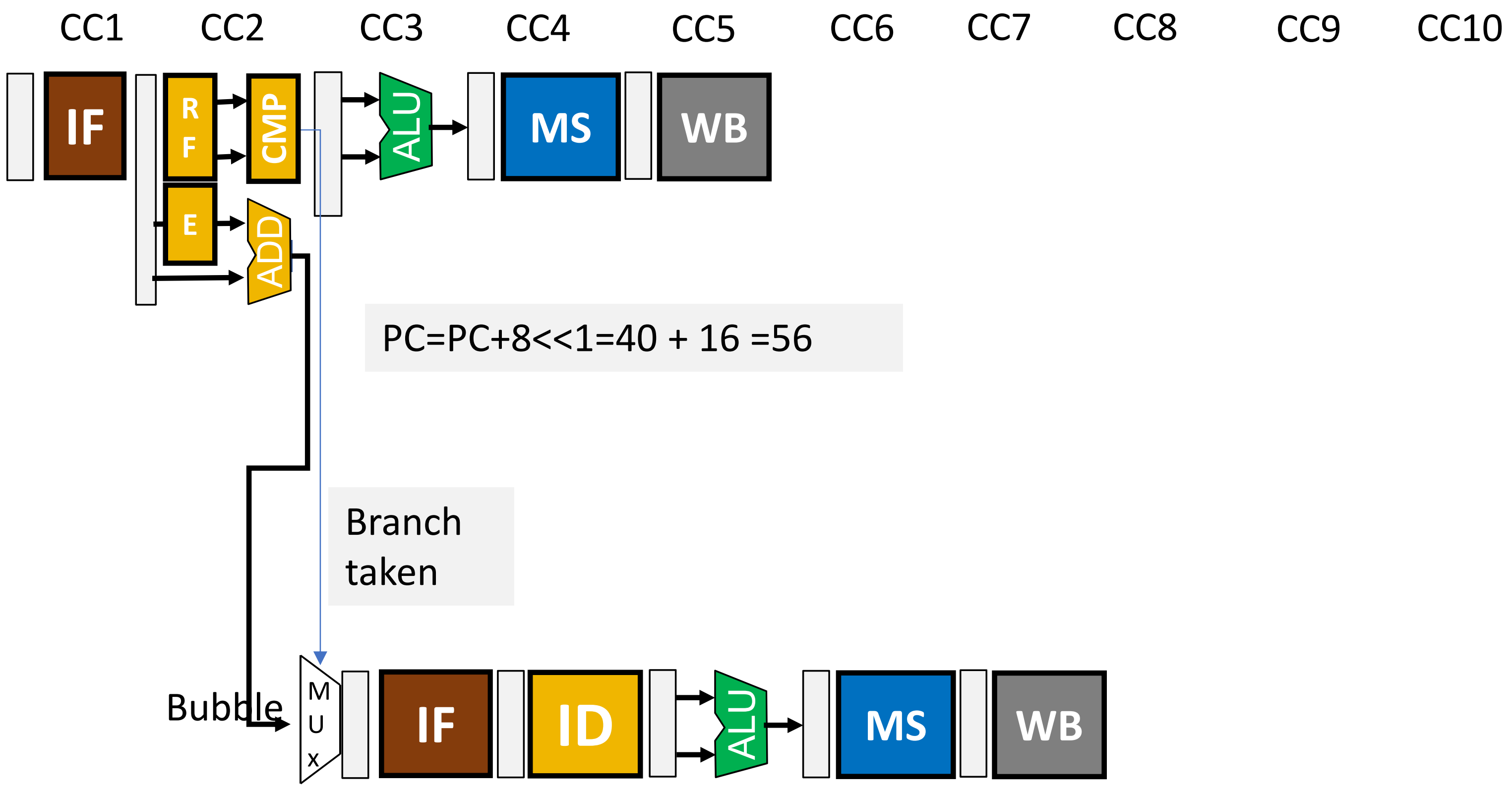
```
40   BEQ a1,a2,L1

44   AND a2,a0,a2

48   OR a0,a2,t1

52   ADD a0,a0,a1

56   L1:
     LW a1,4(a4)
```



Static Branch Prediction

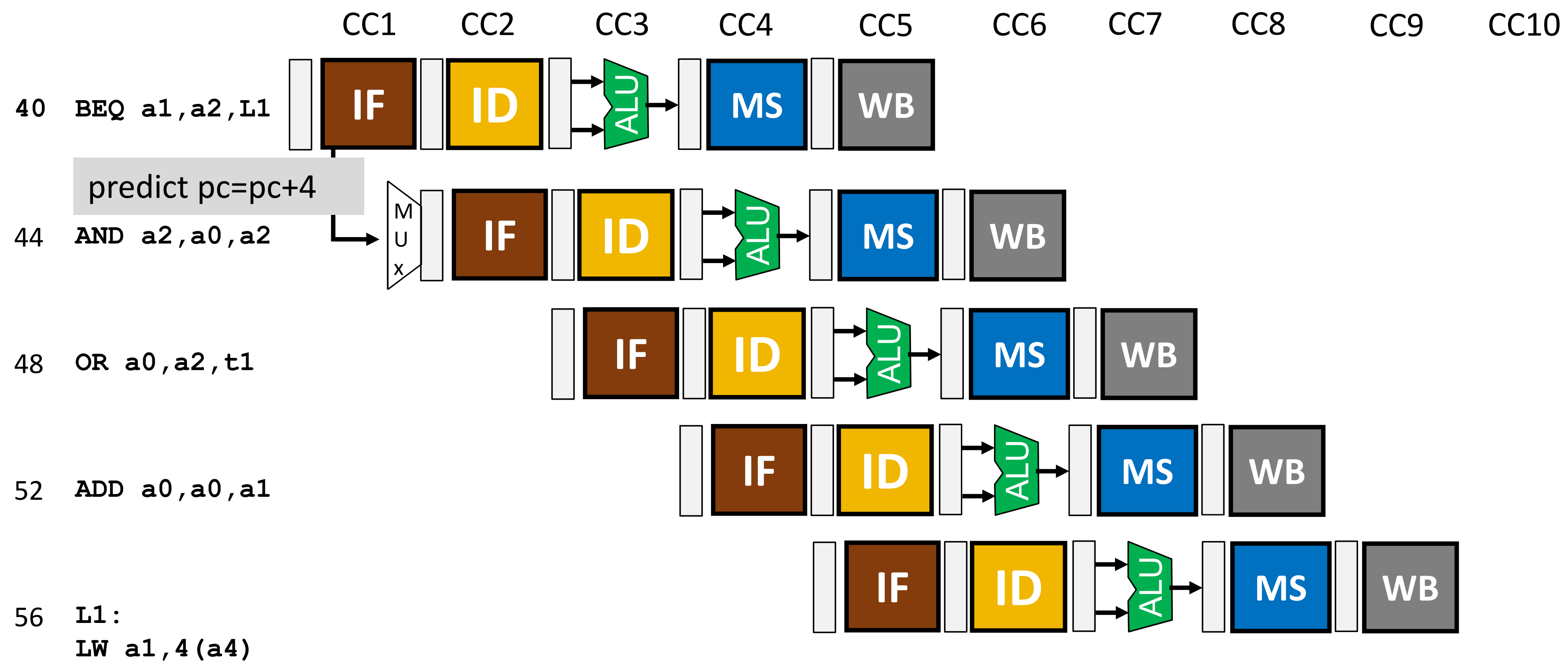
Motivation: Branch Prediction

- Longer pipelines can’t readily determine branch outcome early
  - Branch penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong

- Simple Static Branch Prediction Schemes
  - ➢ **Always not Taken**: Always predict branches not taken – Also called fall through (PC=PC+4)
  - ➢ **Always taken**: Always predict branches taken

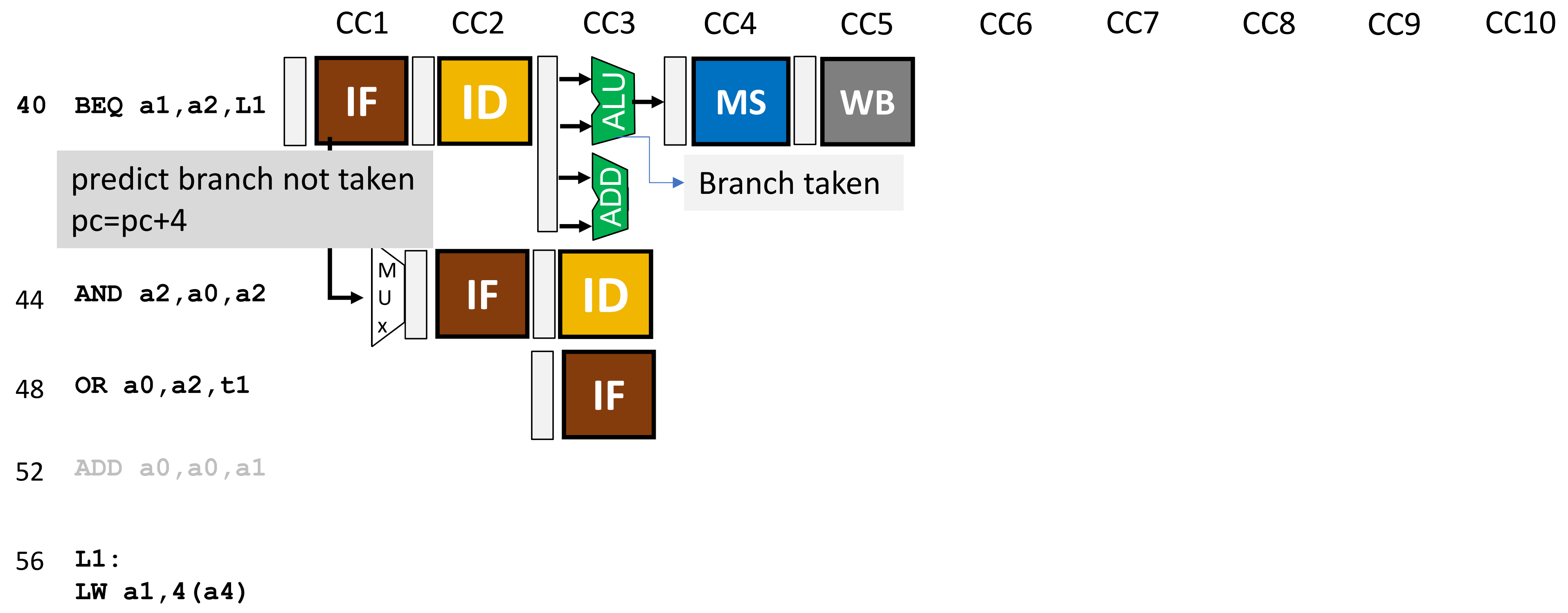
Always Not Taken – Correct Prediction

- Prediction correct (Branch not taken)
- Branch penalty: 0 clock cycles



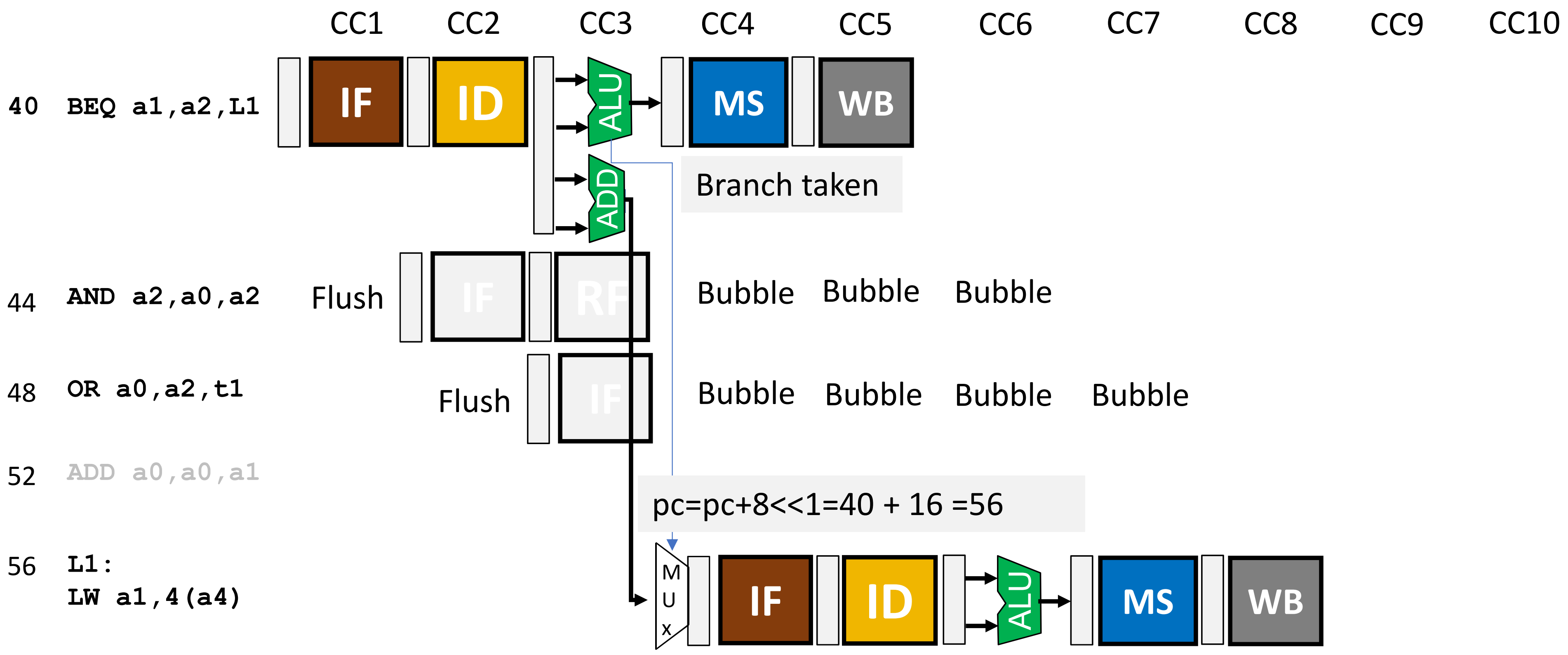
Always not Taken – Incorrect Prediction (1/2)

- Prediction incorrect (Branch not taken) – Flush instructions from pipeline
- Branch penalty: 2 clock cycles



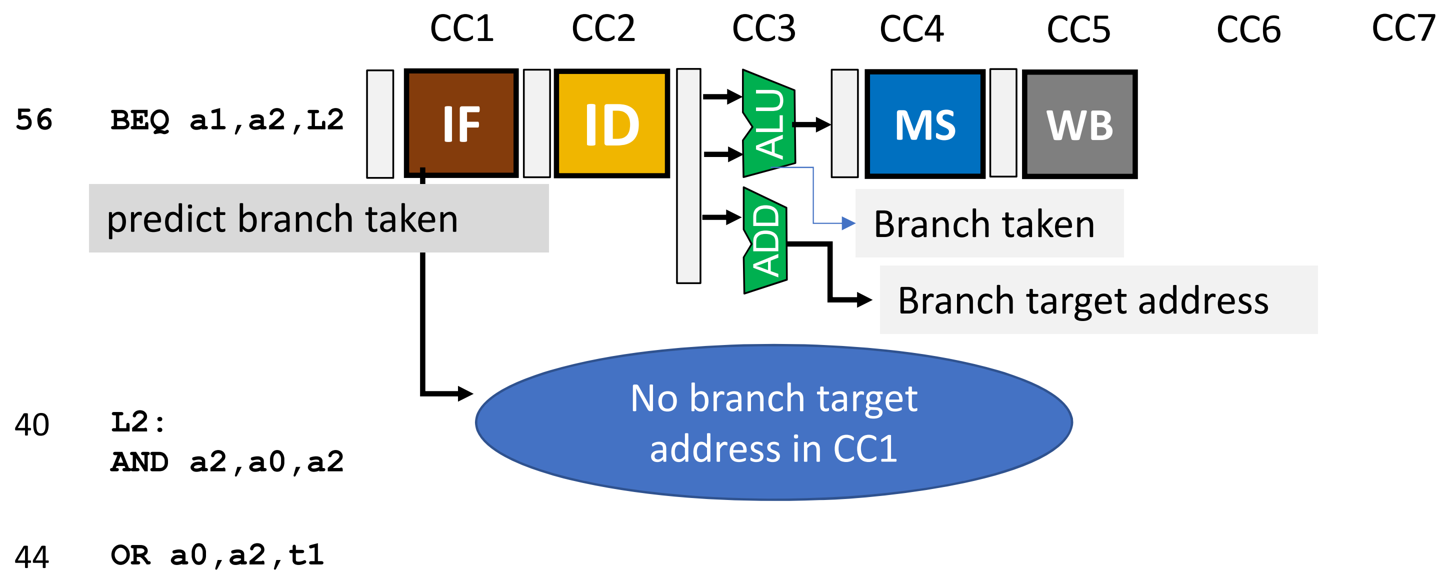
Always not Taken – Incorrect Prediction (2/2)

- Prediction incorrect (Branch not taken) – Flush instructions from pipeline
- Branch penalty: 2 clock cycles



Always Taken – Correct Prediction

- Prediction (Branch taken) –> Branch target address is computed in EX stage



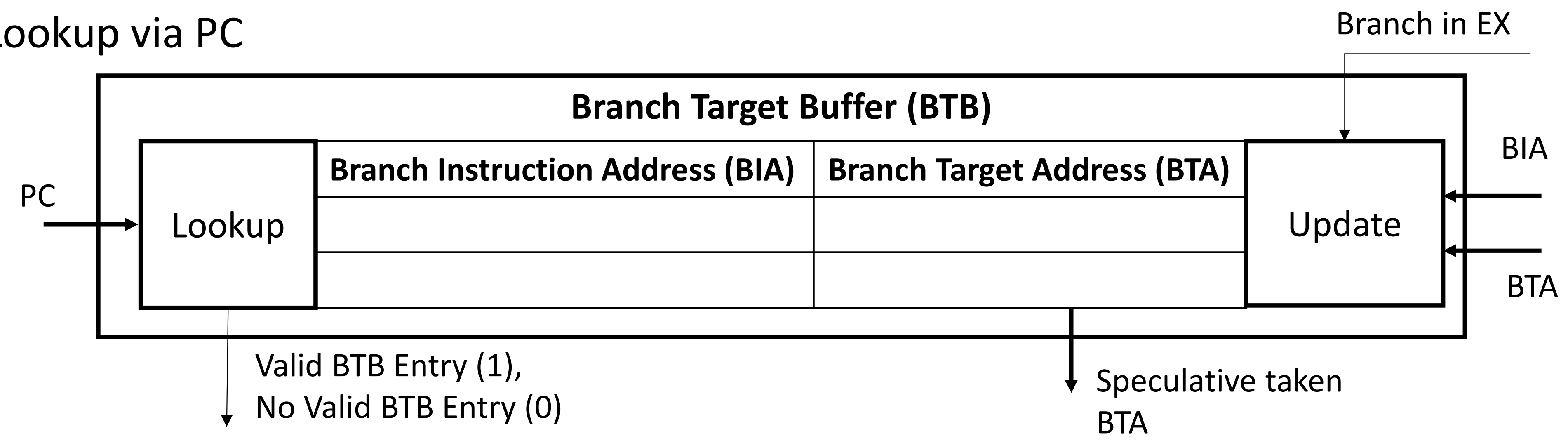
```
56    BEQ a1,a2,L2

40    L2:
      AND a2,a0,a2

44    OR a0,a2,t1
```

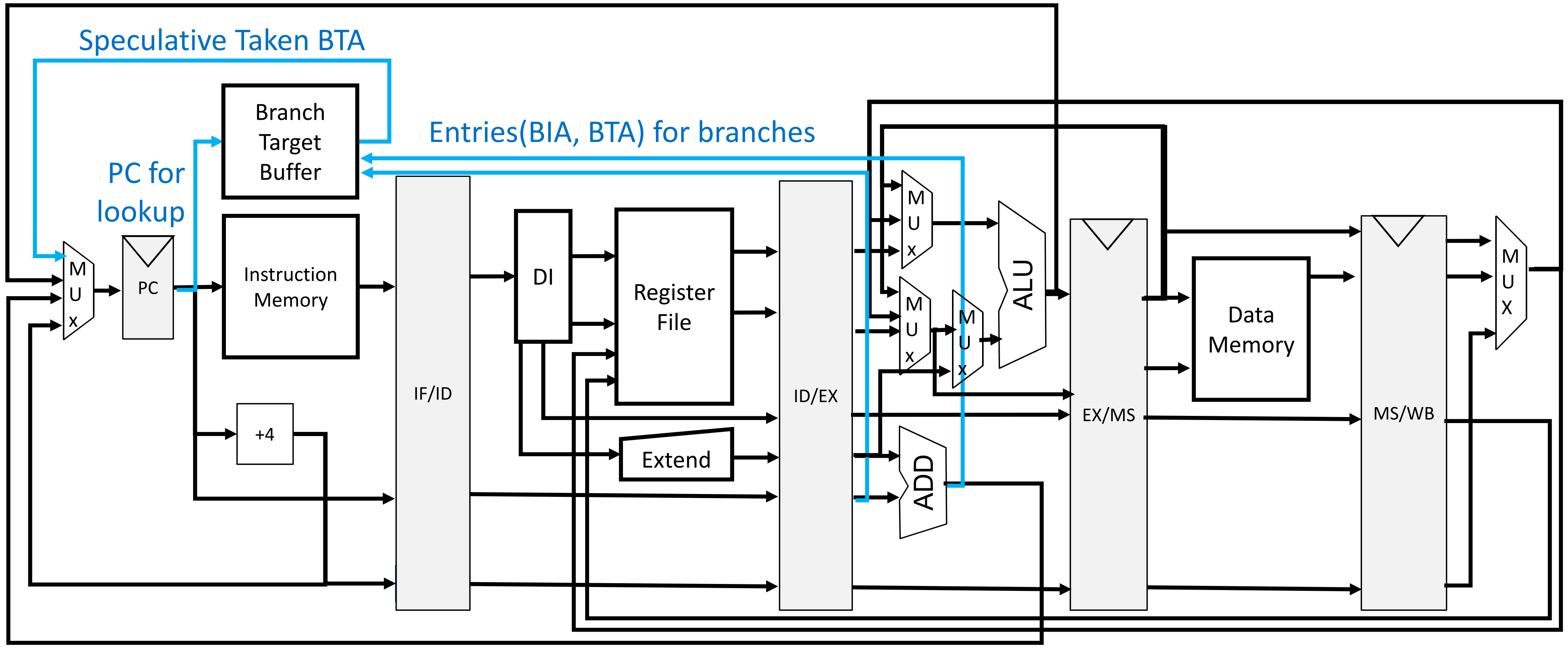
Branch Target Buffer (BTB)

- Stores the Branch Target Address (BTA) for a certain branch (e.g. identified by its own Branch Instruction Address (BTI))
- Content Addressable Memory (Costly for entries)
  - Update policy (similar to caches)
  - Entries entered in pairs (BIA, BTA)
  - entry not available for first branch execution
- Lookup via PC



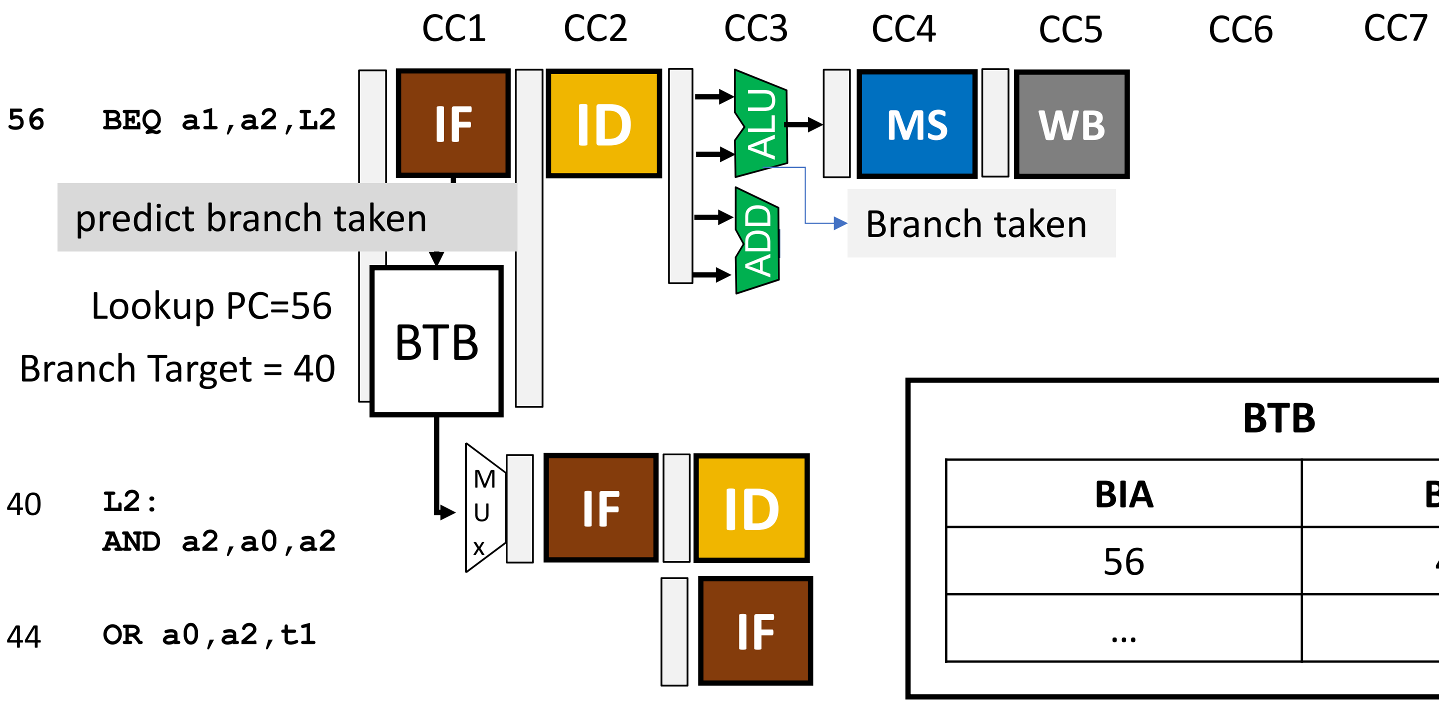
Five-stage Pipeline with Branch Target Buffer

- Only for branches and PC-relative Jumps J, JAL (not JALR, JR, RET)



Always Taken – Correct Prediction (BTB has entry)

- Prediction (Branch taken) - BTA via Branch Target Buffer (BTB)
- Branch penalty: 0 clock cycles



BTB

BIA	BTA
56	40
…	…

First execution of branch we cannot do a branch taken prediction.
Entry was written to BTB on earlier execution of branch with (56,40)

BTFNT: Enhancing Static Branch Prediction

Typical Statistics 60% to 70% of branches are taken

Example:

- 60% are backward branches (negative offset)
  - Loops: Usual more than one iteration (branch will be taken more than once) – taken ~90%
  - Typical behavior: T T T T…T NT
  - About 90% of backward branches are taken

- 40% are forward branches (positive offset)
  - If-(Else) Constructs: Branches go forward (jump over code)
  - About ~20% of forward branches are taken

- Always not taken: (0,6 · 0,9) + (0,4 · 0,2) = 62% mispredictions
- Always taken: (0,6 · 0,1) + (0,4 · 0,8) = 38% mispredictions
- **Enhanced Static Branch Prediction: Backward Taken, Forward Not Taken (BTFNT)**
  - Predict **forward** branches **not taken: ~10% mispredictions**
  - Predict **backward** branches **taken: ~20% mispredictions**
  - **Overall:** (0,6 · 0,1) + (0,4 · 0,2) = 14% mispredictions

Effect of Misprediction Rate and Branch Penalty on CPI

Program with:

- Relative number of branch instructions (branch rate ***b***)
- The branch cycle penalty ***p*** for mispredictions
- The branch misprediction rate ***m***

- **CPI**: Cycles per Instructions (data hazards rare so base CPI=1)

$$\boxed{\textbf{CPI}= \mathbf{1} + \boldsymbol{b} \cdot \boldsymbol{p} \cdot \boldsymbol{m}}$$

Five stage pipeline: ***b***=15%, ***p***=2
Always not taken: ***m***=62% -> **CPI** = 1,186
Always taken: ***m***=38% -> **CPI** = 1,114

BTFNT: ***m***= 14% -> **CPI** = 1,042

Longer pipeline: ***b***=15%, ***p***=5
Always not taken: ***m***=62% -> **CPI** = 1,465
Always taken: ***m***=38% -> **CPI** = 1,285

BTFNT: ***m***= 14% -> **CPI** = 1,105

In **longer** pipelines, branch penalty is more significant

Dynamic Branch Prediction

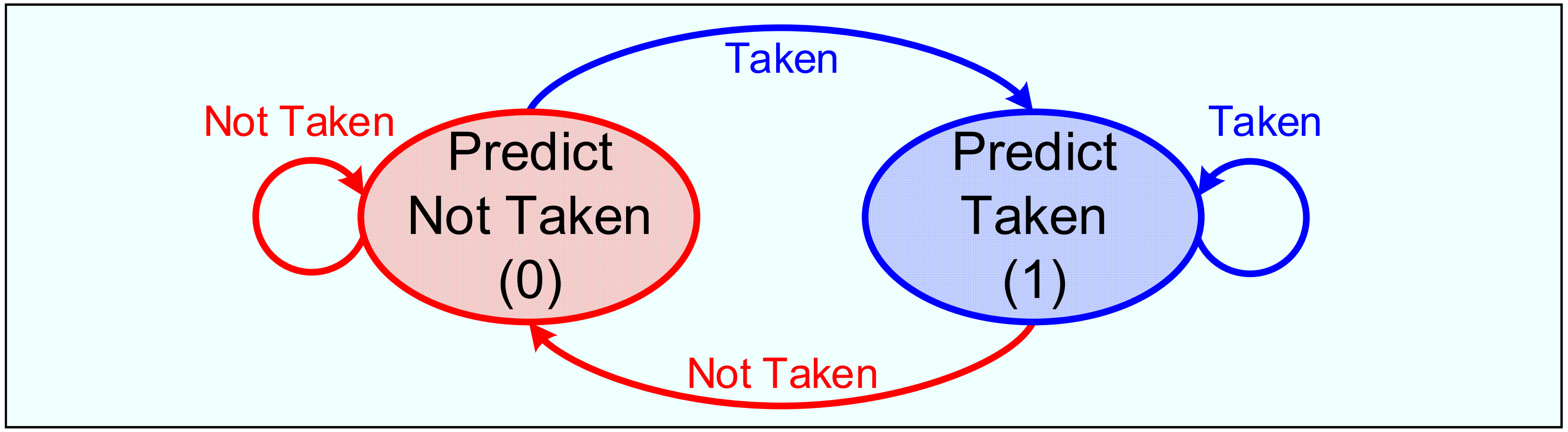
Dynamic Branch Prediction

- In **longer** pipelines, branch penalty is more significant

- *Branch prediction buffer* (aka branch history table (BHT)) for dynamic prediction
  - Stores last outcome (taken/not taken)
  - To execute a branch
    - ➢ 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

Dynamic Branch Prediction: 1-Bit Predictor

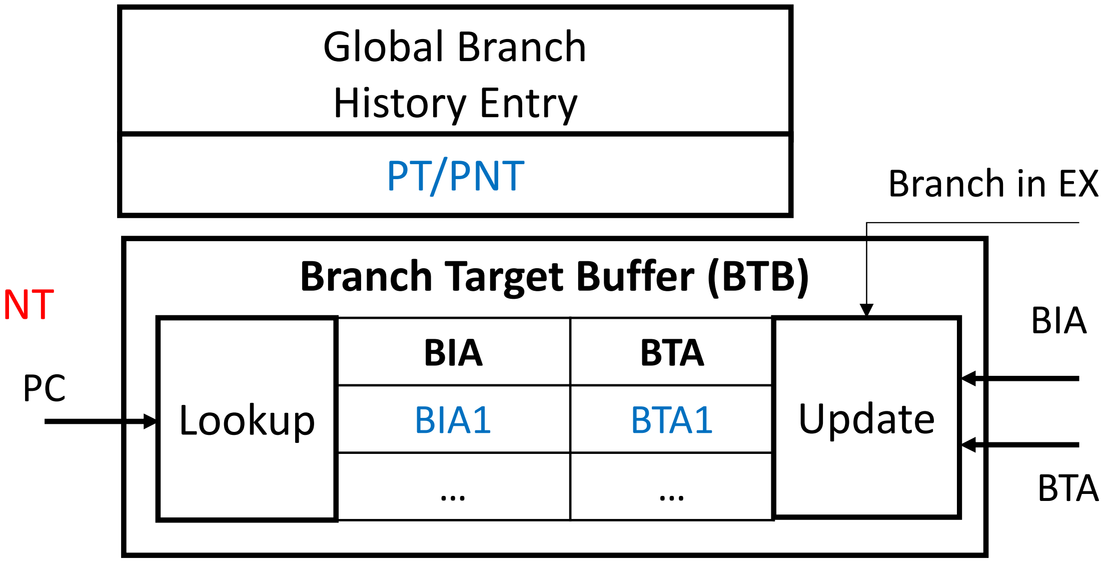
- **Single-Bit / 1-Bit / Last-Time Predictor**
  - Indicates *which direction the branch went last time it executed*
  - PNT: Predict NT (Bit=0): Fetch the instruction from (PC+4)
  - PT: Predict T (Bit=1): Get target address from the BTB



Global Predictor

- One single Branch History Entry for all branches to save last decision

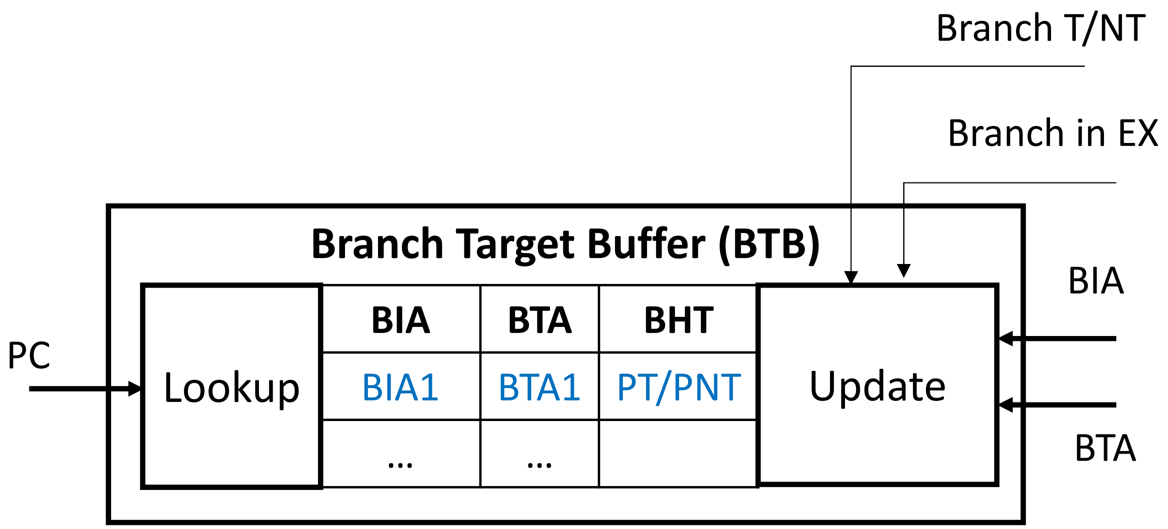
- Branch reaches IF stage
  - Indexed Lookup with PC in BTB
  - No valid BTB entry
    –> predict NT (PNT)
    -> Supply **PC=PC+4**
  - Valid BTB entry
    -> Global Predictor result based on BHT: PT/PNT
    -> Supply **PC=BTA/PC+4**
- Branch reaches EX stage
  - Indexed Lookup with PC in BTB
    -> No BTB entry -> Update BTB (create entry)
    -> Eventually only in case that branch is taken
  - Update Global Branch History Entry



Local Predictor

- Branch History Table (BHT): One entry for each BTB entry

- Branch in IF stage
  - Indexed Lookup with PC in BTB
  - No valid BTB entry
    –> predict NT
    -> Supply **PC=PC+4**
  - Valid BTB entry
    -> Local BHT Predictor result T/NT
    -> Supply **PC=BTA/PC+4**
- Branch in EX stage
  - Indexed Lookup with BIA in BTB
    - No BTB entry -> Update BTB (create entry), initialize BHB with T/NT
    - BTB entry: Update Local BHT with T/NT



Example Nested Loop Program - Static Branch Prediction

- Example Nested Loop Program:

```
for (x = 1024; x > 0; x--)
  for (y = 4; y > 0; y--)
    do_something(x,y);
```

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Inner Loop (L09 Branch Pattern): (T-T-T-NT)
Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

Static Branch Prediction:

- **Always not taken:** ~80% Mispredictions
- **Always taken:** ~20% Mispredictions
- **BTFNT (same as always taken):** ~20% Mispredictions

Example Nested Loop Program - Dynamic Branch Prediction (1bit Global)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  N=No, Y=Yes
  Misprediction rate:

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y												
BTB entry L11	Y												
Global BHT	PNT												
Prediction	NT												
Direction	-												
Correct?	-												

Example Nested Loop Program - Dynamic Branch Prediction (1bit Global)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T....

  N=No, Y=Yes
  Misprediction rate:

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y	Y	Y	Y	Y	Y							
BTB entry L11	Y	Y	Y	Y	Y	Y							
Global BHT	PNT	PNT	PT	PT	PT	PNT							
Prediction	NT	NT	T	T	T	NT							
Direction	-	T	T	T	NT	T							
Correct?	-	*N*	*Y*	*Y*	*N*	*N*							

Example Nested Loop Program - Dynamic Branch Prediction (1bit Global)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  N=No, Y=Yes
  Misprediction rate: ~40% (2 out of five)

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Repeats

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	…	…
BTB entry L11	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	…	…
Global BHT	PNT	PNT	PT	PT	PT	PNT	PT	PT	PT	PT	PNT	…	…
Prediction	NT	NT	T	T	T	NT	T	T	T	T	NT	…	…
Direction	-	T	T	T	NT	T	T	T	T	NT	T	…	…
Correct?	-	*N*	*Y*	*Y*	*N*	*N*	*Y*	*Y*	*Y*	*N*	*N*	…	…

Example Nested Loop Program - Dynamic Branch Prediction (1bit Local)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  Misprediction rate:

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y												
BTB entry L11	Y												
BHT L9	PNT												
BHT L11	PNT												
Prediction	PNT												
Direction	-												
Correct?	-												

Example Nested Loop Program - Dynamic Branch Prediction (1bit Local)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  Misprediction rate: ~40% (2 out of five)

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y	Y	Y	Y	Y	Y							
BTB entry L11	Y	Y	Y	Y	Y	Y							
BHT L9	PNT	PNT	PT	PT	PT	PNT							
BHT L11	PNT	PNT	PNT	PNT	PNT	PNT							
Prediction	PNT	NT	T	T	T	NT							
Direction	-	T	T	T	NT	T							
Correct?	-	*N*	*Y*	*Y*	*N*	*N*							

Example Nested Loop Program - Dynamic Branch Prediction (1bit Local)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  Misprediction rate: ~40% (2 out of five)

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Repeats

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	…	…
BTB entry L11	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	…	…
BHT L9	PNT	PNT	PT	PT	PT	PNT	PNT	PT	PT	PT	PNT	…	…
BHT L11	PNT	PNT	PNT	PNT	PNT	PNT	PT	PT	PT	PT	PT	…	…
Prediction	PNT	NT	T	T	T	NT	NT	T	T	T	T	…	…
Direction	-	T	T	T	NT	T	T	T	T	NT	T	…	…
Correct?	-	*N*	*Y*	*Y*	*N*	*N*	*N*	*Y*	*Y*	*N*	*Y*	…	…

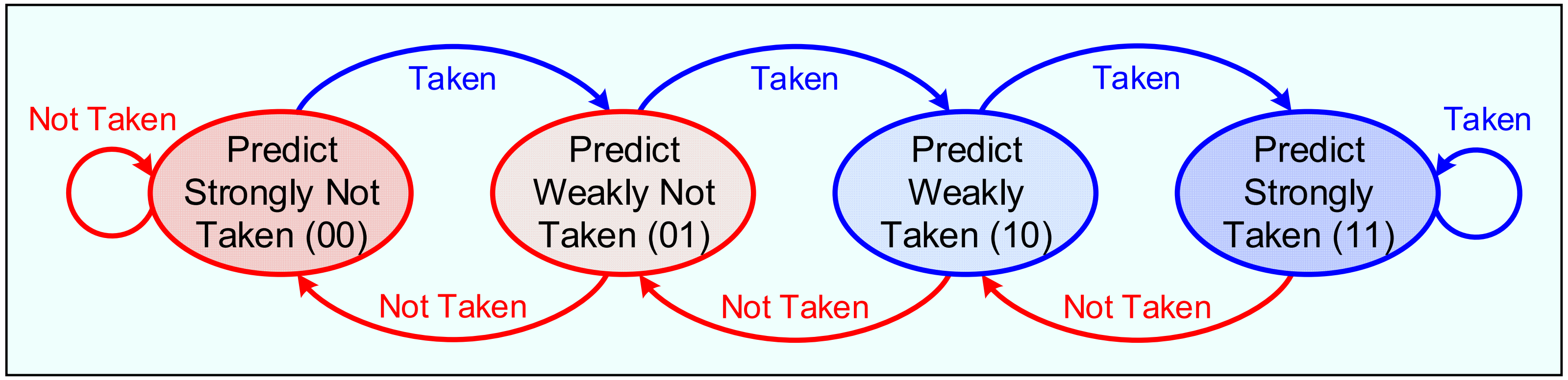
Improving the 1-Bit Predictor

- **Problem:** A 1-bit predictor changes its prediction from T→NT or NT→T too quickly
    - Even though the branch may be mostly taken or mostly not taken

- **Solution Idea:** Add hysteresis to the predictor so that prediction does not change on a single different outcome
    - Use two bits to track the history of predictions for a branch instead of a single bit
    - Can have 2 states for T or NT instead of 1 state for each

2-Bit Predictor

- Prediction does not change on a single misprediction
- 2-Bit entry in BHT => Four States [2 for NT, 2 for T]
  - PSNT: Strongly Not Taken (00), PWNT: Weakly Not Taken (01)
  - PWT: Weakly Taken (10), PST: Strongly Taken (11)
- 2-Bit Counter
  - Increment by 1 if branch taken, otherwise decrement by 1
  - Saturate the counter value at 0 and 3
  - **A prediction must be wrong twice (consecutively) before the prediction bit is changed**


Not Taken
Predict Strongly Not Taken (00)
Taken
Predict Weakly Not Taken (01)
Taken
Predict Weakly Taken (10)
Taken
Predict Strongly Taken (11)
Taken
Not Taken
Not Taken
Not Taken

Example Nested Loop Program - Dynamic Branch Prediction (2bit Global)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  N=No, Y=Yes
  Misprediction rate:

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y												
BTB entry L11	Y												
Global BHT	PWNT												
Prediction	NT												
Direction	-												
Correct?	-												

Example Nested Loop Program - Dynamic Branch Prediction (2bit Global)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  N=No, Y=Yes
  Misprediction rate:

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y	Y	Y	Y	Y								
BTB entry L11	Y	Y	Y	Y	Y								
Global BHT	PWNT	PWNT	PWT	PST	PST								
Prediction	NT	NT	T	T	T								
Direction	-	T	T	T	NT								
Correct?	-	*N*	*Y*	*Y*	*N*								

Example Nested Loop Program - Dynamic Branch Prediction (2bit Global)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  N=No, Y=Yes
  Misprediction rate: ~20% (1 out of five)

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Repeats

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	…	…
BTB entry L11	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	…	…
Global BHT	PWNT	PWNT	PWT	PST	PST	PWT	PST	PST	PST	PST	PWT	…	…
Prediction	NT	NT	T	T	T	T	T	T	T	T	T	…	…
Direction	-	T	T	T	NT	T	T	T	T	NT	T	…	…
Correct?	-	*N*	*Y*	*Y*	*N*	*Y*	*Y*	*Y*	*Y*	*N*	*Y*	…	…

2-bit Predictor: Limits

- Still penalty on regular patterns:
  - Recap: Inner loop iterations: T-T-T-NT
  - Branches often show such regular patterns
- Can we incorporate this regularity? -> Use a history

- Two-level-history adaptive branch predictors (many variants *)
  - Learn the history and loop pattern T-T-T-NT
  - They usually can have higher accuracy
  - This is still 90ties technology *

*Tse-Yu Yeh and Y. N. Patt, "A Comparison Of Dynamic Branch Predictors That Use Two Levels Of Branch History," *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, CA, USA, 1993

- Modern branch predictors for complex processors
  - Based on neural networks
  - Learn patterns, history and interrelation between branches
  - Can achieve very small misprediction rates

Optional, not relevant for exam

A Look at a Real Processor – CVA6

“**CVA6** is a RISC-V compatible application processor core that can be configured as a 32- or 64-bit core: **CV32A6** and **CV64A6”.**

--- CVA& User Manual

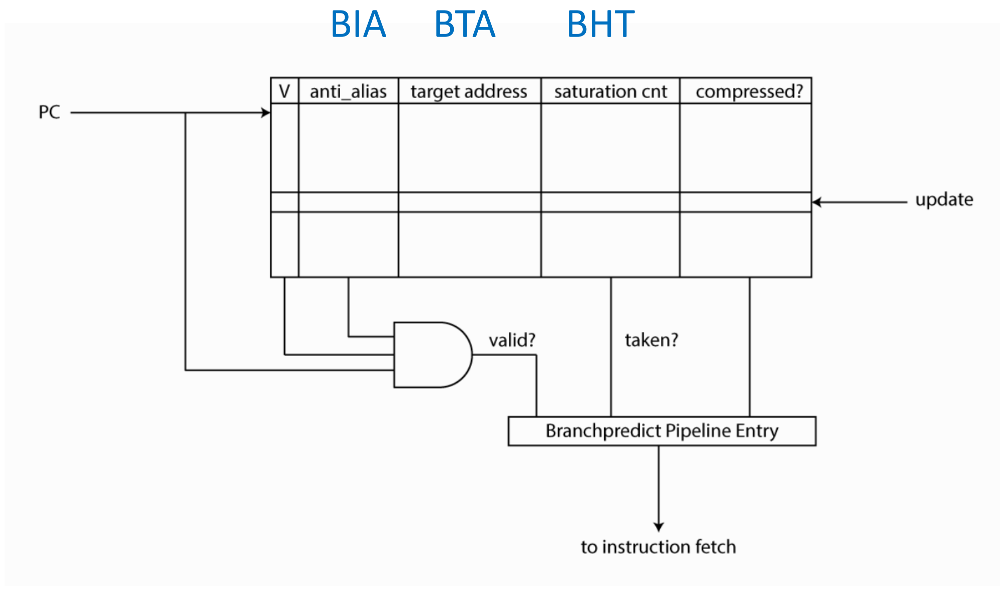
https://docs.openhwgroup.org/projects/cva6-user-manual/01_cva6_user/Introduction.html

Developed initially as part of PULP project (ETH Zürich), now maintained by the OpenHW Group

CVA6 Branch Predictor

"**Branch Predict**: If the BHT and BTB predict a branch on a certain PC, PC Gen sets the next PC to the predicted address and also informs the IF stage that it performed a prediction on the PC. (…)"

„All branch prediction data structures reside in a single register-file like data structure. It is indexed with the appropriate number of bits from the PC and contains information about the predicted target address as well as the outcome of a **configurable-width saturation counter (two by default).** The prediction result is used in the subsequent stage to jump (or not)."



-- CVA6 Design Document (deprecated) – Branch Prediction (05.04.2024)
https://docs.openhwgroup.org/projects/cva6-user-manual/03_cva6_design/pcgen_stage.html

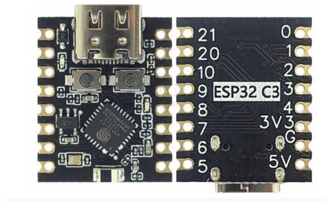
Optional, not relevant for exam

A Look at a Real Processor – ESP32-C3

ESP32-C3 Technical Reference Manual

https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf#riscvcpu

Low Power Mikro-Controller – ESP32-C3



Picture: Alibaba-
Costs less than 1€

Scalar in-order processors with five or less pipeline stages are used in low-cost **micro-controller-type** devices.

„ESP-RISC-V CPU is a 32-bit core based upon RISC-V ISA comprising base integer (I), multiplication/division (M) and compressed (C) standard extensions. The core has **4-stage, in-order, scalar pipeline** optimized for area, power and performance. (…)“

-- ESP32-C3 Technical Reference Manual
https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf#riscvcpu

Optional, not relevant for exam

Trap Handling

Terminology

- Terminology is used often different for different architectures (x86,ARM, RISCV,…).

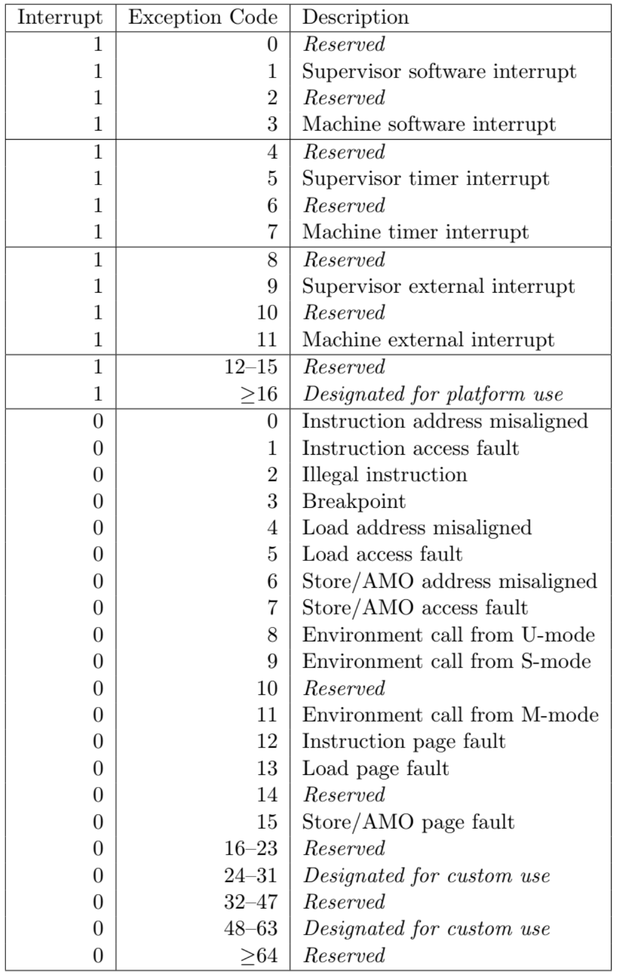
- For RISC-V:
- “We use the term **exception** to refer to an unusual condition occurring at run time associated with an instruction in the current RISC-V hart.”
- “We use the term **interrupt** to refer to an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control.”
- “We use the term **trap** to refer to the transfer of control to a trap handler caused by either an exception or an interrupt.”

—- Volume 1, Unprivileged Specification version 20191213: https://riscv.org/technical/specifications/

Trap Handling

- For a function call the compiler assures that the function call standard of the ABI is kept
- An exception and interrupt can happen during execution of a function ceither due to an instruction (e.g. memory access error) or due to an external event (device raises an interrupt)
- For a trap, we are in the middle of execution of a function and must save the context of the current execution before calling a trap handler to handle the exception or interrupt
- RISC-V has certain so-called **Control Status Registers (CSRs)** to identify the cause of a trap

Causes for Traps



Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

- —- Volume 2, Privileged Specification version 20211203: https://riscv.org/technical/specifications/

  Word trap is mentioned 301 times

- Different architectures treat exceptions differently e.g. division by zero is not raising an exception in RISC-V

Trap Handling Basics

- Trap is detected.
- Change mode
- Jump to trap handler
- Trap handler saves context
- Trap handler identifies cause (exception/interrupt)
- Corresponding exception/interrupt handler is called
  - Some handlers do not return if they can not recover from an exception
- Trap handler restores context
- Change mode
- Jump back to program execution

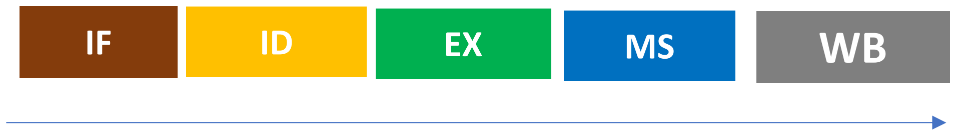
Precise vs. Imprecise traps

- Precise traps:
  - Associated with a certain instruction (e.g. illegal instruction exception)
  - Easier to debug
- Imprecise trap:
  - Not associated with an instruction
  - Hard to debug
  - OR: Pipelined execution makes it hard to associate the exception with an instruction (This is an issue with certain pipelines, which we see in next lecture)

Summary

Where we are

- Five-Stage Scalar In-order Processor Pipeline
  - Forwarding to mitigate data hazards
  - Branch prediction to mitigate control hazards



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

- Upcoming Lecture: Multi-cycle Functional Units (DIV/MUL) and Out-of-Order (OoO)

Thank you for your attention!

BACKUP

Registers of RISC-V

- RISC-V has 32 integer registers
- Processors can have different register width, we look at RV32 with 32-bit width
- Each register has two IDs (x0-x31) and **an ABI name** that indicates its role
- ABI stands for Application Binary Interface (ABI)

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	Saved register/frame pointer	Callee
x9	s1	Saved Register	Callee
x10-11	**a0-1**	**Function arguments, Return values**	**Caller**
x12-17	**a2-7**	**Function arguments**	**Caller**
x18-27	s2-11	Saved registers	Callee
x28-31	**t3-6**	**Temporaries**	**Caller**

Application Binary Interface (ABI) – Function Call Convention

- ABI also specifies rules for register usage in passing arguments and results for function calls
  - **Callee-saved registe**rs: If function foo1 (caller) calls foo2 (callee), then foo2 is not allowed to modify this value (it needs to save it and restore it before returning to foo1)
  - **Caller-saved registers**: If function foo1 (caller) calls foo2 (callee), then foo1 needs to save this register before calling foo2 if it wants to keep the value in it because foo1 is allowed to modify it

- According to ABI parameters are passed to a function in registers a0-a7
- The function should return its return value in register a0 (if <=32-bit value)

RISC-V Instructions

- The RISC-V ISA is modular with base instruction sets and a large variation of extensions
  - We look at **RV32IM**
- 32-bit Integer Instruction Set RV32I
  - Integer Register-Register Instructions (R-type)
    - Runs an arithmetic or logical operation on registers
    - Both operands are values in registers
  - Integer Register-Immediate Instructions (I-type)
    - Second operand is an immediate (constant) value
  - Control Transfer Instructions
    - Unconditional jumps
    - Conditional Branches
  - Load Store Instructions
    - Move data between memory and registers
    - Load-store Architecture: Operations on registers only
- 32-bit Integer Multiplication RV32M Extension -> Next Session
  - Integer Multiplication Instructions
  - Integer Division Instructions

Computer Systems

Advanced Processor Pipelines 2

Daniel Mueller-Gritschneder

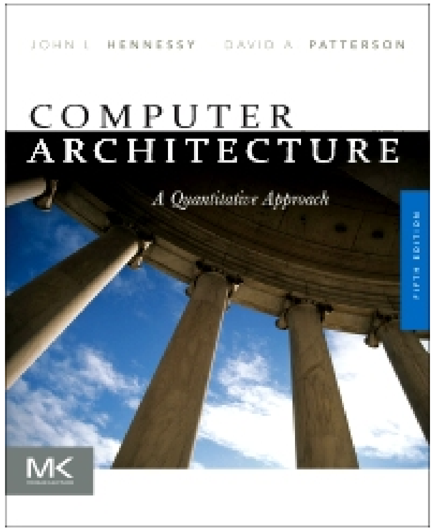
22.04.2024 V04g

Sources

This book covers the basics of how to design a simple in-order scalar processor pipeline in detail in hardware.

- Literature: „Digital Design and Computer Architecture: RISC-V Edition“, by Sarah L. Harris and David Harris
  - https://shop.elsevier.com/books/digital-design-and-computer-architecture-risc-v-edition/harris/978-0-12-820064-3
  - https://pages.hmc.edu/harris/ddca/ddcarv.html (Includes resources for students!)
  - They also provide slideshows – the basis for ours! You can investigate extended version at their website.
- Available at TU's library: https://catalogplus.tuwien.at/permalink/f/qknpf/UTW_alma21139903990003336

Sources



So-called application processors have many additional features:
Branch prediction, Out of order execute, Scoreboard, Superpipelining, Multi-issue, Superscalar, VLIW, Multi-threading, …

Disclaimer: The book provides advanced concepts from real complex processor designs. We only study the concepts at a high level. For simplicity, the used pipeline models in this lecture are reduced strongly in complexity.

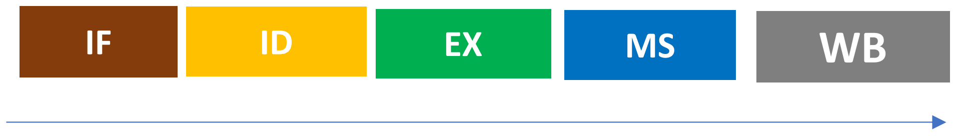
But: We will have a look at some current RISC-V processor designs

Literature: „**Computer Architecture A Quantitative Approach”** 5th Edition - September 16, 2011
Authors: John L. Hennessy, David A. Patterson eBook ISBN: 9780123838735

- https://shop.elsevier.com/books/computer-architecture/hennessy/978-0-12-383872-8

- Available at TU’s library:
https://catalogplus.tuwien.at/permalink/f/8agg25/TN_cdi_askewsholts_vlebooks_9780123838735

RECAP: Five-Stage In-Order Scalar Pipeline



- Five Stage
- In-order pipeline
- Scalar pipeline

- Each stage takes one cycle to complete

➢Single access cycle to instruction and data memory: Works for small and slow micro-controller-type processors with on-chip embedded SRAM memories

➢Single cycle operations, works for simple instructions (ADD, Compare,…)

`SLLI a2,a1,2`	IF	ID	EX	MS	WB		
`ADD a2,a0,a2`		IF	ID	EX	MS	WB	
`LW a0,0(a2)`			IF	ID	EX	MS	WB
`ADD a0,a0,a1`				IF	ID	stall	EX
`RET`					IF	stall	ID

- Scalar processor: Can execute at maximum 1 instruction per cycle (IPC <=1)

Content

- Multi-cycle Functional Units (FUs)
- Load and Store Optimizations
- Instruction Dependencies (RAW, WAW, WAR)
- Dynamic Scheduling with Scoreboard (Out of Order – OoO)
- Register Renaming
- Superscalar

Optional, not relevant for exam

- A look at a real RISC-V processor: CVA6
- Pipeline Support for Precise Traps

Multi-Cycle Operations

Integer Multiplication Instructions

- Signed-signed Multiplication
  - Multiplying two 32bit values can result in a value of up to 64 bit
  - `MUL a3,a1,a2`
    - Behavior: a3 ← a1*a2 // only the lower 32bit
  - `MULH a4,a1,a2`
    - Behavior: a4 ← a1*a2 // only the higher 32bit
  - Example:
    - `MULH a4,a1,a2`
    - `MUL  a3,a1,a2`
      Behavior: [a4 a3] = a1*a2 // full 64 bit
- Unsigned-unsigned multiplication `MULHU`
- Signed-Unsigned multiplication `MULHSU`

Integer Division Instructions

- Signed-signed Division
  - `DIV a3,a1,a2`
    - Behavior: a3 ← a1 / a2
  - `REM a4,a1,a2`
    - Behavior: a4 ← a1 modulo a2 // remainder
- Unsigned-unsigned division `DIVU, REMU`

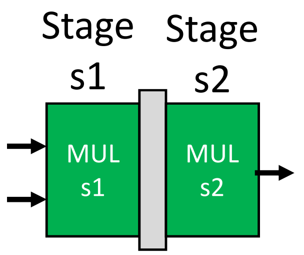
Pipelined Functional Units (FUs)

- Complex computations require deep circuit logic
- Critical path in deep logic limits the design's frequency
- Similar to processor design, break FU into stages and integrate registers to build a pipeline

➢ **Latency** (in cycles) equals to number of pipeline stages

➢ **Initialization Interval**: Delay (in cycles) between start of two computations

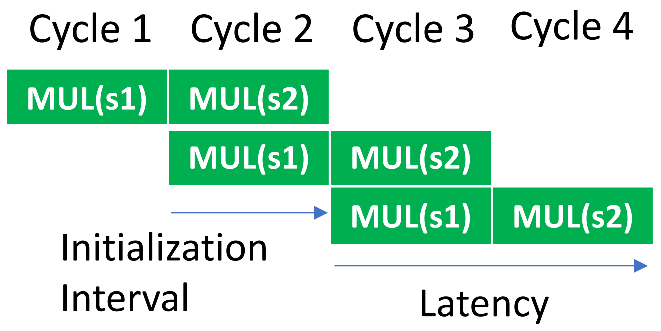
- Example: 2-stage Multiplier



Latency = 2 Cycles

Initialization Interval = 1 Cycle

```
MUL a0,a0,t0
MUL a1,a1,t1
MUL a2,a2,t2
```



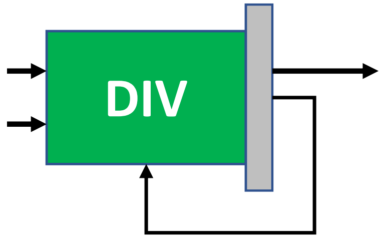
Serial Functional Units (FUs)

- Often complex operations such as divisions can be computed by iterative algorithms
- The number of iterations (required clock cycles) often depends on the input values
- These iterations can be implemented on a serial FU, which is busy as long as it computes

➢ **Latency** equals to number of cycles required for computation

➢ **Initialization Interval** equals to number of cycles required for computation

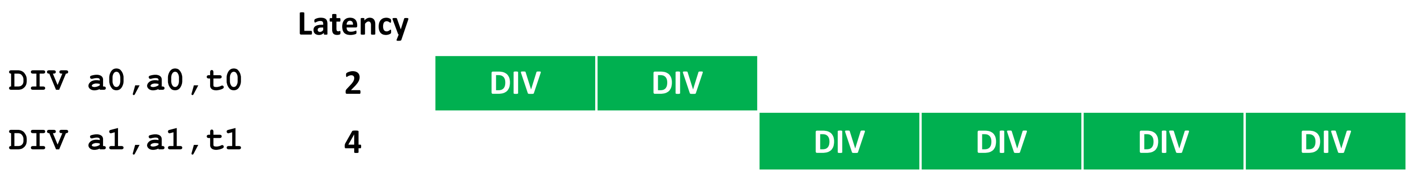
- Example: Serial Divider



1-64 clock cycles

Latency = 1-64 Cycles

Initialization Interval = Latency



	Latency
`DIV a0,a0,t0`	2
`DIV a1,a1,t1`	4

Example: RISC-V CVA6 Processor

“Multiplier

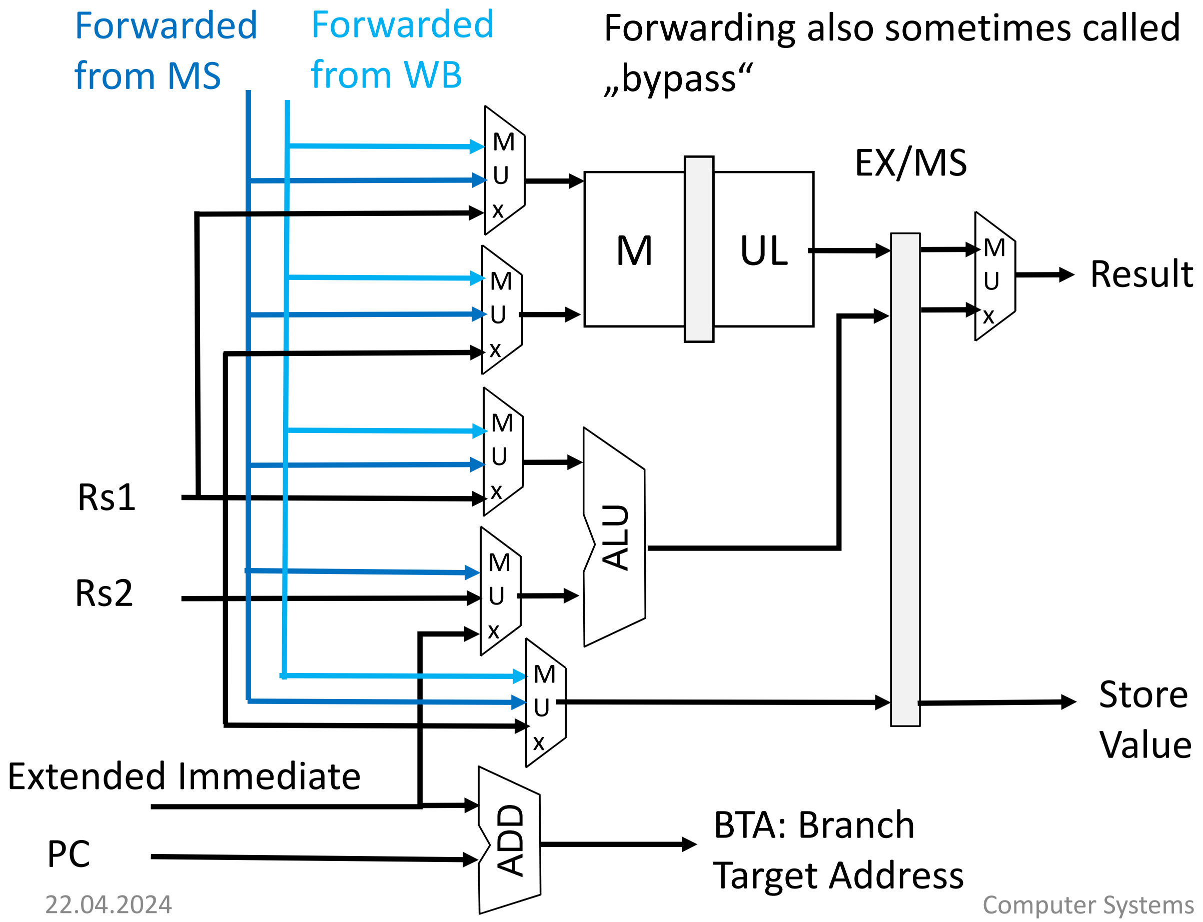
The multiplier contains a division and multiplication unit. Multiplication is performed in two cycles and is fully pipelined (re-timing needed). The division is a simple serial divider which needs 64 cycles in the worst case.”*

*https://docs.openhwgroup.org/projects/cva6-user-manual/03_cva6_design/ex_stage.html

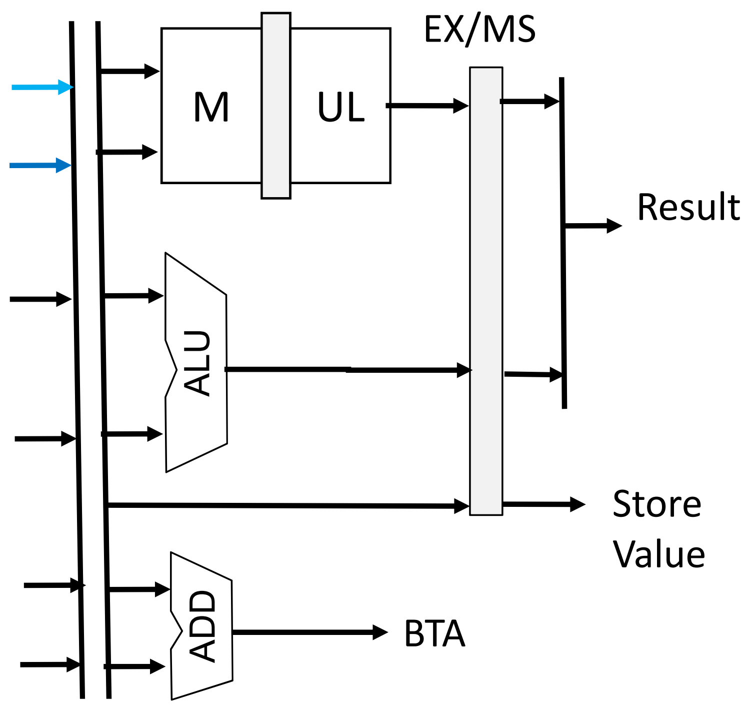
Integration of Multi-cycle Functional Units

- Multi-cycle Functional Units are integrated into the EX stage
- Example only for Multiplier

Forwarding also sometimes called „bypass“

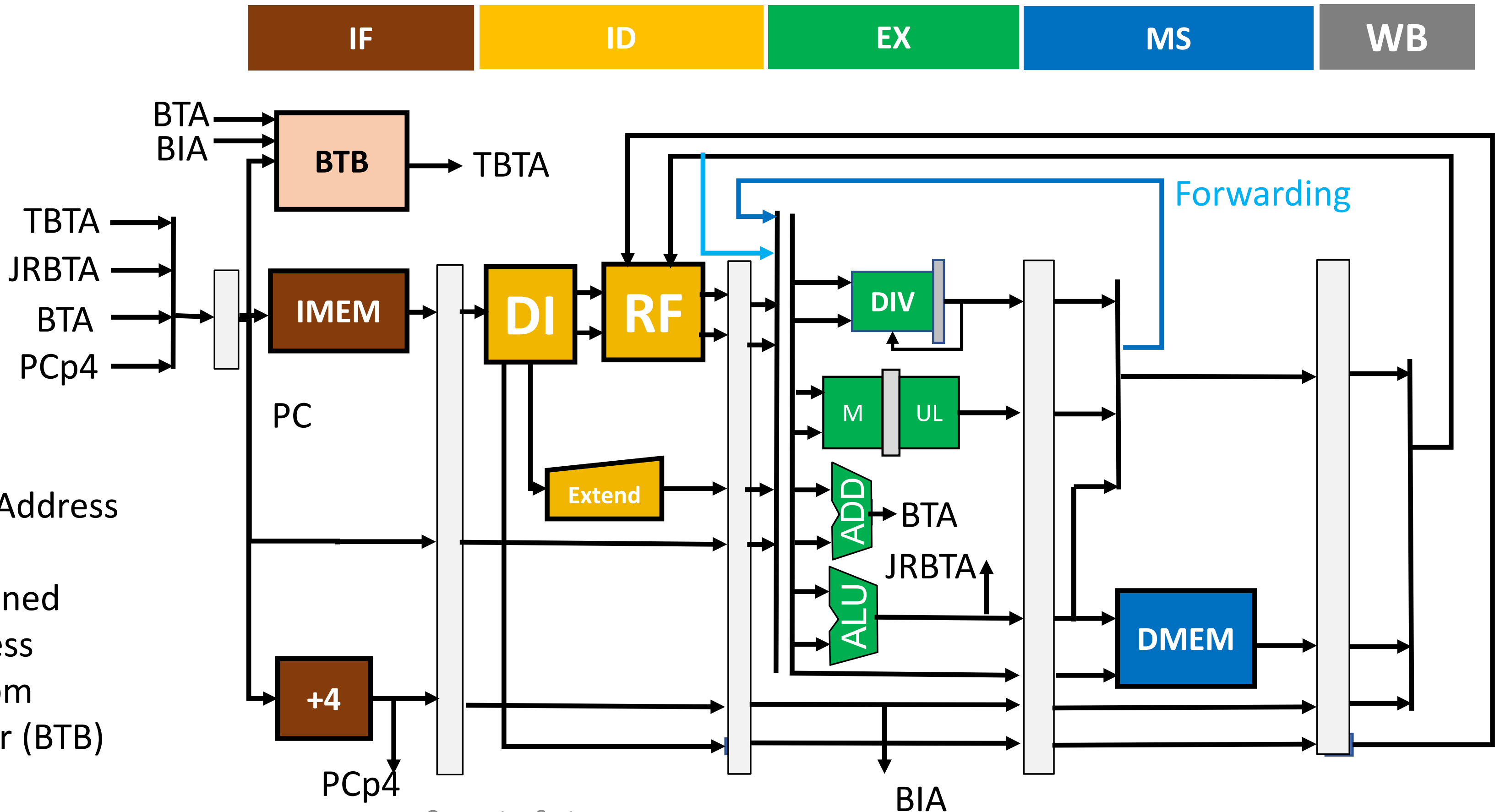


Simplified Illustration Style for Multiplexing



Scalar Five-Stage Pipeline with Multi-cycle FUs and Forwarding

- Multi-cycle Functional Units are integrated into the EX stage
- Simplified diagram

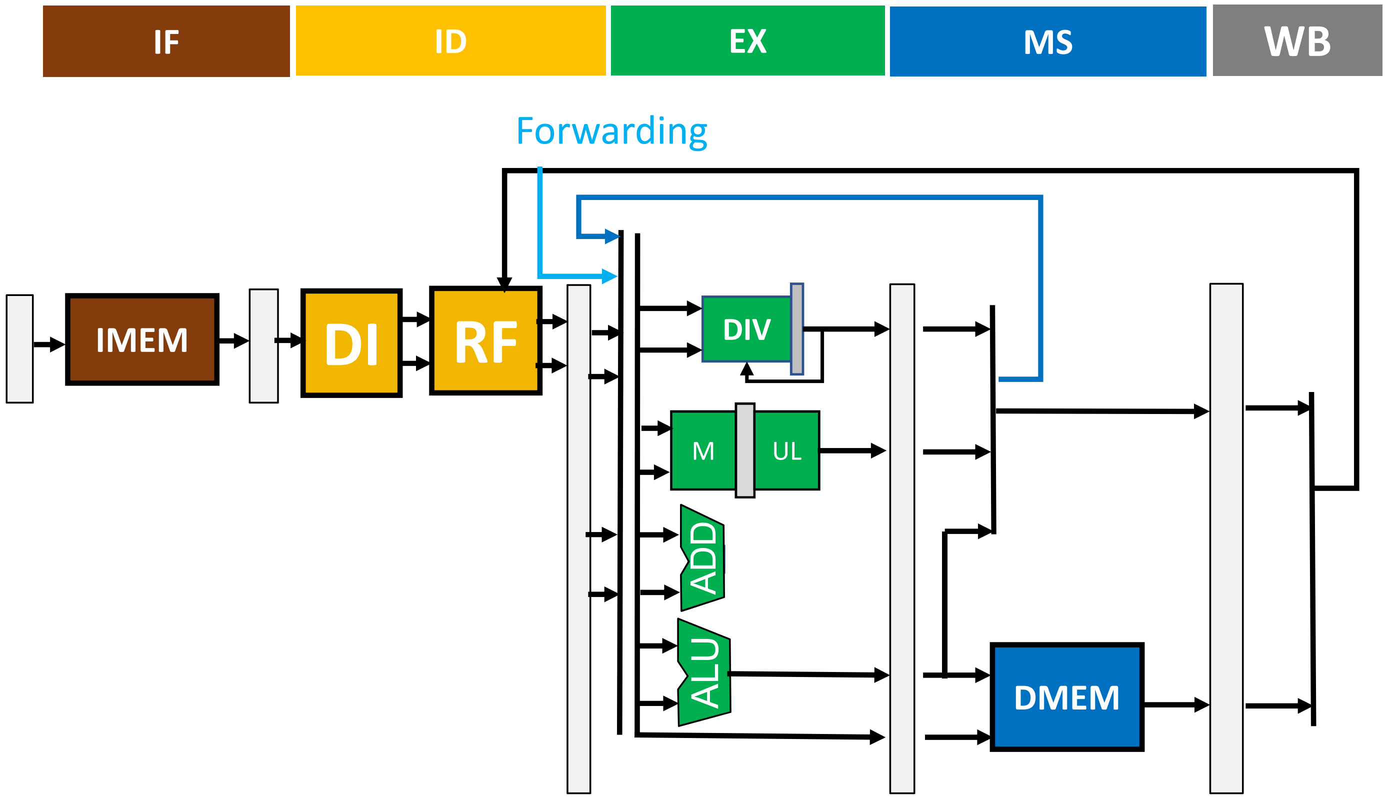


BTA: Branch Target Address
PCp4: PC+4
JRBTA: Register-defined branch target address
TBTA: Taken-BTA from Branch Target Buffer (BTB)

Scalar Five-Stage Pipeline with Multi-cycle FUs and Forwarding

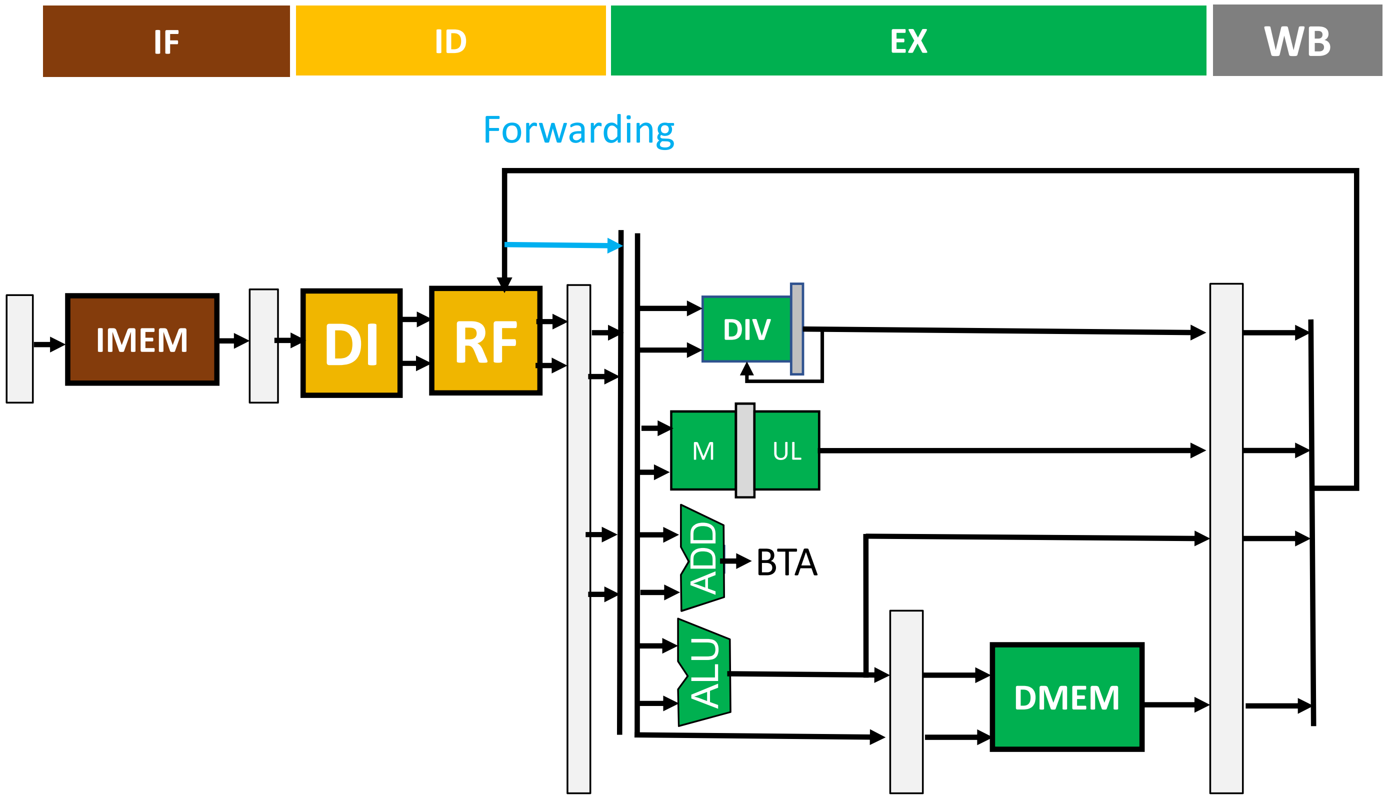
- Multi-cycle Functional Units are integrated into the EX stage
- **Further simplified diagram** (PC Generation, Extend, PC+rd address not shown, but of course still needed!)

Focus on the computation flow



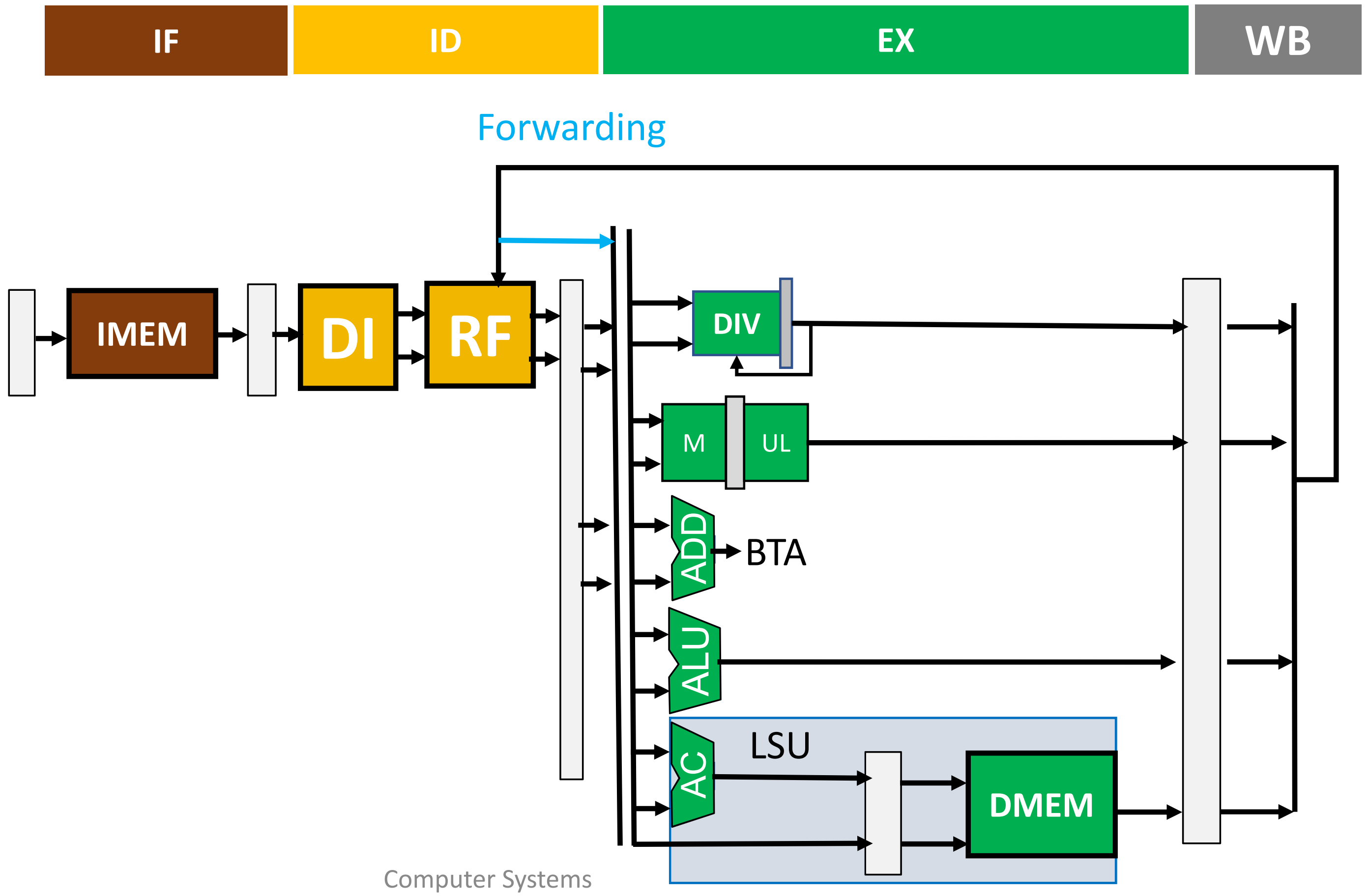
Scalar Four-Stage Pipeline with Multi-cycle FUs with Forwarding

- The DIV and MUL do not need to make memory accesses
- Move the memory stage (MS) after the ALU (which is required for the address computation for load/store)
- Merges MS and EX stage (four stages)
- Single forwarding path required in four-stage pipeline
- Such changes need additional control in control path



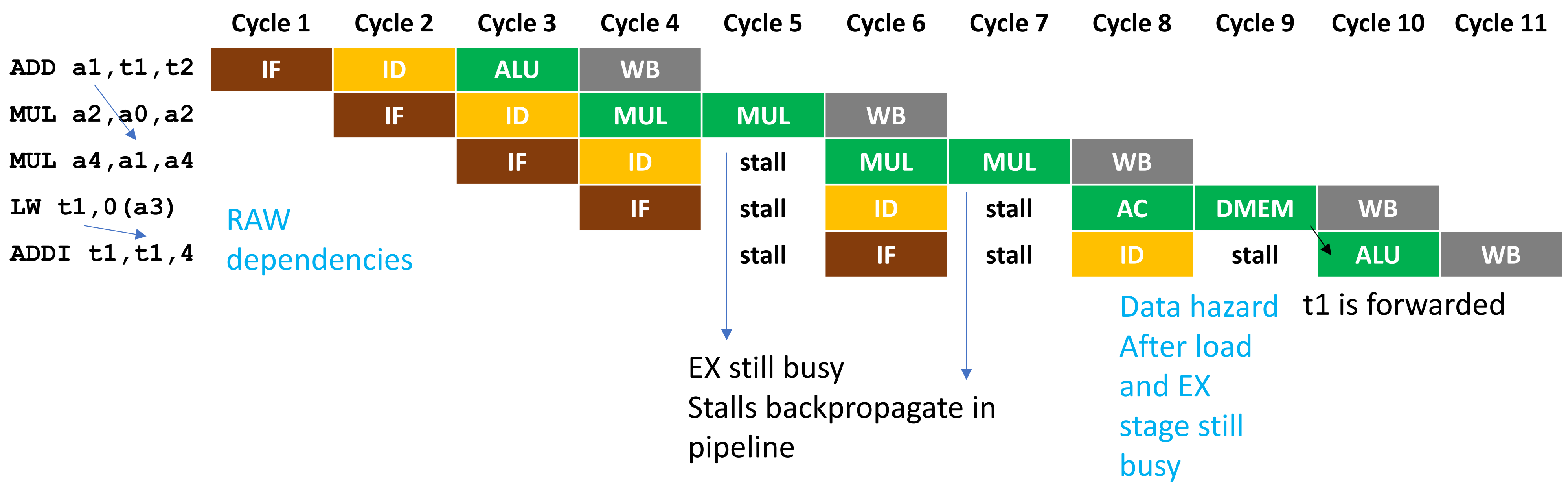
Scalar Four-Stage Pipeline with Multi-cycle FUs and Load Store Unit (LSU)

- We can add a second address computation adder (AC) to form a simple so-called load/store unit (LSU)



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

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

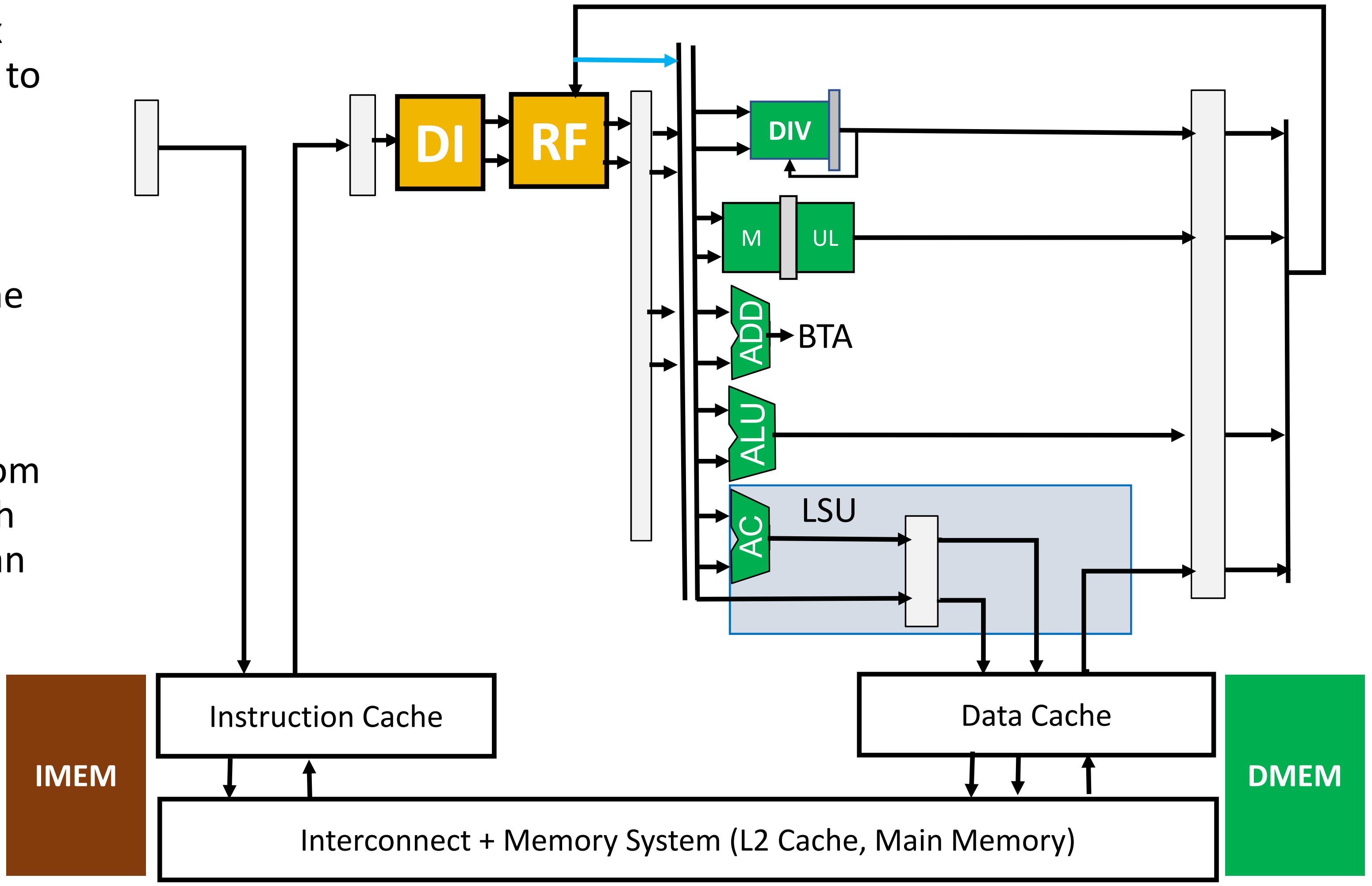
➢ Except for: Pipelined MUL can use Initialization Interval for two consecutive MUL (still need to check for RAW dependency between the MUL)

	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

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.



Instruction Cache Misses

- Instruction cache miss causes several cycles of delay for instruction fetch (IF), depending on speed to catch fresh instruction block from memory system

- Instructions are usually reloaded to cache in blocks (cache line size) so that usually there are several cache hits after a cache miss (depending on jumps/branches in program)

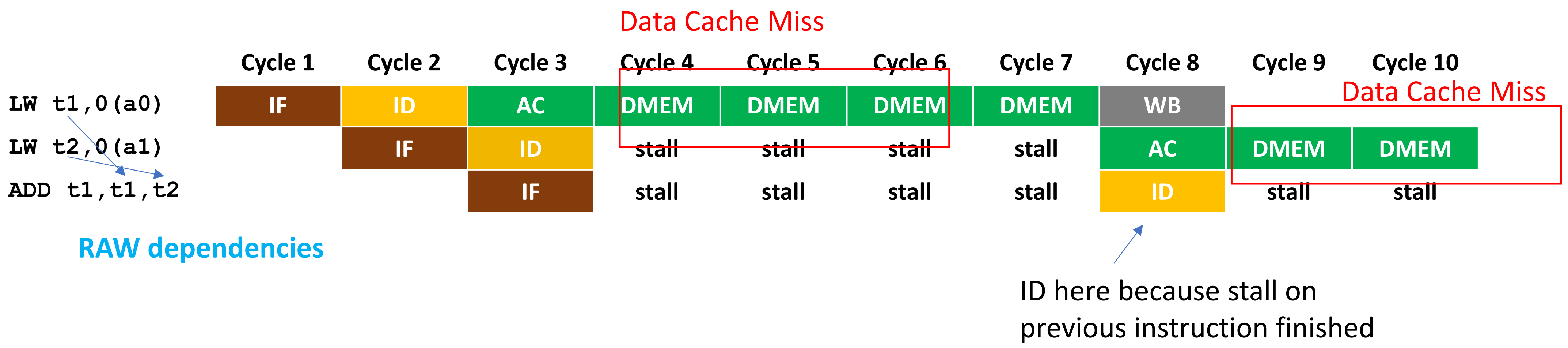
	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
`ADD a2,t1,t2`	IF	ID	ALU	WB						
`MUL a2,a0,a2`		IF	IF	IF	IF	ID	MUL	MUL	WB	
`MUL a4,a1,a4`						IF	ID	MUL	MUL	WB

Instruction Cache Miss

- Advanced caches pre-fetch the next block before the cache miss happens to hide cache refill latencies.

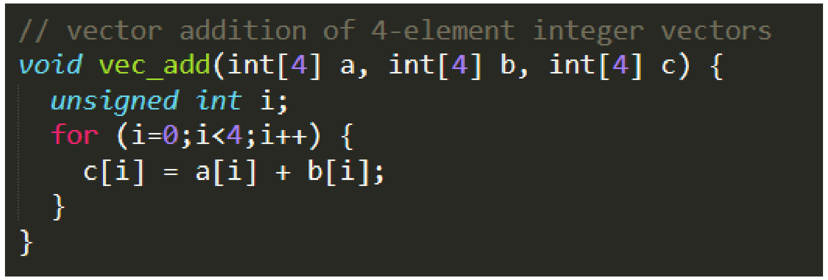
Load Cache Miss

- Data cache misses lead to extra cycles for loads as the data needs to get fetched from another memory (level 2 cache, main memory)
- Example (function vec_add, see first session): We load from two different addresses a0 and a1 (worst case both loads lead to a data cache miss)



Example vec_add: Loads from two different addresses (a0,a1)

- Example C-Code 3



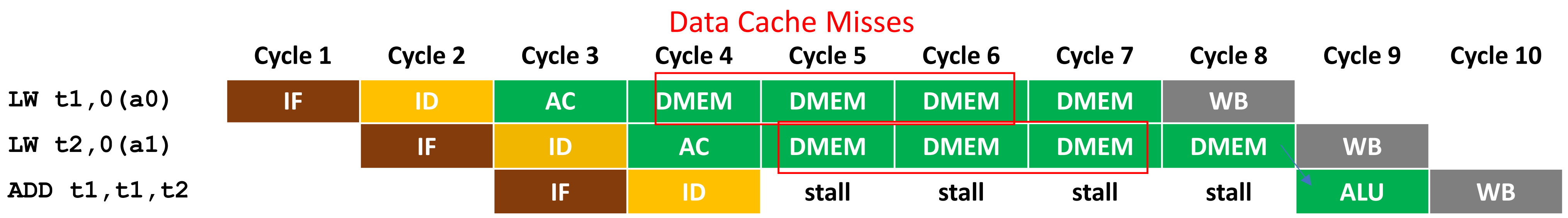
```c
// vector addition of 4-element integer vectors
void vec_add(int[4] a, int[4] b, int[4] c) {
  unsigned int i;
  for (i=0;i<4;i++) {
    c[i] = a[i] + b[i];
  }
}
```

RISC-V Code

```
# base address of a: a0,
# base address of b: a1,
# base address of c: a2,
# i: t0,  constant 4: t3
vec_add:
  LI t0,0        # i=0
  LI t3,4        # t3=4
vec_add_for:
  LW t1,0(a0)     # t1 = a[i]
  LW t2,0(a1)     # t2 = b[i]
  ADD t1,t1,t2    # t1 = a[i] + b[i]
  SW t1,0(a2)     # c[i] = t1
  ADDI a0,a0,4    #next element is base address + 4
  ADDI a1,a1,4    #next element is base address + 4
  ADDI a2,a2,4    #next element is base address + 4
  ADDI t0,t0,1    # i++
  BLTU t0,t3,vec_add_for # for (i < 4)
  RET   # void return
```

Nonblocking Loads (1/2)

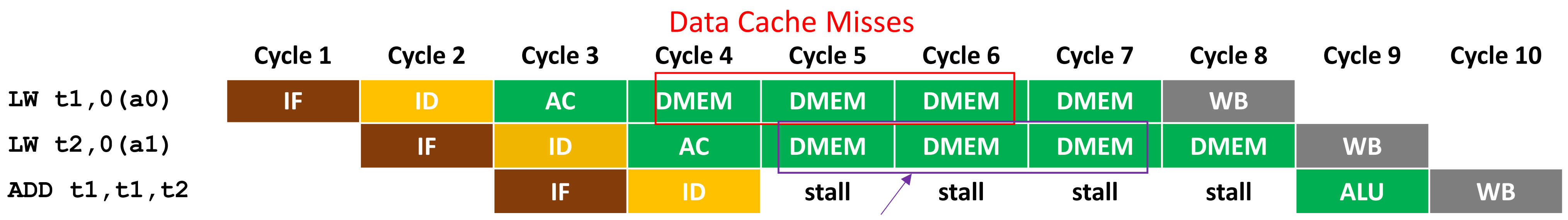
- Load accesses are for longer times *in flight* due to cache misses
- Most interconnects/caches allow to overlap multiple memory accesses
- Allows to execute multiple load accesses in overlapping fashion
- Example (function vec_Add): Cache observes both addresses for load accesses and may need to reload cache lines for both accesses when both miss.



	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

Nonblocking Loads (2/2)

- Cache usually returns values in-order (some caches/interconnects support to return data out-of-order)
- Example (function 3): When only the first load misses, the second load still needs to wait in the LSU when the LSU returns results in-order.



	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.

Store Cache Miss

- Depending on Store Policy: Write-back data cache:
  - Additional latencies for stores possible when a dirty cache line needs to be replaced.
  - Dirty cache line needs first to be written to memory before it can be replaced
- Write through data cache:
  - Long store latency because the data is written not only to cache but also to main memory.

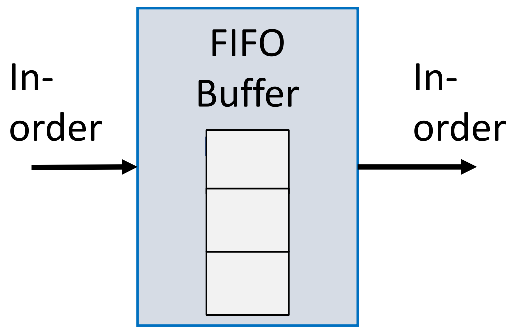
Example: We store to two different addresses a0 and a1 (first store misses)

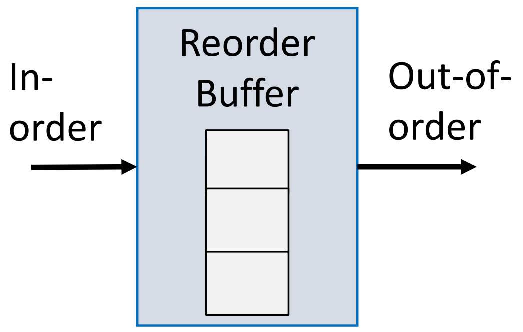
Data Cache Misses

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10	Cycle 11
`SW t1,0(a0)`	IF	ID	AC	DMEM	DMEM	DMEM	DMEM	WB			
`SW t2,0(a1)`		IF	ID	stall	stall	stall	stall	AC	DMEM	WB	
`LI t2,4`			IF	stall	stall	stall	stall	ID	stall	ALU	WB

Buffers

- A buffer can store several values
- FIFO (First-in-first-out) buffer: Values can be read only from the buffer in the same order they are written to the buffer
- Reorder buffer: We can look up and read any value in the buffer





Store Buffer

- It is not really necessary to wait until a store write completes

- Store Unit (SU) with Store Buffer:
  - ➢ Put store address and data to store buffer (sometimes called *“Posted stores”)*
  - ➢ Store buffer performs memory store access (MSA) independently from pipeline
  - ➢ Only stall pipeline for stores when store buffer is full

- Load Unit (LU): Load more complex:
  - ➢ need to first look whether address is in store buffer then in cache
  - ➢ or need to wait until SB is empty.

Nonblocking Stores with Store Buffer

- Store accesses are for longer times *in flight* due to cache misses
- Store Buffer store accesses and pipeline continues execution
- Store Buffer writes data to memory via Memory Store Access (MSA).
- Only stall pipeline for stores when store buffer is full
- Example:

	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	ID	AC	SB	SB	SB	MSA			
`SW t2,0(a1)`		IF	ID	AC	SB	SB	SB	MSA		
`LI t2,4`			IF	ID	ALU	WB				

Execution Scheme: Scalar Four-Stage Pipeline with Pipelined FUs and Load Store Optimization

- Version 3: Static Scheduling with pipelined FUs and Load Store Optimization
  - Allow only one single instruction in EX stage
  - Except for:
    - Pipelined MUL can use Initialization Interval for two consecutive MUL
    - Certain number of nonblocking Loads can be in EX stage (then EX stalls)
    - Certain number of stores can be posted in the SB depending on SB size (EX stalls when SB full). When Store is posted in SB, it does not count as instruction in EX stage.

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10	Cycle 11	Cycle 12
`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					
`SW a2,0(a3)`				IF	ID	stall	AC	SB	SB	MSA		
`ADDI a3,a3,4`					IF	stall	ID	ALU	WB			
`SW a2,0(a3)`						stall	IF	ID	AC	SB	SB	MSA

Performance of Scalar Four-Stage Pipeline with Pipelined FUs and Load Store Optimization

- We still only allow one instruction to execute in EX stage except for some instruction types (MUL, Store, Load) in Version 3
- Multi-cycle operations cause many stalls (stiff scalar execution scheme)

	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					
`DIV a4,a1,a4`			IF	ID	stall	DIV	DIV	DIV	DIV	WB	
`LW t1,0(a3)`				IF	stall	ID	stall	stall	stall	AC	…
`ADDI a3,a3,4`					stall	IF	stall	stall	stall	ID	…

- Can we interleave instructions to make better use of parallel units, maybe even just start them when they are ready, possibly out-of-order (OoO)?
- We want to exploit so-called **Instruction Level Parallelism**

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!

Challenges for Exploiting Instruction Level Parallelism: Instruction Dependencies

- Start instructions in EX stage when FUs are available?

➢Instructions can *overtake* each other due to different FU latencies.

- **Challenge:** The assembly program defines a **program order** for the instructions.
- Requires consideration of instruction dependencies during pipelined execution to preserve program order.

	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	MUL	WB				
`DIV a4,a1,a4`			IF	ID	DIV	DIV	DIV	DIV	WB	
`SW a4,0(a3)`				IF	ID	AC	MSA			
`ADDI a4,a3,4`					IF	ID	ALU	WB		

RAW dependency was ignored (data hazard!)

DIV must write back result first
So-called Write-after-Write
(WAW) dependency

Instruction Dependencies

A closer look at RAW, WAR and WAW!

Types of Instruction Dependencies

- Read-after-Write (RAW): Also „*True dependency*“
  - Result of one instruction (write) is needed as input for another instruction (read)
  - May cause data hazards (*we seen this one already*)
- Write-after-Read (WAR): Also „*anti-dependency*“
  - A value is used (read) and then updated (write)
  - The update (write) is not allowed to overtake the use (read)
- Write-after-Write (WAW): Also „*output dependency*“
  - A value us updated (write) and then updated again (write)
  - The second update may not overtake the first update
  - Often created when registers are reused for different variables

Example for RAW:

XOR **a1**,a2,a4

RAW

ADD a3,**a1**,t1

Example for WAR:

SW a1,0(**a2**)

WAR

ADDI **a2**,a3,4

Example for WAW:

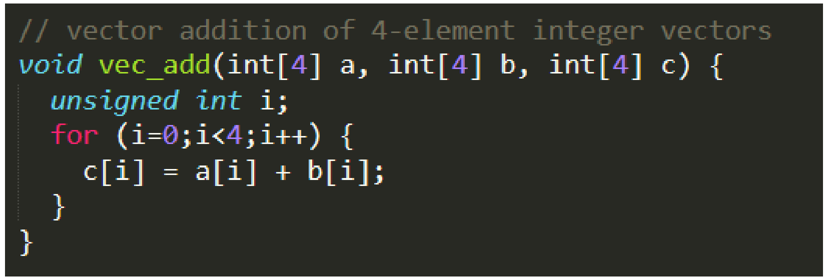
LW **a1**,0(a2)

WAW

LI **a1**,a3,4

Dep. For Example Program (vec_add)

- Example C-Code 3



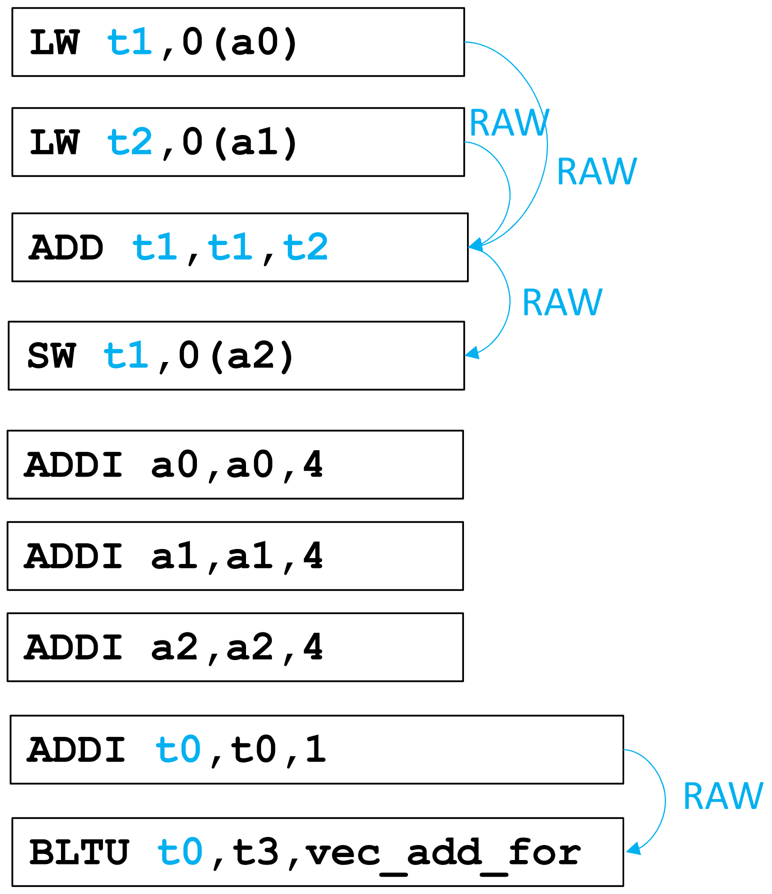
```c
// vector addition of 4-element integer vectors
void vec_add(int[4] a, int[4] b, int[4] c) {
  unsigned int i;
  for (i=0;i<4;i++) {
    c[i] = a[i] + b[i];
  }
}
```

```
# base address of a: a0,
# base address of b: a1,
# base address of c: a2,
# i: t0,  constant 4: t3
vec_add:
  LI t0,0       # i=0
  LI t3,4       # t3=4
vec_add_for:
  LW t1,0(a0)    # t1 = a[i]
  LW t2,0(a1)    # t2 = b[i]
  ADD t1,t1,t2   # t1 = a[i] + b[i]
  SW t1,0(a2)    # c[i] = t1
  ADDI a0,a0,4   #next element is base address + 4
  ADDI a1,a1,4   #next element is base address + 4
  ADDI a2,a2,4   #next element is base address + 4
  ADDI t0,t0,1   # i++
  BLTU t0,t3,vec_add_for # for (i < 4)
  RET   # void return
```

Dep. For Example Program (vec_add) (RAW)

- Mark all RAW dependencies for the following code block:

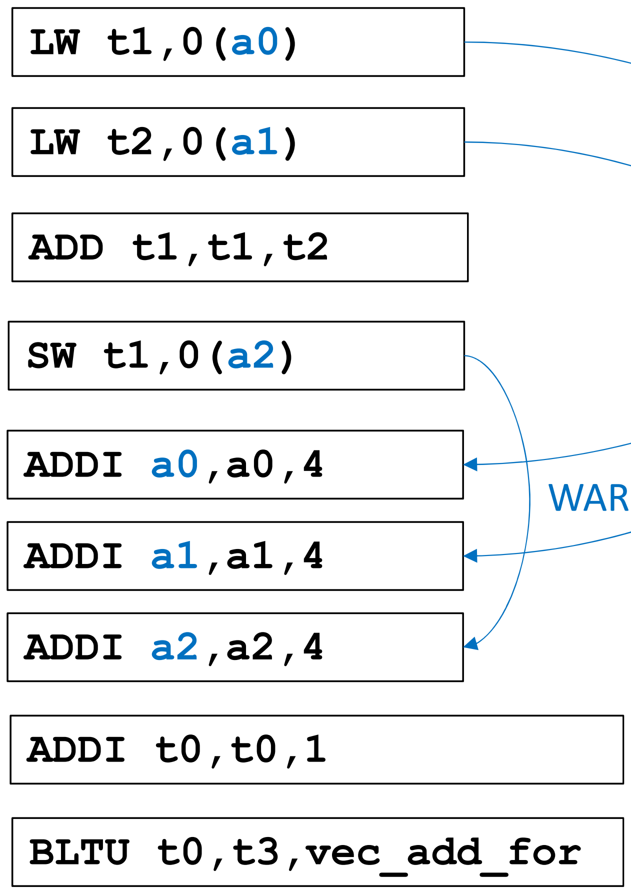
```
LI t0,0
LI t3,4
vec_add_for:
 LW t1,0(a0)
 LW t2,0(a1)
 ADD t1,t1,t2
 SW t1,0(a2)
 ADDI a0,a0,4
 ADDI a1,a1,4
 ADDI a2,a2,4
 ADDI t0,t0,1
 BLTU t0,t3,vec_add_for
 RET
```



Dep. For Example Program (vec_add) (WAR)

- Mark all WAR dependencies for the following code block:

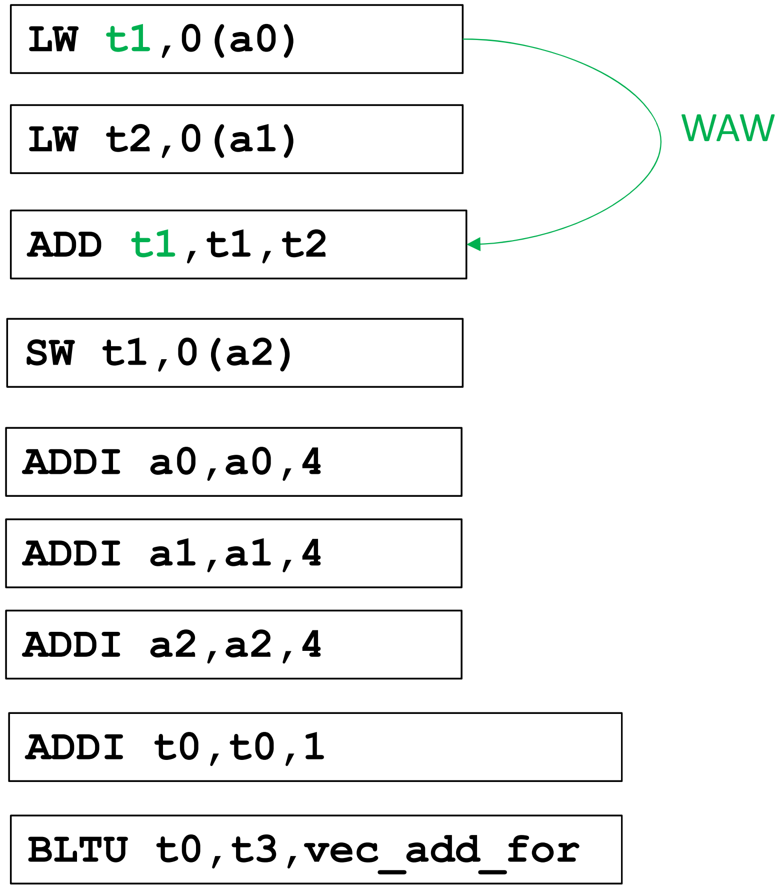
```
LI t0,0
LI t3,4
vec_add_for:
  LW t1,0(a0)
  LW t2,0(a1)
  ADD t1,t1,t2
  SW t1,0(a2)
  ADDI a0,a0,4
  ADDI a1,a1,4
  ADDI a2,a2,4
  ADDI t0,t0,1
  BLTU t0,t3,vec_add_for
  RET
```



Dep. For Example Program (vec_add) (WAW)

- Mark all WAW dependencies for the following code block:

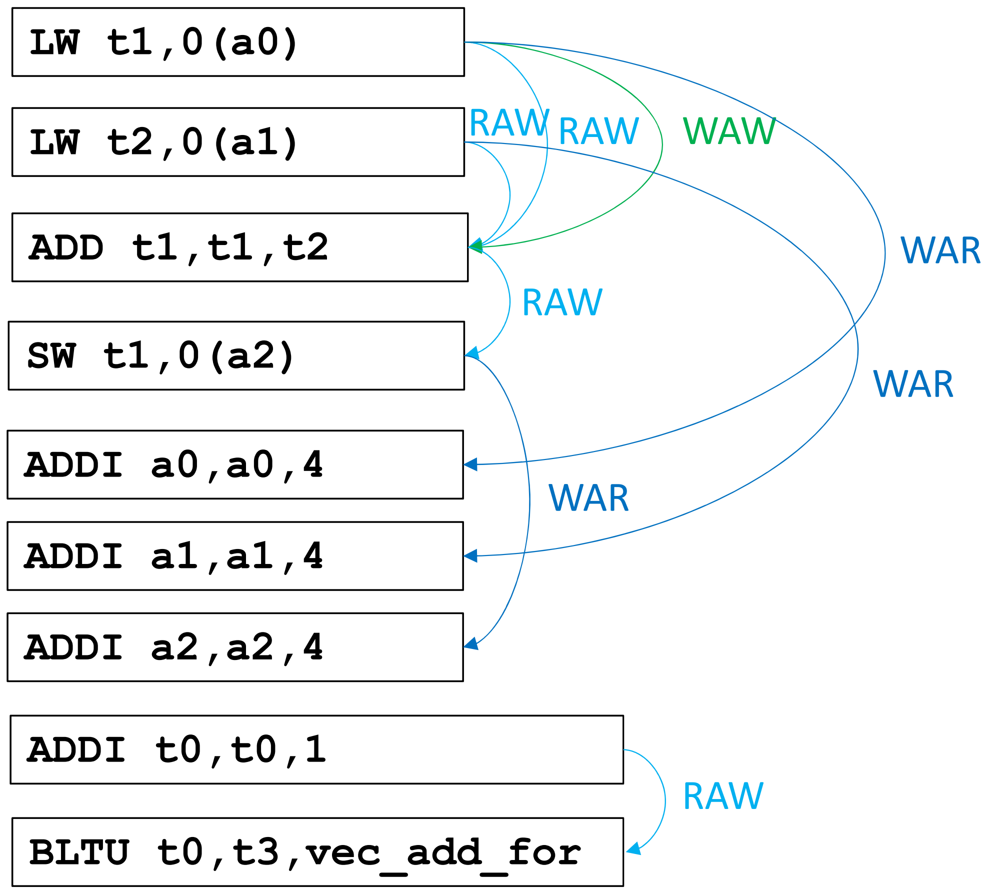
```
LI t0,0
LI t3,4
vec_add_for:
  LW t1,0(a0)
  LW t2,0(a1)
  ADD t1,t1,t2
  SW t1,0(a2)
  ADDI a0,a0,4
  ADDI a1,a1,4
  ADDI a2,a2,4
  ADDI t0,t0,1
  BLTU t0,t3,vec_add_for
  RET
```



Dep. For Example Program (vec_add) (ALL)

- Mark all dependencies for the following code block:

```
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
```



Challenges with Interleaving Instruction Execution in EX Stage

1. We have to consider **RAW, WAR** and **WAW** dependencies.

2. **Structural hazards** must be avoided, e.g., FU is already busy.

3. Some instructions can cause so-called **exceptions** (e.g. memory fault on load/store) (See optional content for what is required for precise exceptions).

Dynamic Scheduling With Scoreboard

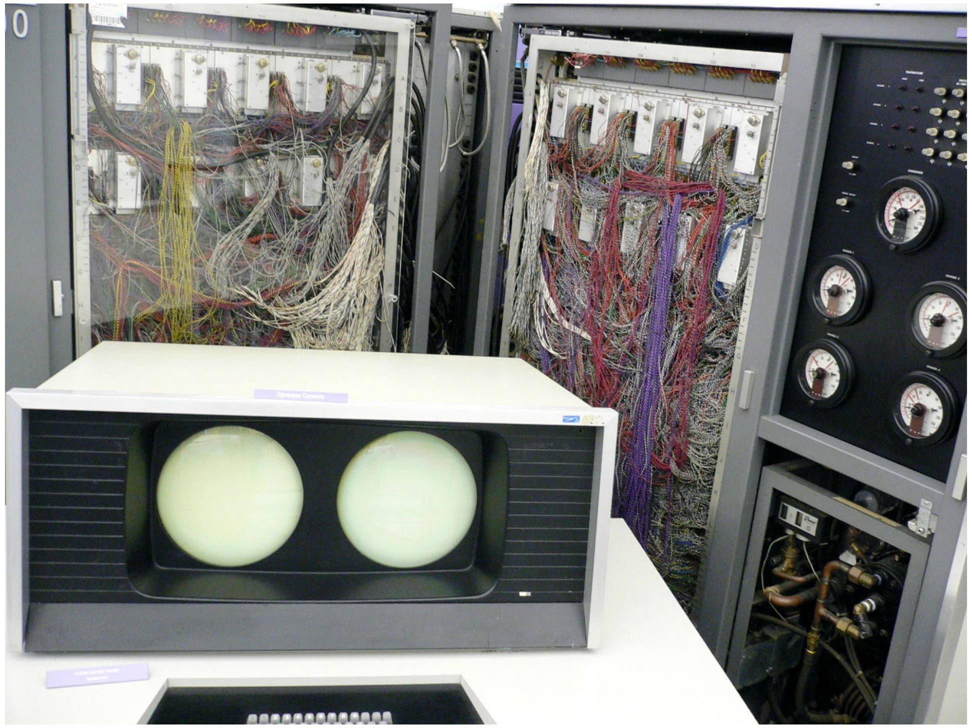
Out-of-Order (OoO, O3) Pipeline

Computer Architecture A Quantitative Approach – Section C7

The CDC 6600 Project [‘1964]

- First implementation of Scoreboard (Out-of-Order)
- 16 separate non-pipelined functional units (7 int, 4 Floating Point (FP), 5 memory)

- **Out-of-order (OoO) execution** is also called **dynamic instruction scheduling**



Steve Jurvetson
CC BY 2.0

The CDC 6600 Project [‘1964]

CDC 6600 Scoreboard

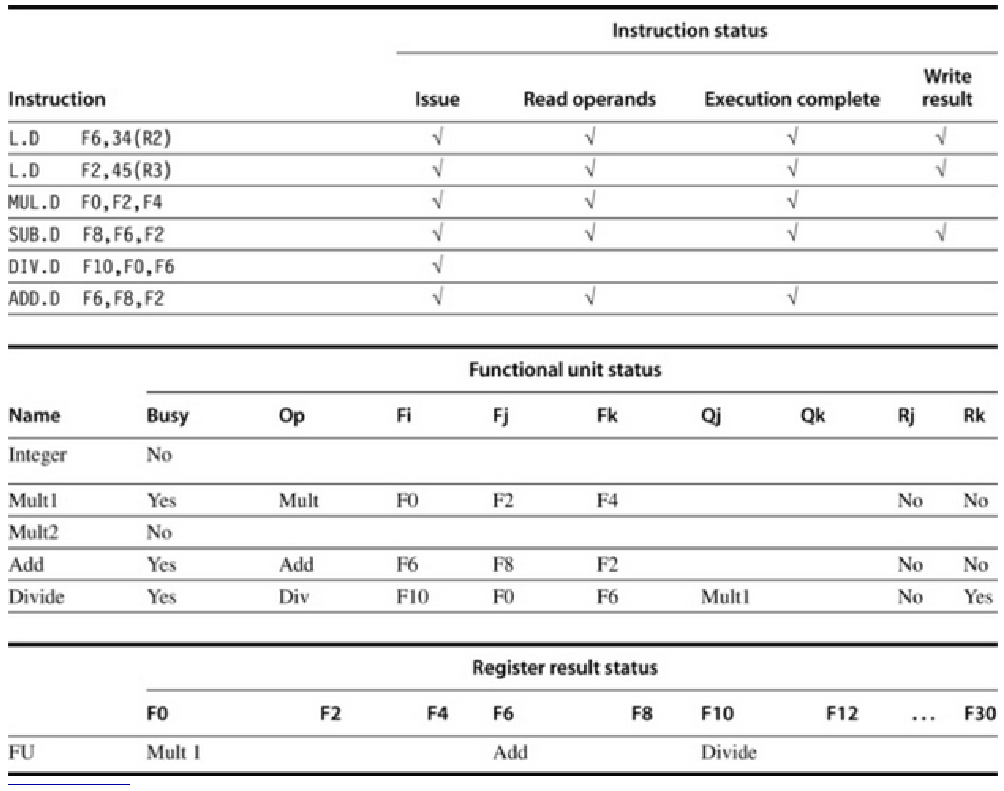
- Three main components

➢Instruction status

➢Functional unit status

➢Register result status

- For an example of use of Scoreboard in CDC 6600 see:
- *Computer Architecture*
  A Quantitative Approach – Section C7



	Instruction status			
Instruction	Issue	Read operands	Execution complete	Write result
`L.D F6,34(R2)`	√	√	√	√
`L.D F2,45(R3)`	√	√	√	√
`MUL.D F0,F2,F4`	√	√	√	
`SUB.D F8,F6,F2`	√	√	√	√
`DIV.D F10,F0,F6`	√			
`ADD.D F6,F8,F2`	√	√	√	

	Functional unit status								
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			No	No
Mult2	No								
Add	Yes	Add	F6	F8	F2			No	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

	Register result status							
	F0	F2	F4	F6	F8	F10	F12	... F30
FU	Mult 1			Add		Divide		

Split of ID Stage

“To implement out-of-order execution, we must split the ID pipe stage into two stages:

- 1. *Issue*—Decode instructions, check for structural hazards.
- 2. *Read operands*—Wait until no data hazards, then read operands.”

- “In a **dynamically scheduled pipeline**, all instructions pass through the issue stage in order (in-order issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus **enter execution out of order**”

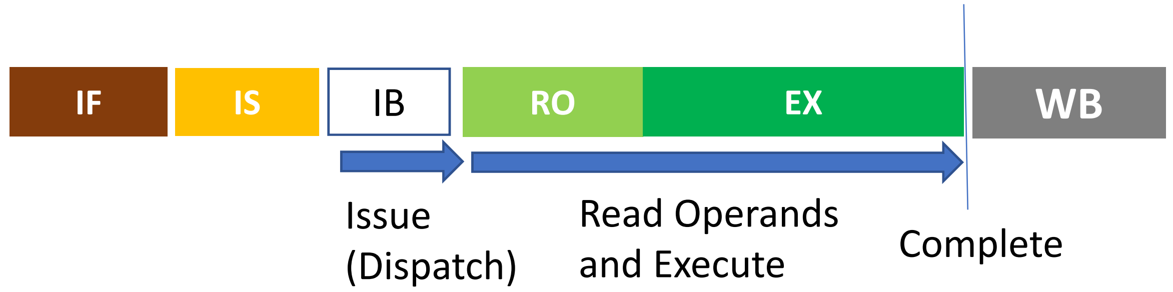
-- *Computer Architecture A Quantitative Approach – 5th Ed. Section C7*

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*

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*

Example OoO Processor: Simple Scoreboard Data Structure

- Simplified CDC-style Scoreboard Data Structure to track execution
- For Scheme 2, One Issue Buffer
- Logical, not HW implementation

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete

Scoreboard (ScB)

FU Status (Ready?)

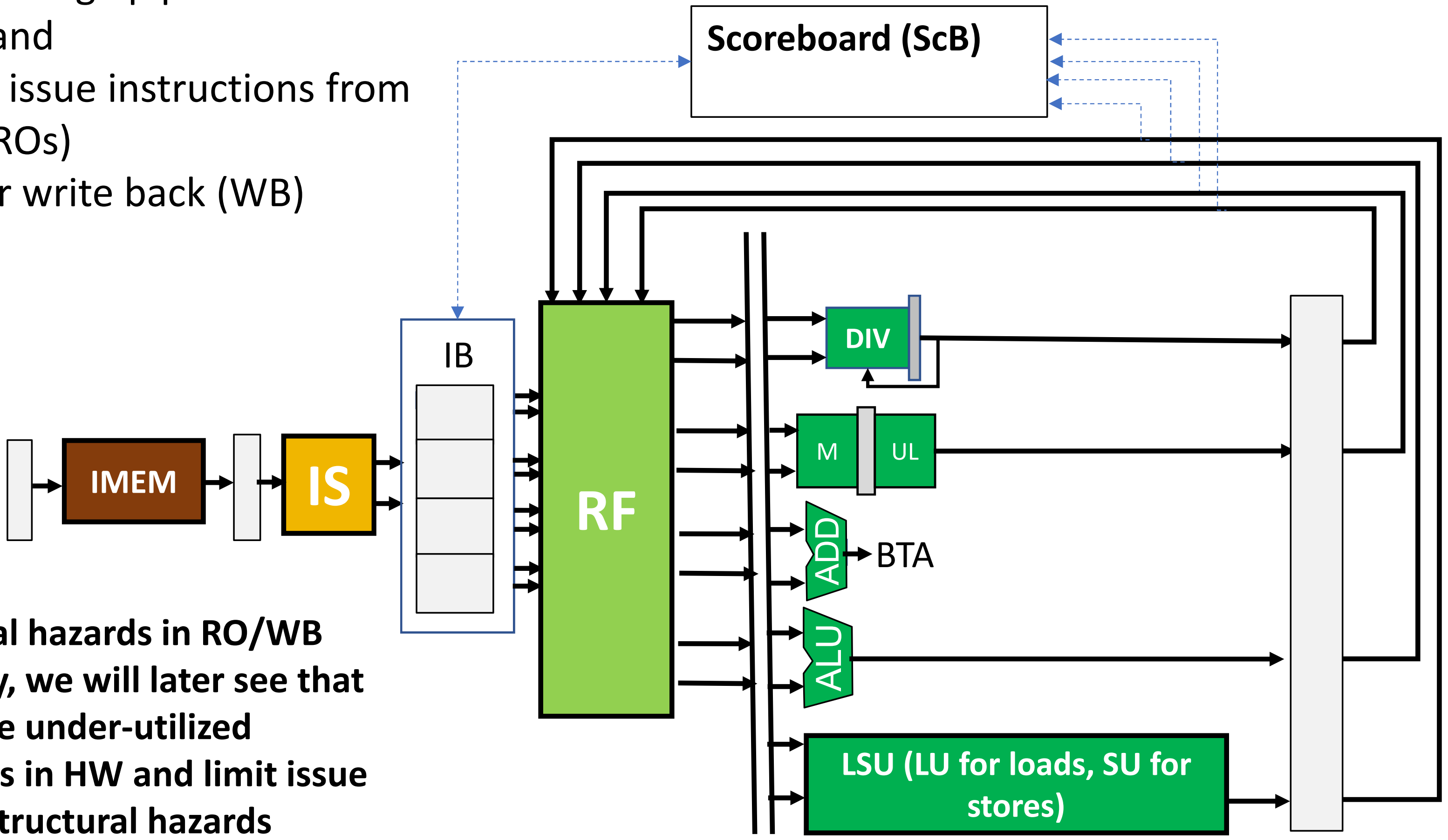
DIV	MUL	ALU	ADD	SU	LU

RO: Instruction read operands (started the computation)
Complete: Instruction finished computation (in last EX stage)

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

Example OoO Processor: FUs in EX stage

For simplicity all FUs have fixed latency:

FU	**Latency**	**Initialization Interval**	
ALU	1	1	
ADD	1	1	
MUL	2	1	Pipelined
DIV	4	4	Serial (fixed latency)
LSU			
LU	2	1	Nonblocking
SU	1	1	Store buffered

- Instruction can only be issued when FU is available.
- SU and LU share same port, cannot be issued together
- We assume instruction cannot be issued to EX same cycle it was added to IB by ID

Example OoO Processor – Pipeline Diagram - Cycle 2

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																		
`LW x13,0(x7)`		IF																		
`DIV x17,x13,x12`																				
`ADDI x18,x12,28`																				
`MUL x19,x12,x18`																				
`MUL x10,x17,x14`																				
`ADD x10,x10,x13`																				
`SW x10,0(x11)`																				
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
➡ LW	x12	x9		8		

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU

Example OoO Processor – Pipeline Diagram - Cycle 3

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	LU																
`LW x13,0(x7)`		IF	IS																
`DIV x17,x13,x12`			IF																
`ADDI x18,x12,28`																			
`MUL x19,x12,x18`																			
`MUL x10,x17,x14`																			
`ADD x10,x10,x13`																			
`SW x10,0(x11)`																			
`LW x10,4(x8)`																			
`ADDI X13,x10,4`																			

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
LW	x12	x9		8	x	
LW	x13	x7		0		

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
					Busy 1

Example OoO Processor – Pipeline Diagram - Cycle 4

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	LU	LU																
`LW x13,0(x7)`		IF	IS	LU																
`DIV x17,x13,x12`			IF	IS																
`ADDI x18,x12,28`				IF																
`MUL x19,x12,x18`																				
`MUL x10,x17,x14`																				
`ADD x10,x10,x13`																				
`SW x10,0(x11)`																				
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
LW	x12	x9		8	x	x
LW	x13	x7		0	x	
DIV	x17	x13	x12			

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
					Busy 2

Example OoO Processor – Pipeline Diagram - Cycle 5

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU															
`DIV x17,x13,x12`			IF	IS	IB	RAW														
`ADDI x18,x12,28`				IF	IS															
`MUL x19,x12,x18`					IF															
`MUL x10,x17,x14`																				
`ADD x10,x10,x13`																				
`SW x10,0(x11)`																				
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
LW	x13	x7		0	x	x
DIV	x17	x13	x12			
ADDI	x18	X12		28		

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
					Busy 1

Example OoO Processor – Pipeline Diagram - Cycle 6

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV														
`ADDI x18,x12,28`				IF	IS	ALU														
`MUL x19,x12,x18`					IF	IS														
`MUL x10,x17,x14`						IF														
`ADD x10,x10,x13`																				
`SW x10,0(x11)`																				
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
DIV	x17	x13	x12		x	
ADDI	x18	x12		28	x	x
MUL	x19	x12	x18			

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
Busy		Busy			

Example OoO Processor – Pipeline Diagram - Cycle 7

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV													
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL													
`MUL x10,x17,x14`						IF	IS													
`ADD x10,x10,x13`							IF													
`SW x10,0(x11)`																				
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
DIV	x17	x13	x12		x	
MUL	x19	x12	x18		x	
MUL	x10	x17	x14			

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
Busy	Busy(s1)				

Example OoO Processor – Pipeline Diagram - Cycle 8

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV												
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL												
`MUL x10,x17,x14`						IF	IS	IB	RAW											
`ADD x10,x10,x13`							IF	stall	WAW											
`SW x10,0(x11)`								stall												
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
DIV	x17	x13	x12		x	
MUL	x19	x12	x18		x	x
MUL	x10	x17	x14			

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
Busy	Busy(s2)				

Example OoO Processor – Pipeline Diagram - Cycle 9

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV											
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	RAW										
`ADD x10,x10,x13`							IF	stall	stall	WAW										
`SW x10,0(x11)`								stall	stall											
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
DIV	x17	x13	x12		x	x
MUL	x10	x17	x14			

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
Busy					

Example OoO Processor – Pipeline Diagram - Cycle 10

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL										
`ADD x10,x10,x13`							IF	stall	stall	stall	WAW									
`SW x10,0(x11)`								stall	stall	stall										
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
MUL	x10	x17	x14		x	

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
	Busy(s1)				

Example OoO Processor – Pipeline Diagram - Cycle 11

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL									
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	WAW								
`SW x10,0(x11)`								stall	stall	stall	stall									
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
MUL	x10	x17	x14		x	x

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
	Busy(s2)				

Example OoO Processor – Pipeline Diagram - Cycle 12

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
➡ `ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS								
`SW x10,0(x11)`								stall	stall	stall	stall	IF								
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
➡ ADD	x10	x10	x13			

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU

Example OoO Processor – Pipeline Diagram - Cycle 13

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS	ALU							
➡ `SW x10,0(x11)`								stall	stall	stall	stall	IF	IS							
`LW x10,4(x8)`													IF							
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
ADD	x10	x10	x13		x	x
➡ SW		x11	x10	0		

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
		busy			

Example OoO Processor – Pipeline Diagram - Cycle 14

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS	ALU	WB						
`SW x10,0(x11)`								stall	stall	stall	stall	IF	IS	SU						
`LW x10,4(x8)`													IF	stall	WAR					
`ADDI X13,x10,4`														stall						

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
SW		x11	x10	0	x	x

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
				Busy 1	

Example OoO Processor – Pipeline Diagram - Cycle 15

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS	ALU	WB						
`SW x10,0(x11)`								stall	stall	stall	stall	IF	IS	SU	SB					
`LW x10,4(x8)`													IF	stall	IS					
`ADDI X13,x10,4`														stall	IF					

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
LW	x10	x18		4		

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU

Example OoO Processor – Pipeline Diagram - Cycle 16

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS	ALU	WB						
`SW x10,0(x11)`								stall	stall	stall	stall	IF	IS	SU	SB					
`LW x10,4(x8)`													IF	stall	IS	LU				
`ADDI X13,x10,4`														stall	IF	IS				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
LW	x10	x18		4	x	
ADDI	x13	x10		4		

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
					Busy 1

Example OoO Processor – Pipeline Diagram - Cycle 17

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS	ALU	WB						
`SW x10,0(x11)`								stall	stall	stall	stall	IF	IS	SU	SB					
`LW x10,4(x8)`													IF	stall	IS	LU	LU			
`ADDI X13,x10,4`														stall	IF	IS	IB	RAW		

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
LW	x10	x18		4	x	x
ADDI	x13	x10		4		

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
					Busy 1

Example OoO Processor – Pipeline Diagram - Cycle 18

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS	ALU	WB						
`SW x10,0(x11)`								stall	stall	stall	stall	IF	IS	SU	SB					
`LW x10,4(x8)`													IF	stall	IS	LU	LU	WB		
`ADDI X13,x10,4`														stall	IF	IS	IB	ALU		

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
ADDI	x13	x10		4	x	x

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
		busy			

Example OoO Processor – Pipeline Diagram - Cycle 19

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	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS	ALU	WB						
`SW x10,0(x11)`								stall	stall	stall	stall	IF	IS	SU	SB					
`LW x10,4(x8)`													IF	stall	IS	LU	LU	WB		
`ADDI x13,x10,4`														stall	IF	IS	IB	ALU	WB	

10 instructions
3 cycles ramp-up (4-stage pipeline)
Total 19 cycles -3 cycles = 16 cycles

CPI = 1,6

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete

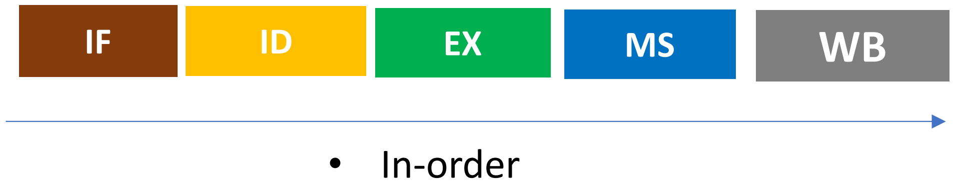
FU Status (Ready?)

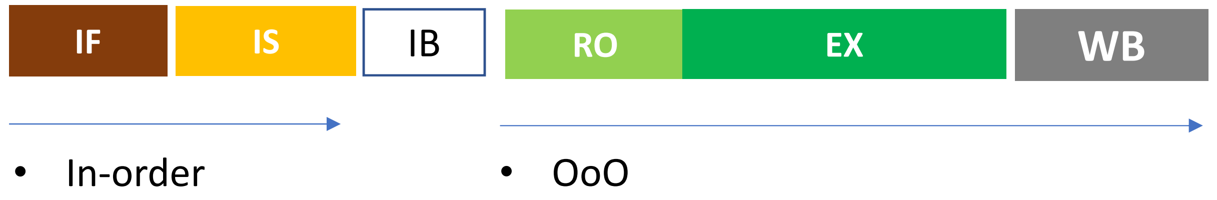
DIV	MUL	ALU	ADD	SU	LU

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.





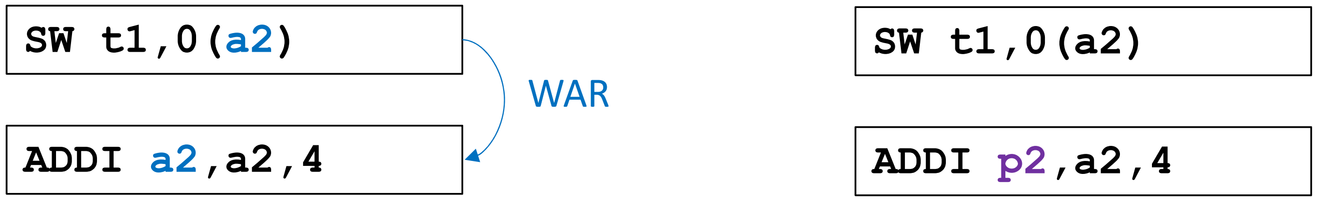
Register Renaming

Out-of-Order Limitations

- WAW and WAR limit further reordering
  - Not real dependencies
  - Artificially added: limitation of registers
- Problem with limited registers
  - Number of registers limited by ISA
  - Compiler optimizations limited
  - Especially with different execution paths
- Approach: CPU solves problem by register renaming

Register Renaming

- Approach: Rename to microarchitecture register names
  - More microarchitecture registers than logical ISA registers
  - Entirely eliminates WAR and WAW hazards
  - Not visible to the outside world



```
SW t1,0(a2)

ADDI a2,a2,4
```

```
SW t1,0(a2)

ADDI p2,a2,4
```

- Introduced by Robert Tomasulo (1967)
  - Reservation stations (FU-specific IBs) before FUs store instructions and reg. names
  - Tomasulo Algorithm: *Computer Architecture A Quantitative Approach 5th Ed. – Chapter 3*

Example: Register Renaming removes WAW, RAW stalls

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`LW x12,8(x9)`	IF	IS	LU	LU	WB									
`LW x13,0(x7)`		IF	IS	LU	LU	WB								
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB				
`ADDI x18,x12,28`				IF	IS	ALU	WB							
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB					
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB		
`ADD p10,x10,x13`							IF	IS	IB	IB	IB	ALU	WB	
`SW p10,0(x11)`								IF	IS	IB	IB	IB	SU	SB
`LW p11,4(x8)`									IF	IS	LU	LU	WB	
`ADDI x13,p11,4`										IF	IS	IB	ALU	WB

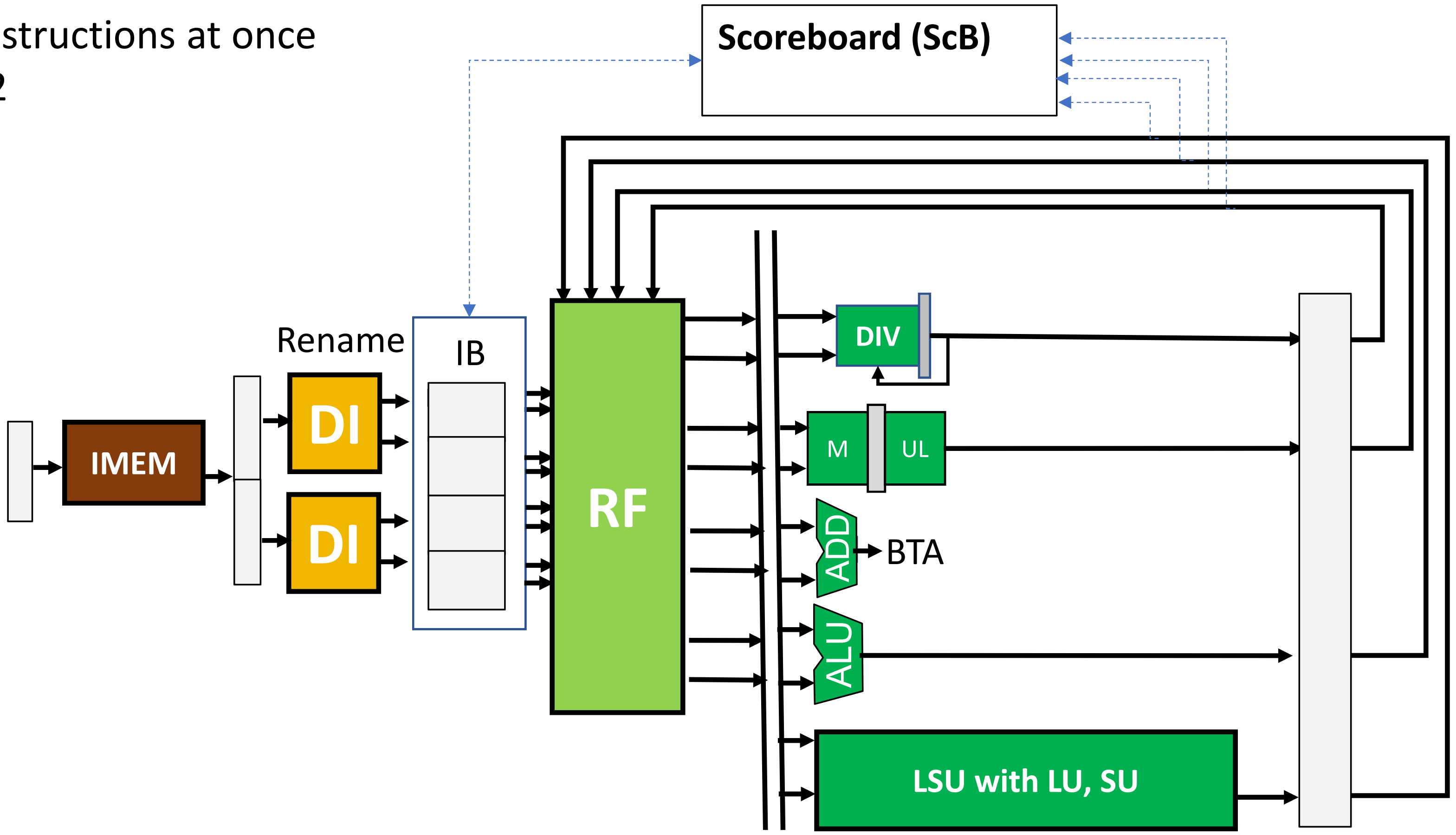
CPI = 1,1

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 ADD caused 4 stall cycles that are gone now but the RAW still requires it to wait.

BUT: Instructions behind ADD can execute earlier in OoO fashion.

Simple Superscalar (Scoreboard) – Dual Fetch and Decode

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



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

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB										
`LW x13,0(x7)`	IF	IS	IB	LU	LU	WB									
`DIV x17,x13,x12`		IF	IS	IB	IB	DIV	DIV	DIV	DIV	WB					
`ADDI x18,x12,28`		IF	IS	ALU	WB										
`MUL x19,x12,x18`			IF	IS	MUL	MUL	WB								
`MUL x10,x17,x14`			IF	IS	IB	IB	IB	IB	IB	MUL	MUL	WB			
`ADD p10,x10,x13`				IF	IS	IB	IB	IB	IB	IB	IB	ALU	WB		
`SW p10,0(x11)`				IF	IS	IB	IB	IB	IB	IB	IB	IB	SU	SB	
`LW p11,4(x8)`					IF	IS	LU	LU	WB						
`ADDI x13,p11,4`					IF	IS	IB	IB	ALU	WB					
…						IF	Stall	Stall	…						
…						IF	Stall	Stall	…						

Issue buffer full

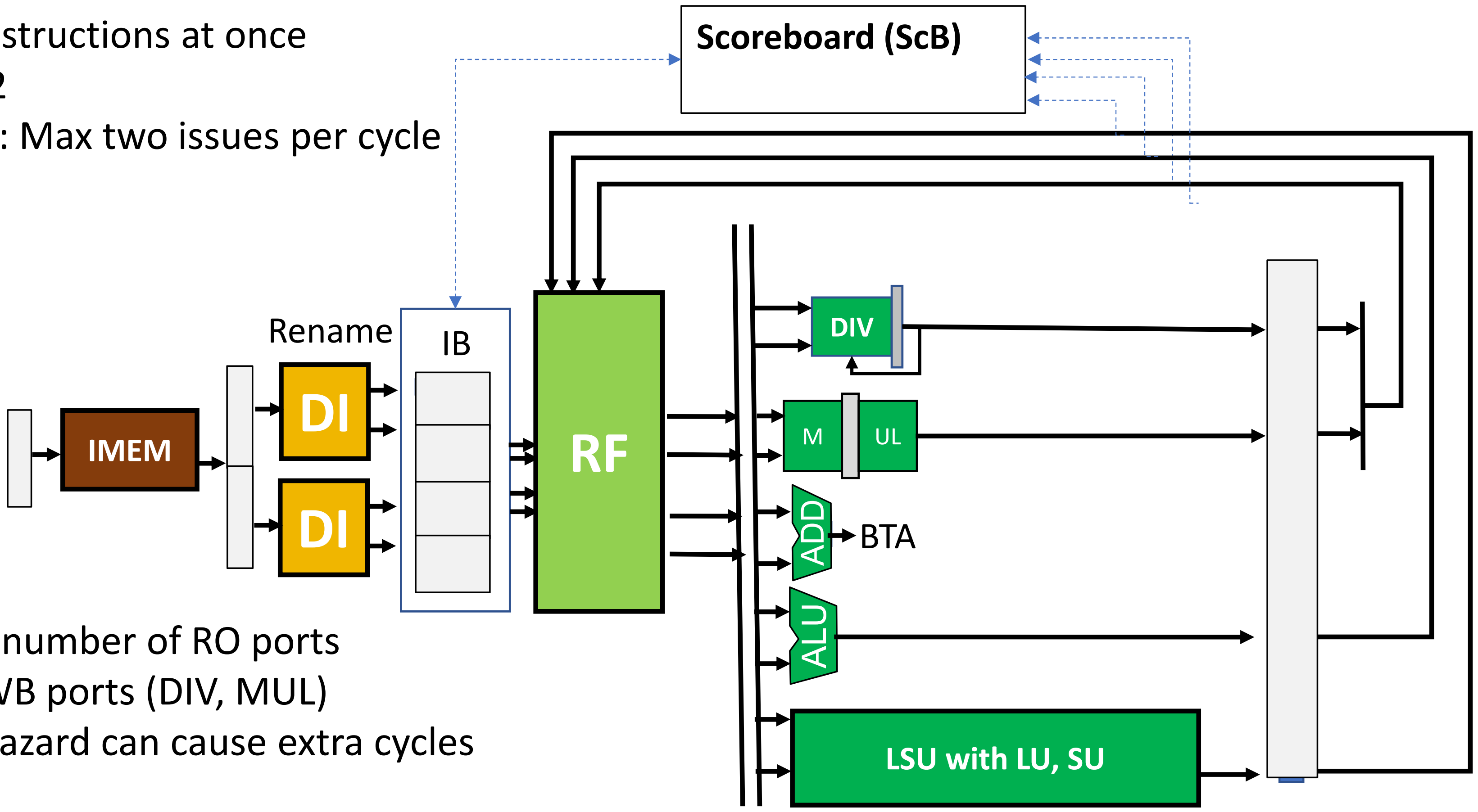
Due to this chain of RAW dependencies IPC <=1, Otherwise IPC could go above 1

CPI = 1,0

Fetching more instructions assures the issue buffer is always filled.
BUT: Instruction Level Parallelism can limit instructions executing in parallel

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

Wide instruction fetch can
fetch two instructions at once
Ideal IPC = 2
Reduce HW: Max two issues per cycle



Reduce the number of RO ports
and share WB ports (DIV, MUL)
Structural hazard can cause extra cycles

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	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB										
`LW x13,0(x7)`	IF	IS	IB	LU	LU	WB									
`DIV x17,x13,x12`		IF	IS	IB	IB	DIV	DIV	DIV	DIV	WB					
`ADDI x18,x12,28`		IF	IS	ALU	WB										
`MUL x19,x12,x18`			IF	IS	MUL	MUL	WB								
`MUL x10,x17,x14`			IF	IS	IB	IB	IB	IB	IB	MUL	MUL	WB			
`ADD p10,x10,x13`				IF	IS	IB	IB	IB	IB	IB	IB	ALU	WB		
`SW p10,0(x11)`				IF	IS	IB	IB	IB	IB	IB	IB	IB	SU	SB	
`LW p11,4(x8)`					IF	IS	LU	LU	WB						
`ADDI x13,p11,4`					IF	IS	IB	IB	ALU	WB					
…						IF	Stall	Stall	…						
…						IF	Stall	Stall	…						

No RO structural hazard as first LW was issued a cycle earlier

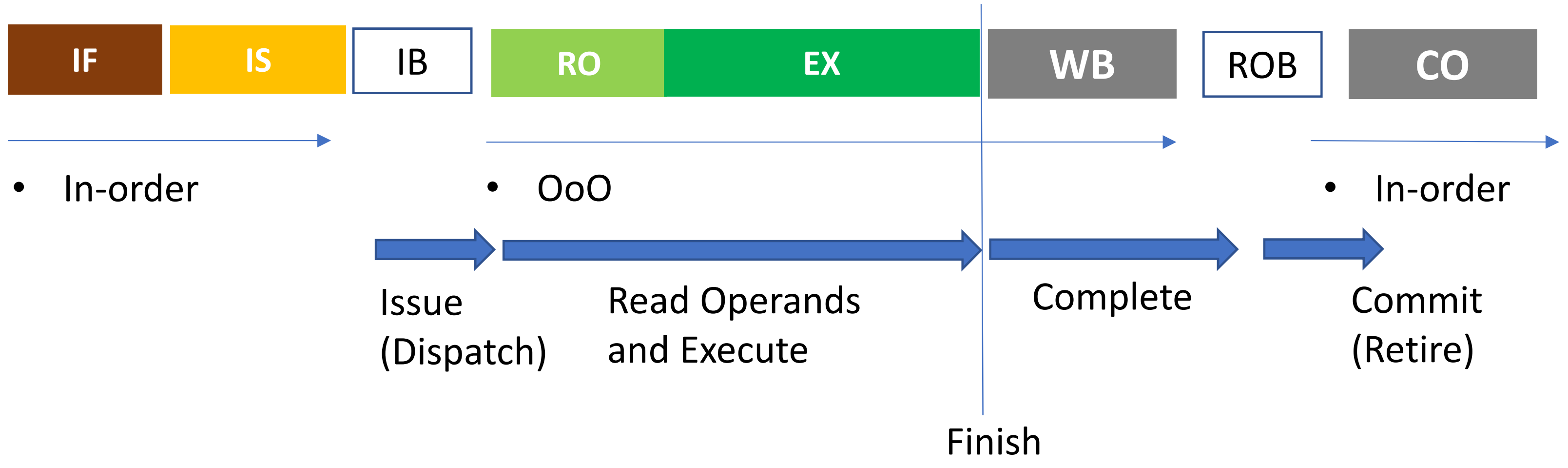
CPI = 1,0

We still observe no structural hazards. ILP limits instruction issue below 2.

Reorder Buffer (ROB)

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)

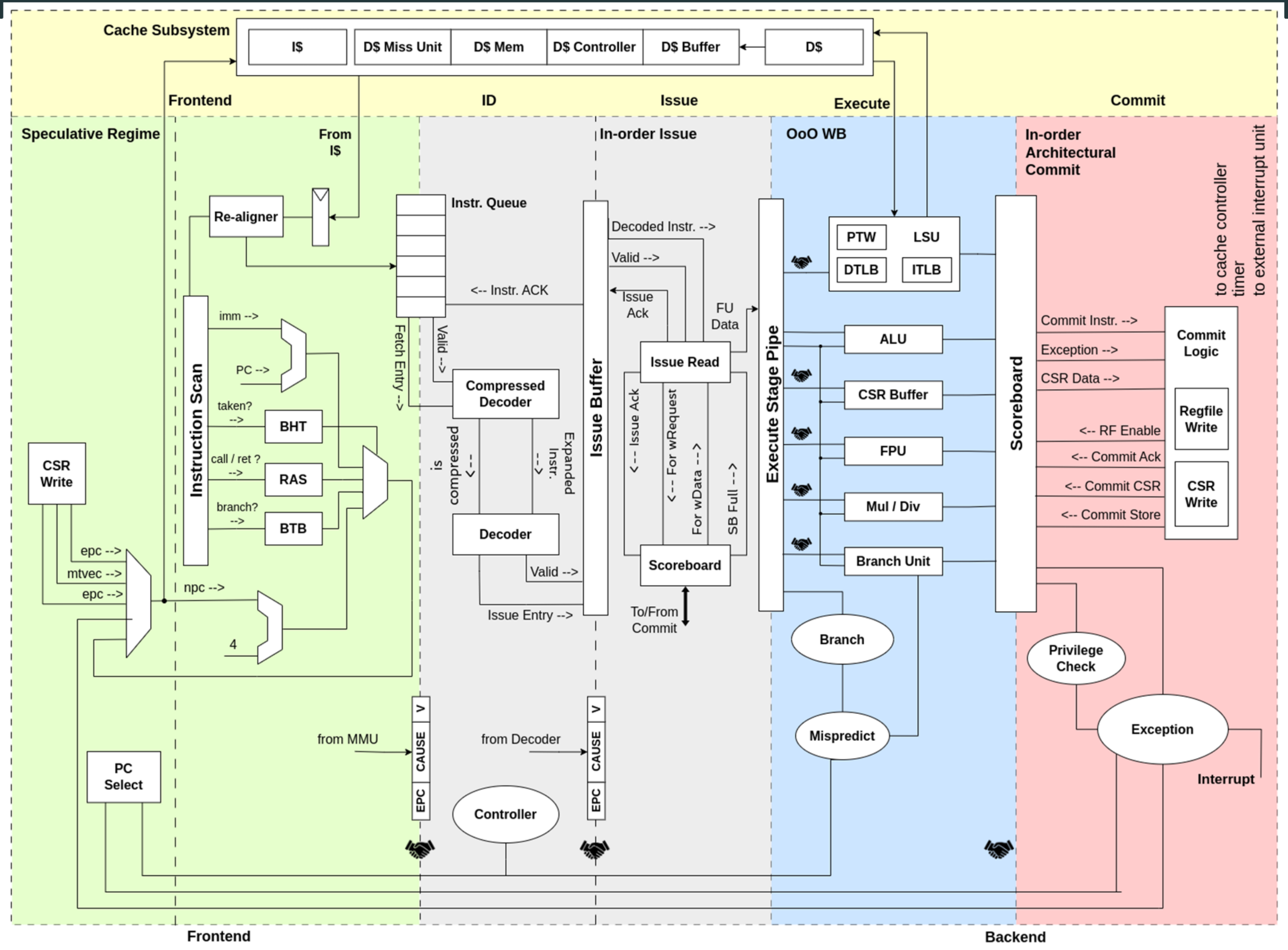


Optional, not relevant for exam

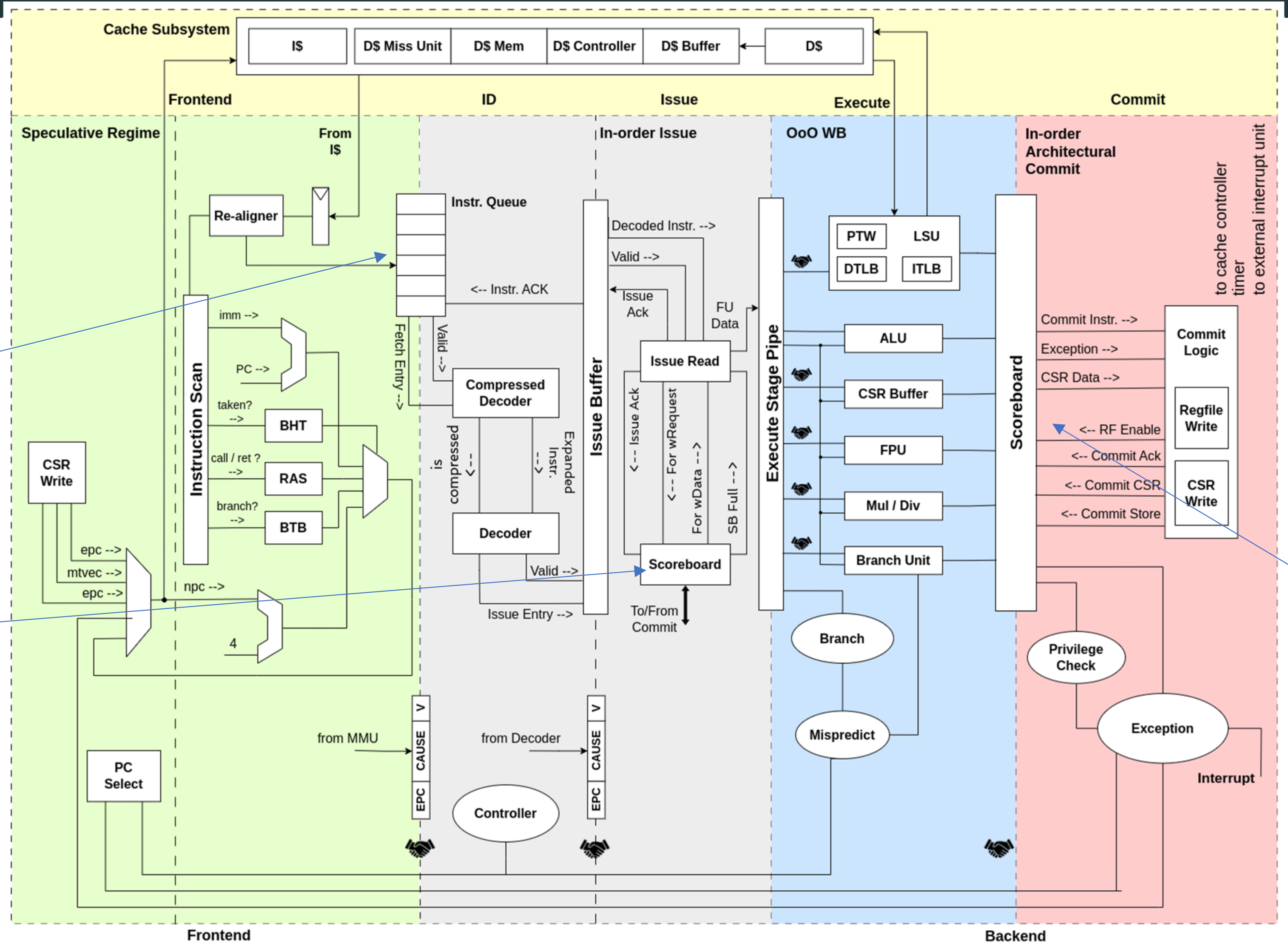
A Look at a Real Processor

CVA6

CVA6 Pipeline Diagram: https://github.com/openhwgroup/cva6



CVA6 Pipeline Diagram: https://github.com/openhwgroup/cva6



Fetch buffer between IF and ID

Scoreboard

In-order commit. Sorts the OoO WB

Optional, not relevant for exam

Pipeline Support for Precise Traps

Challenge with OoO Pipelines and Exceptions

- Some instructions can cause exceptions
  - Memory fault on load/store
  - Before entering exception handling all previous instructions should have committed (done their write back)
  - No instruction after the one that caused the exception should have committed (done their write back)

	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)

Implementing Precise Exceptions in OoO Pipelines

- For Precise Exception:
  - Before entering exception handling all previous instructions should have committed
  - All previous stores should have written to memory or SB should continue to write them to memory

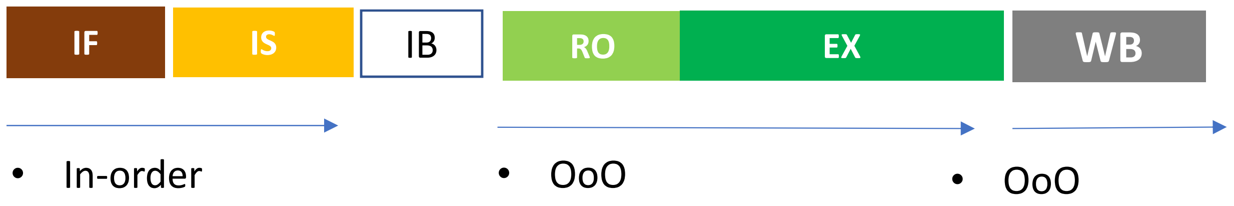
  - No instruction after the instruction that caused the exception should have committed, instead they should be deleted (killed)
  - No store after the instruction that caused the exception should have written to memory from the SB, instead they should be deleted (killed) from the SB

- Scoreboard approach did not support precise exceptions
- Different approaches to implement precise exceptions: e.g. Reorder-Buffer (ROB) sorts all WB commits and makes sure store buffer only sends committed stores to memory

Summary

Where we are

- Four-Stage Superscalar Out-of-order Processor Pipeline
  - Exploit Instruction Level Parallelism to hide extra cycles of multi-cycle FUs.
  - Scoreboard to track instruction dependencies



- Four Stage
- Out-of-order (OoO) pipeline
- Superscalar pipeline (Multi-Issue)

- Upcoming Lecture: More on Multi-Issue Processors (targeting IPC > 1)

Thank you for your attention!

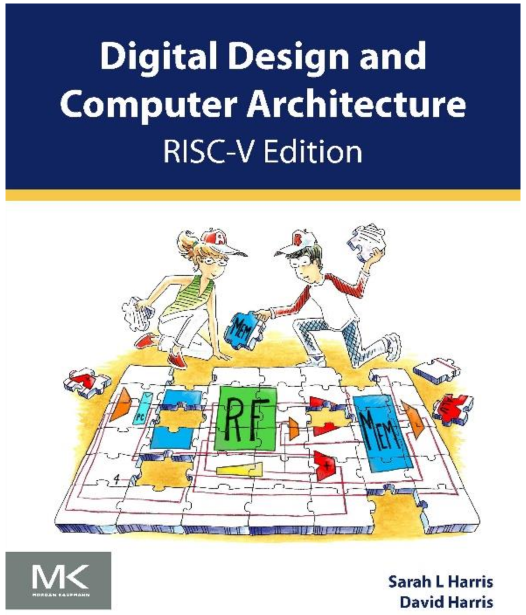
Computer Systems

Advanced Processor Pipelines 3

Daniel Mueller-Gritschneder

25.04.2024

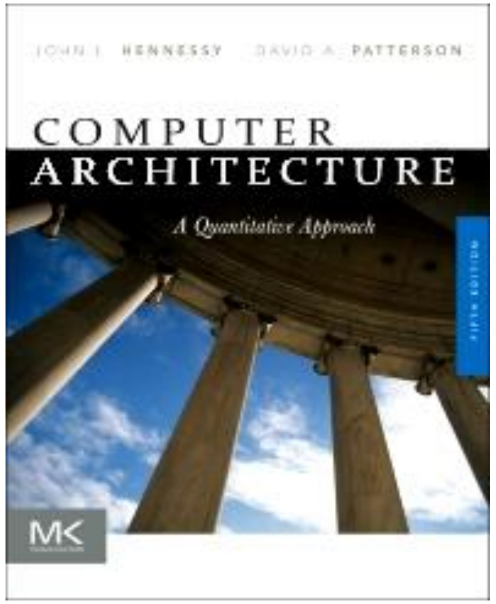
Sources



This book covers the basics of how to design a simple in-order scalar processor pipeline in detail in hardware.

- Literature: „Digital Design and Computer Architecture: RISC-V Edition“, by Sarah L. Harris and David Harris
  - https://shop.elsevier.com/books/digital-design-and-computer-architecture-risc-v-edition/harris/978-0-12-820064-3
  - https://pages.hmc.edu/harris/ddca/ddcarv.html (Includes resources for students!)
  - They also provide slideshows – the basis for ours! You can investigate extended version at their website.
- Available at TU's library: https://catalogplus.tuwien.at/permalink/f/qknpf/UTW_alma21139903990003336

Sources



So-called application processors have many additional features:
Branch prediction, Out of order execute, Scoreboard, Superpipelining, Multi-issue, Superscalar, VLIW, Multi-threading, …

Disclaimer: The book provides advanced concepts from real complex processor designs. We only study the concepts at a high level. For simplicity, the used pipeline models in this lecture are reduced strongly in complexity.

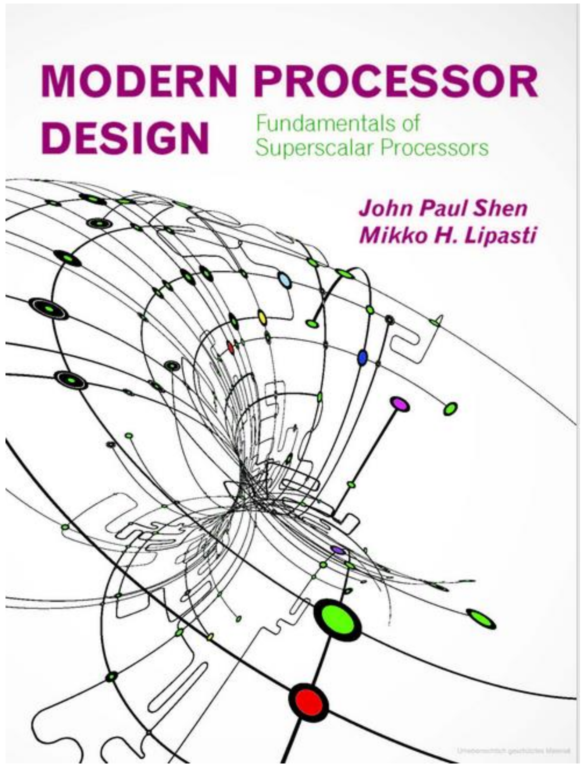
But: We will have a look at some current RISC-V processor designs

Literature: „**Computer Architecture A Quantitative Approach”** 5th Edition - September 16, 2011
Authors: John L. Hennessy, David A. Patterson eBook ISBN: 9780123838735

- https://shop.elsevier.com/books/computer-architecture/hennessy/978-0-12-383872-8

- Available at TU's library:
  https://catalogplus.tuwien.at/permalink/f/8agg25/TN_cdi_askewsholts_vlebooks_9780123838735

Sources



Advanced concepts for superscalar.

Literature: **Shen & Lipasti : Modern Processor Design (2005)**

Lecture slides available: https://pharm.ece.wisc.edu/mikko/

Content

- Processors‘ Performance
- Superpipelining
- VLIW
- Superscalar
- Multi-threading

Optional, not relevant for exam

- A look at a real RISC-V processor: BOOM, A15

Processors’ Performance

Superpipelining and Multi-Issue

Processors‘ Performance

- Recap of Last lecture: Superscalar processor reached CPI=1

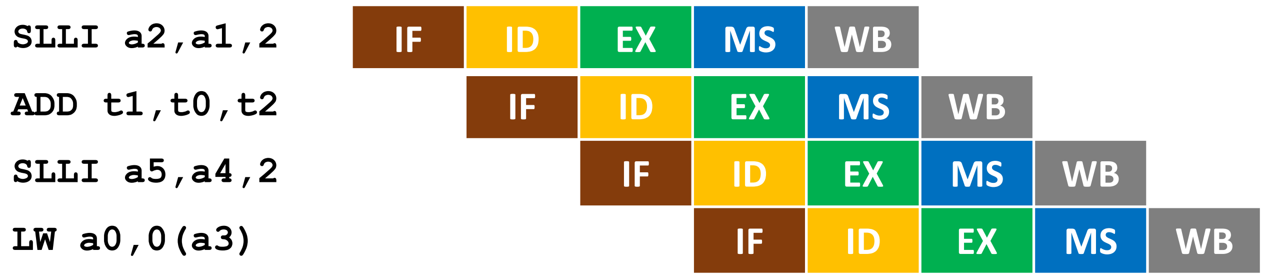
Performance of a processor (IC is instruction count):

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

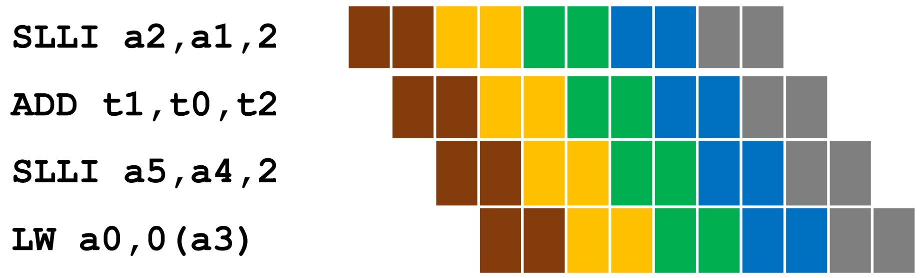
- Superpipelining aims at increasing performance via frequency
- Superscalar, VLIW aims at increasing performance via IPC
- Compiler optimization can improve instruction count (IC) and IPC

Superpipelining and Multi-Issue

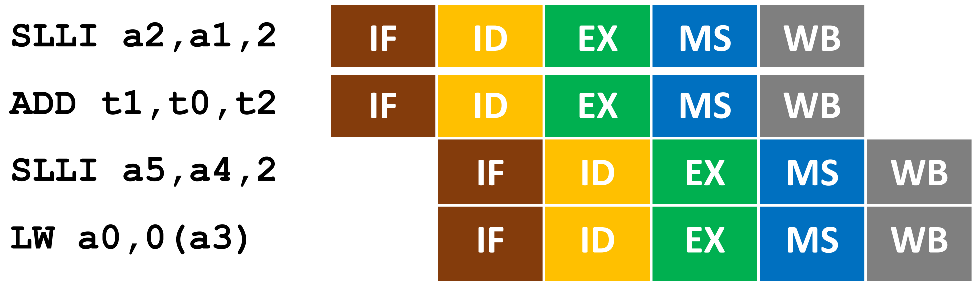
- Scalar five-stage pipeline



- Superpipelining concept:



Multi-Issue concept:

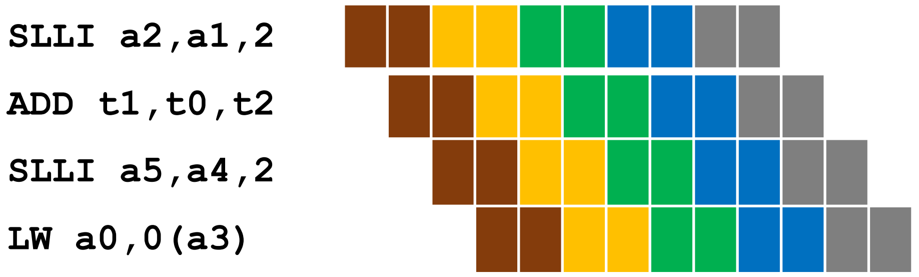


- **Superpipelining** aims at higher clock frequency by increasing number of pipeline stages!
- **Multi-Issue processors** enable CPI < 1 (IPC > 1) by fetching, decoding and executing multiple instructions in parallel

Superpipelining

Superpipelining

- **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)

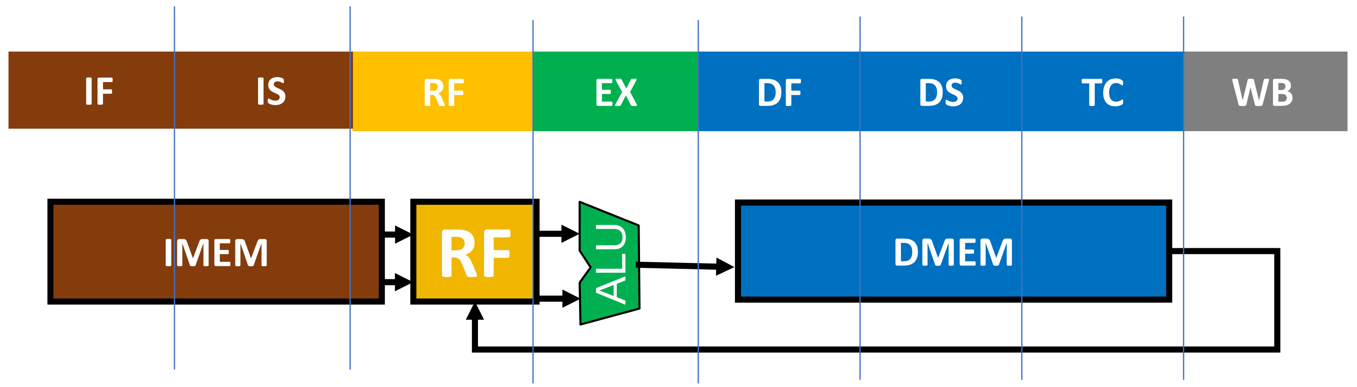


```
SLLI a2,a1,2
ADD t1,t0,t2
SLLI a5,a4,2
LW a0,0(a3)
```

- 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)

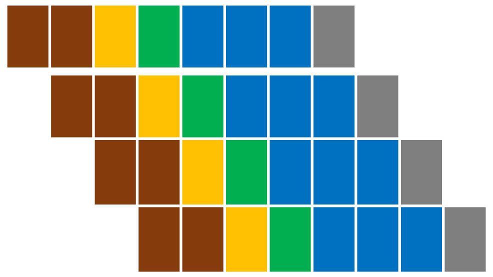


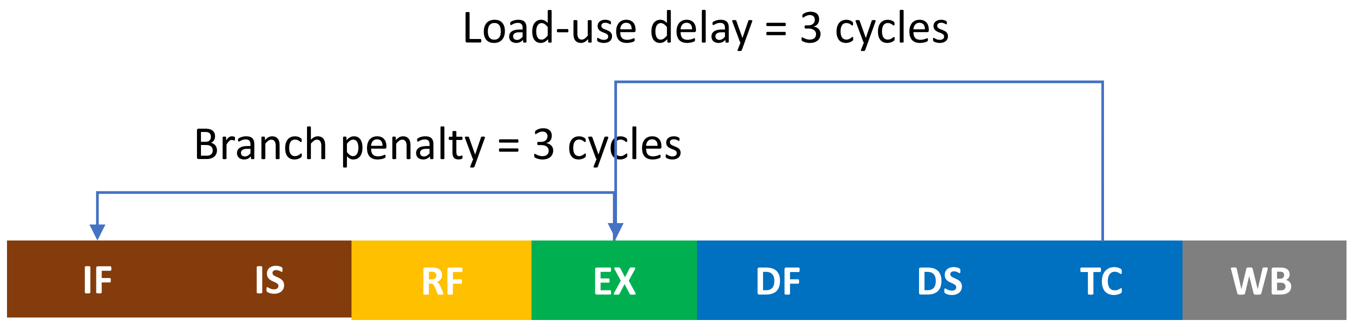
- **IF** — First half of instruction fetch;
- **IS** — Second half of instruction fetch, complete instruction cache access.
- **RF** — Instruction decode and register fetch
- **EX** — Execution, which includes effective address calculation, ALU operation, and branch-target computation and condition evaluation.
- **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*

Example: MIPS R4000

- Execution Scheme





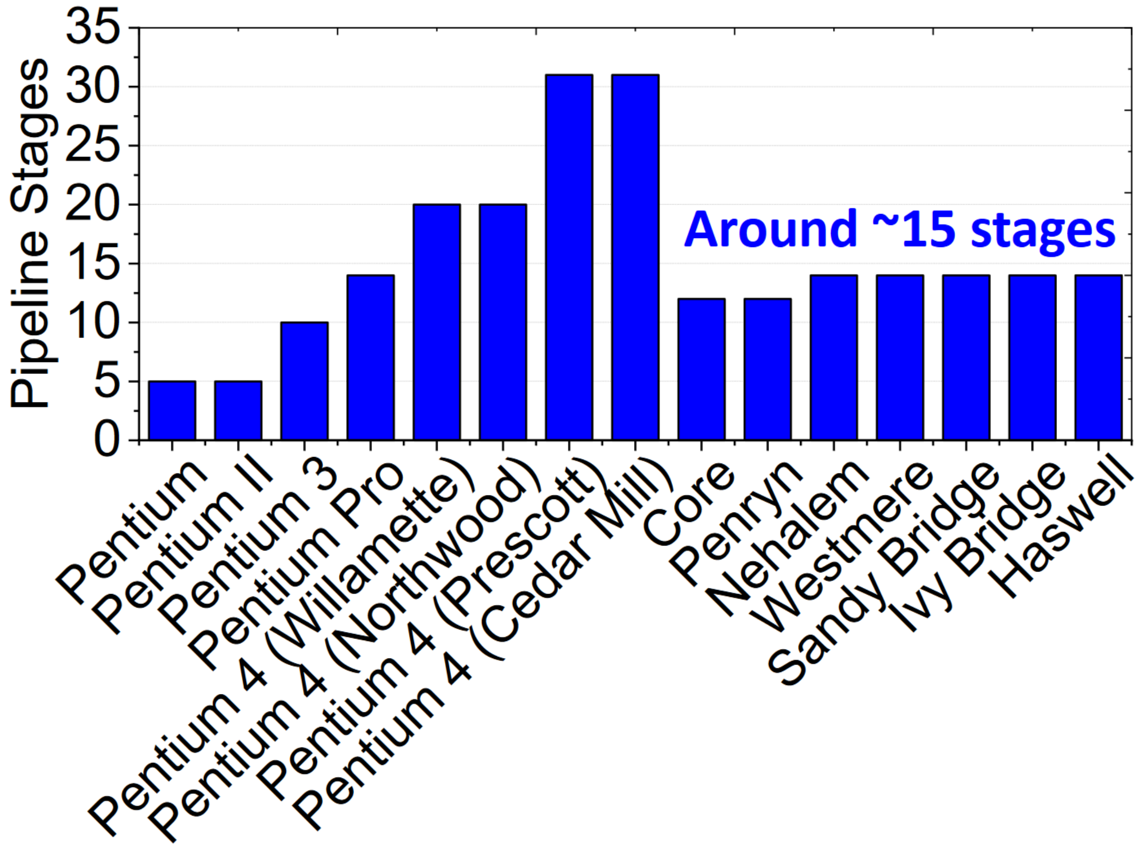
- Instruction dependences have higher penalties (due to deeper pipeline)
    - Branch decision later available -> prediction even more important as more instructions must be flushed (In MIPS R4000: branch computed in EX stage -> 3 cycles branch penalty)
    - Forwarding can‘t remove all stall cycles for RAW dependencies (e.g. Load-use data needs three cycles to become available).

Limits of Superpipelining

- Number of pipeline stages:
  Desktop CPUs: 12-20 stages.
- Embedded CPUs: all from 1-20 stages.

➢ Original Source “Runtime Aware Architectures”, Mateo Valero, HiPEAC CSW 2014,
taken from Lecture Myoungsoo Jung (Slide 6):
http://camelab.org/uploads/Main/lecture06-istruction-paralllel-processing.pdf

➢ See for example
https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures
for a list of the number of pipeline stages for recent Intel‘s processors



Multi-Issue

Static and Dynamic 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

Speculation

- “Guess” what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - **Speculate on branch outcome**, execute instructions after branch
    - Roll back, if path taken is different
  - **Speculate on store** that precedes load does not refer to same address
    - We can execute the 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
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can **look ahead** for instructions to execute
  - Buffer results until it determines they are actually needed (written to the registers or memory)
  - Flush buffers on incorrect speculation

Very Long Instruction Word (VLIW)

Static Multi-Issue

Static Multiple Issue

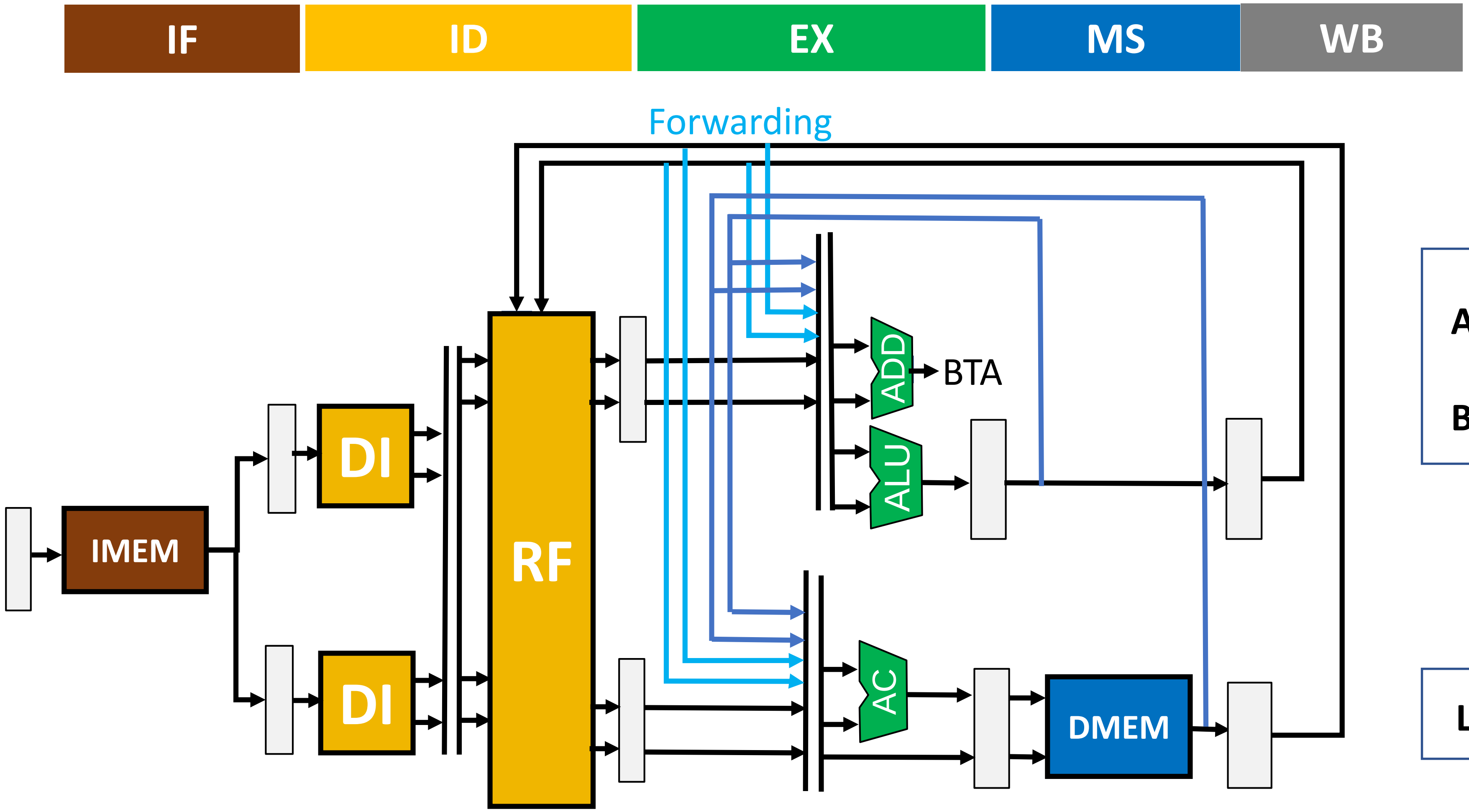
- Compiler groups instructions into “issue packets” (sometimes also called bundles)
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- **Think of an issue packet as a very long instruction**
  - Specifies multiple concurrent operations
  - $\Rightarrow$ **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
  - Possibly some dependencies between packets
  - 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)



Slot 1

ALU
Branch Comp.

Slot 2

Load/Store

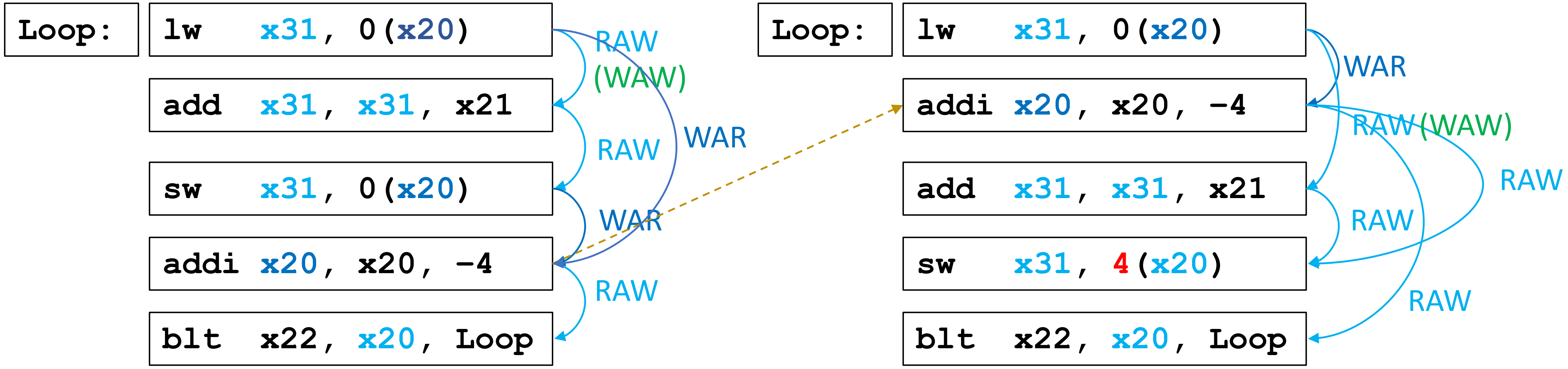
Hazards in the Dual-Issue RISC-V

- More instructions executing in parallel
- **RAW data hazard**
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result in load/store in same packet

    -
      ```
      add x10, x0, x1
      lw  x2, 0(x10)
      ```
    - Split into two packets, effectively a stall
- **Load-use hazard**
  - Still one cycle use latency, but now two instructions
- **More aggressive scheduling required**

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
```



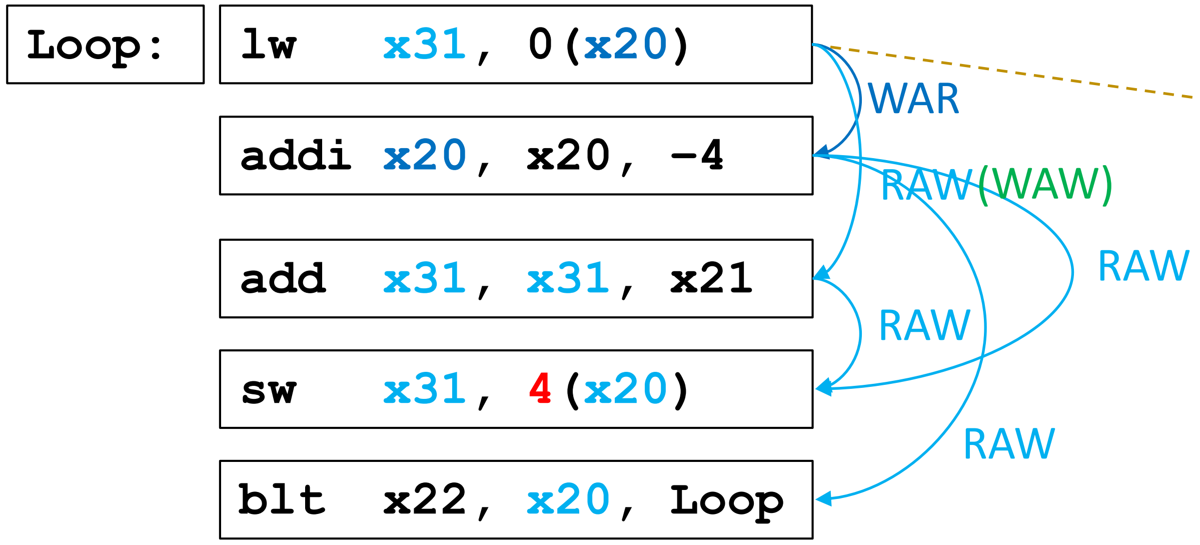
```
Loop: lw   x31, 0(x20)
      add  x31, x31, x21
      sw   x31, 0(x20)
      addi x20, x20, -4
      blt  x22, x20, Loop
```

```
Loop: lw   x31, 0(x20)
      addi x20, x20, -4
      add  x31, x31, x21
      sw   x31, 4(x20)
      blt  x22, x20, Loop
```

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

Scheduling Example

- **Schedule this for dual-issue RISC-V**

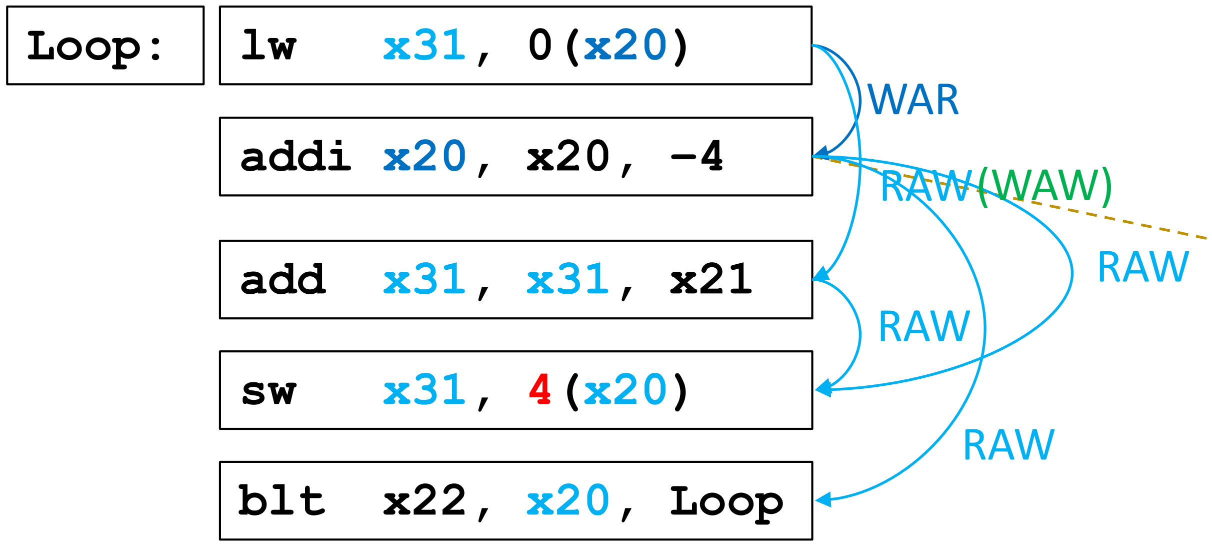


```
Loop:  lw    x31, 0(x20)
       addi  x20, x20, -4
       add   x31, x31, x21
       sw    x31, 4(x20)
       blt   x22, x20, Loop
```

	Slot1 : ALU/BRANCH	Slot 2: Load/store
`Loop:`		`lw x31, 0(x20)`

Scheduling Example

- **Schedule this for dual-issue RISC-V**



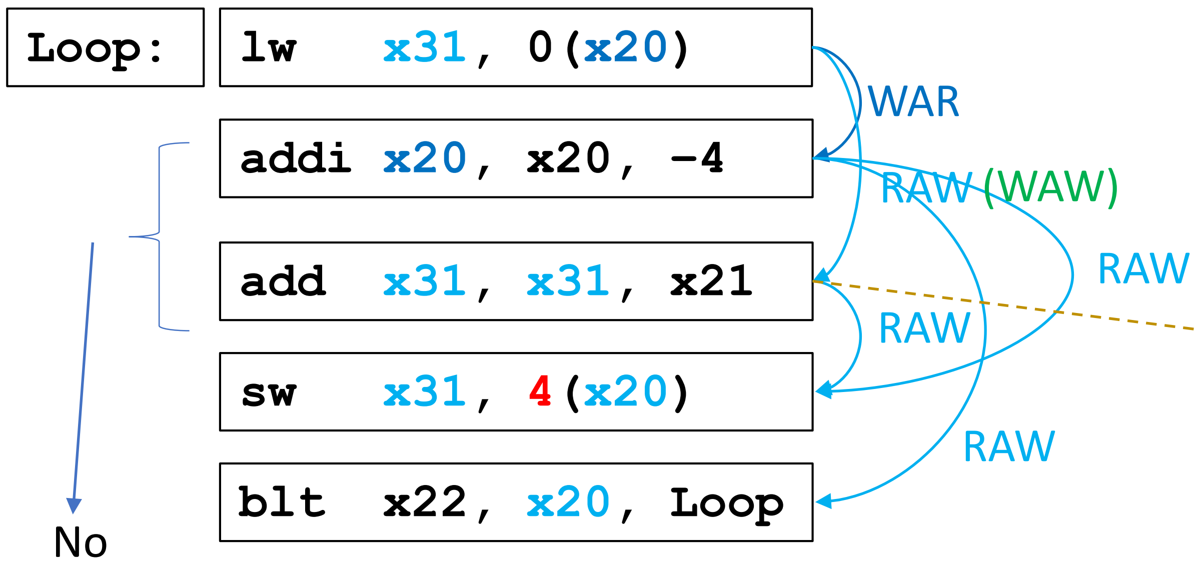
```
Loop:  lw    x31, 0(x20)
       addi  x20, x20, -4
       add   x31, x31, x21
       sw    x31, 4(x20)
       blt   x22, x20, Loop
```

WAR hazard to lw

	Slot1 : ALU/BRANCH	Slot 2: Load/store
`Loop:`		`lw x31, 0(x20)`
	`addi x20, x20, -4`	

Scheduling Example

- **Schedule this for dual-issue RISC-V**



```
Loop:  lw    x31, 0(x20)
       addi  x20, x20, -4
       add   x31, x31, x21
       sw    x31, 4(x20)
       blt   x22, x20, Loop
```

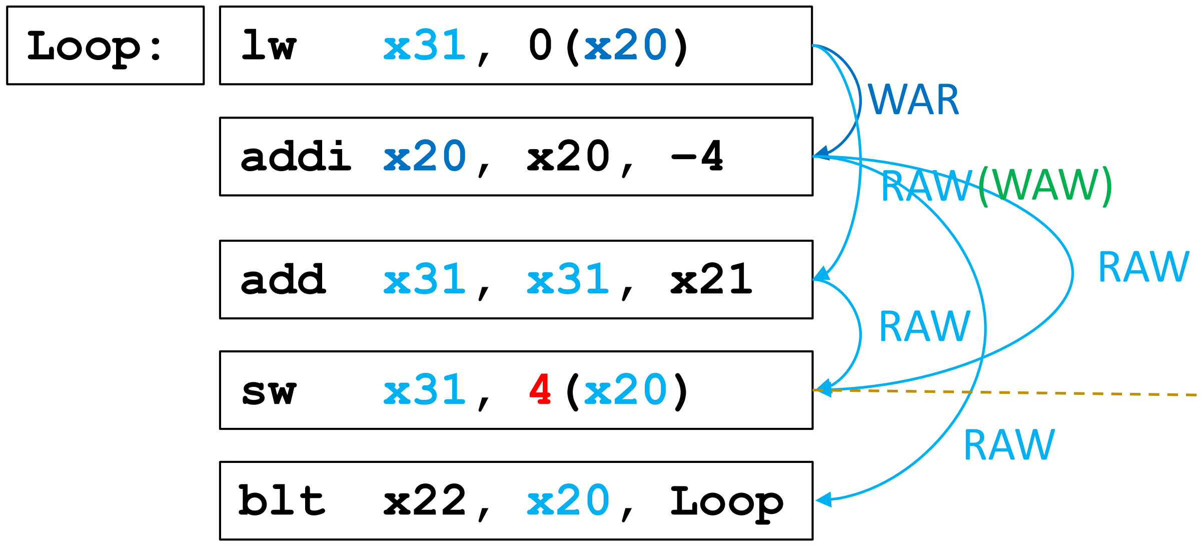
No
dependencies but go into same slot

RAW + (WAW) hazard to lw
Slot 1 already occupied in pack 2

	Slot1 : ALU/BRANCH	Slot 2: Load/store
`Loop:`		`lw    x31, 0(x20)`
	`addi  x20, x20, -4`	
	`add   x31, x31, x21`	

Scheduling Example

- **Schedule this for dual-issue RISC-V**



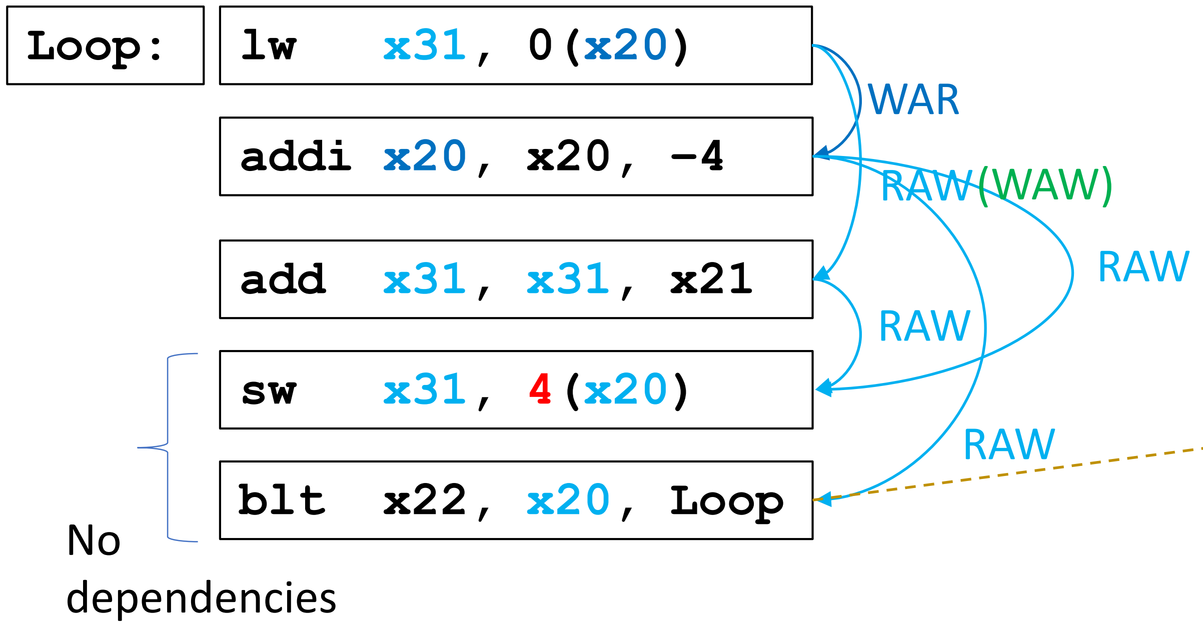
```
Loop:  lw    x31, 0(x20)
       addi  x20, x20, -4
       add   x31, x31, x21
       sw    x31, 4(x20)
       blt   x22, x20, Loop
```

RAW to add

	Slot1 : ALU/BRANCH	Slot 2: Load/store
`Loop:`		`lw    x31, 0(x20)`
	`addi x20, x20, -4`	
	`add   x31, x31, x21`	
		`sw    x31, 4(x20)`

Scheduling Example

- **Schedule this for dual-issue RISC-V**

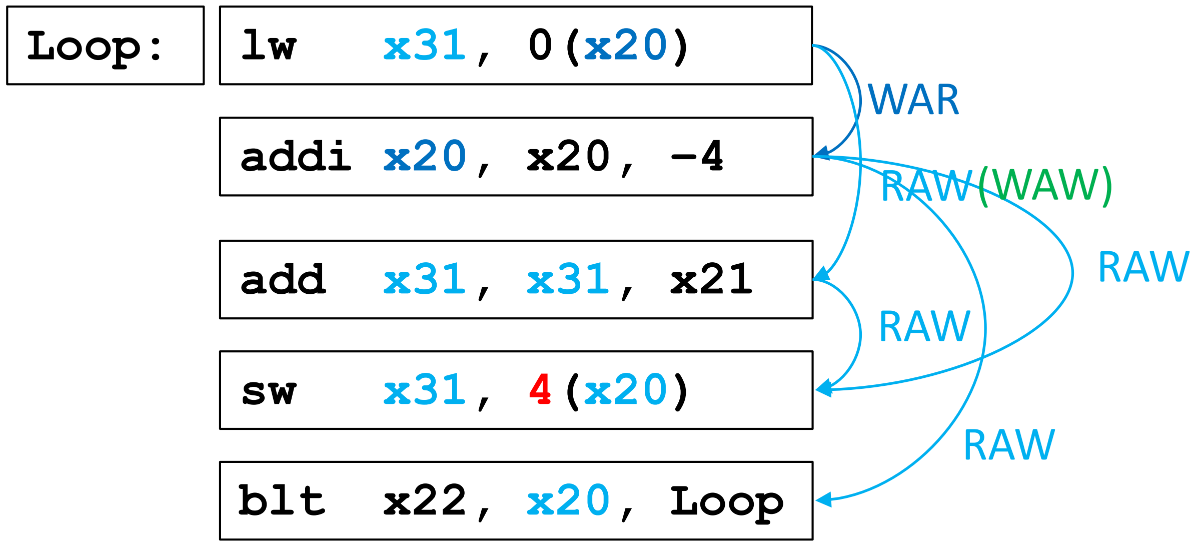


```
Loop:  lw    x31, 0(x20)
       addi  x20, x20, -4
       add   x31, x31, x21
       sw    x31, 4(x20)
       blt   x22, x20, Loop
```

	Slot1 : ALU/BRANCH	Slot 2: Load/store
`Loop:`		`lw x31, 0(x20)`
	`addi x20, x20, -4`	
	`add x31, x31, x21`	
	`blt x22, x20, Loop`	`sw x31, 4(x20)`

Scheduling Example

- **Schedule this for dual-issue RISC-V**



```
Loop:  lw    x31, 0(x20)
       addi  x20, x20, -4
       add   x31, x31, x21
       sw    x31, 4(x20)
       blt   x22, x20, Loop
```

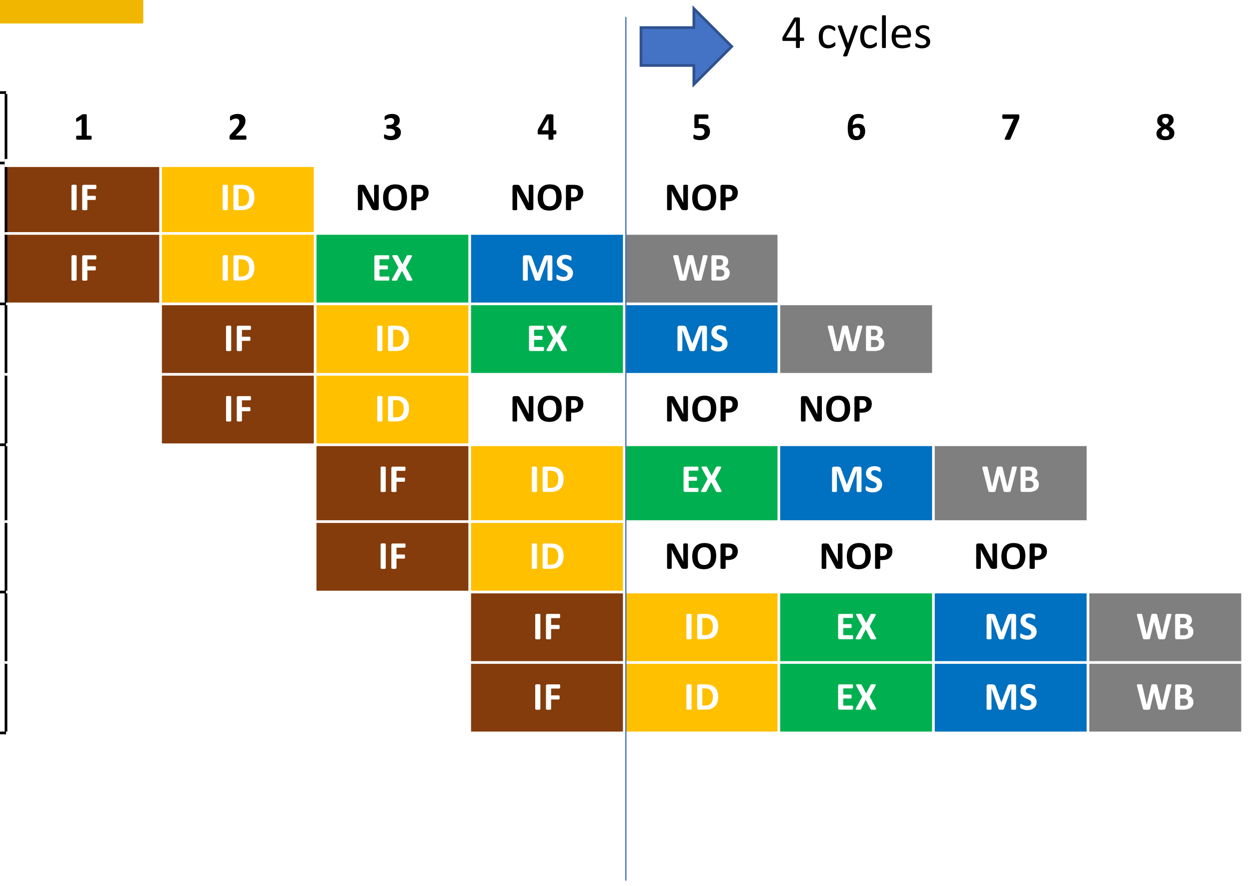
Fill up with **nop**

	Slot1 : ALU/BRANCH	Slot 2: Load/store
`Loop:`	`nop`	`lw x31, 0(x20)`
	`addi x20, x20, -4`	`nop`
	`add x31, x31, x21`	`nop`
	`blt x22, x20, Loop`	`sw x31, 4(x20)`

Example Baseline VLIW Processor with Two Slots – Execution Latencies = 1

- Performance: IPC = 5 instr / 4 cycles = 1.25 (peak IPC = 2)

Slot1 : ALU/BRANCH	Slot 2: Load/store
`nop`	`lw x31, 0(x20)`
`addi x20, x20, -4`	`nop`
`add x31, x31, x21`	`nop`
`blt x22, x20, Loop`	`sw x31, 4(x20)`



1	2	3	4	5	6	7	8
IF	ID	NOP	NOP	NOP			
IF	ID	EX	MS	WB			
	IF	ID	EX	MS	WB		
	IF	ID	NOP	NOP	NOP		
		IF	ID	EX	MS	WB	
		IF	ID	NOP	NOP	NOP	
			IF	ID	EX	MS	WB
			IF	ID	EX	MS	WB

Compiler Optimization - Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces 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 dependence” - Reuse of a register name
- Unroll factor: Number of loop body replications
- Fully unrolled: Number of loop body replications equal to number of iterations

Unrolled Code Example

- Unroll factor = 4:

```
Loop: lw   x31, 0(x20)
      add  x31, x31, x21
      sw   x31, 0(x20)
      addi x20, x20, -4
      blt  x22, x20, Loop
```

```
lp:     lw      x28,0(x20)      # x28=array element
        Lw      x29,-4(x20)     # x29=array element
        lw      x30,-8(x20)     # x30=array element
        lw      x31,-12(x20)    # x31=array element
        add     x28,x28,x21     # add scalar in x21
        add     x29,x29,x21     # add scalar in x21
        add     x30,x30,x21     # add scalar in x21
        add     x31,x31,x21     # add scalar in x21
        sw      x28,0(x20)      # store result
        sw      x29,-4(x20)     # store result
        sw      x30,-8(x20)     # store result
        sw      x31,-12(x20)    # store result
        addi    x20,x20,-16     # decrement pointer
        blt     x22,x20,lp      # branch if x22 < x20
```

Loop Unrolling Example - – Optimized Code for VLIW

Optimization:

lw, sw **offsets** are adapted to move **addi** into first pack.

No load-use RAW data hazards, so no influence on performance

	ALU/branch	Load/store	cycle
Loop:	**addi x20, x20, -16**	**lw x28, 0(x20)**	1
	nop	**lw x29, 12(x20)**	2
	add x28, x28, x21	**lw x30, 8(x20)**	3
	add x29, x29, x21	**lw x31, 4(x20)**	4
	add x30, x30, x21	**sw x28, 16(x20)**	5
	add x31, x31, x21	**sw x29, 12(x20)**	6
	nop	**sw x30, 8(x20)**	7
	blt x22, x20, Loop	**sw x31, 4(x20)**	8

- **IPC = 14/8 = 1.75**
- Closer to 2, but at cost of registers and code size
- Instruction Count (IC) of loop also reduced, less loop iteration checks

Limits of VLIW

- Branches and Labels break sequential instruction execution (code basic blocks)
- Hard to find sufficient Instruction Level Parallelism in single basic block

➢ Compiler Optimization techniques:
  - ➢ Loop unrolling
  - ➢ function inlining: function becomes part of the caller code
  - ➢ SW pipelining: schedules instructions from different iterations together
  - ➢ trace scheduling & superblocks: schedule beyond basic block boundaries

- Code Size Increase (e.g. due to loop unrolling, function inlining)

- Binary Compatibility: If the micro-architecture is changed, VLIW code may not be compatible anymore because it depends on the latencies.

Dynamic Multi-Issue

Superscalar

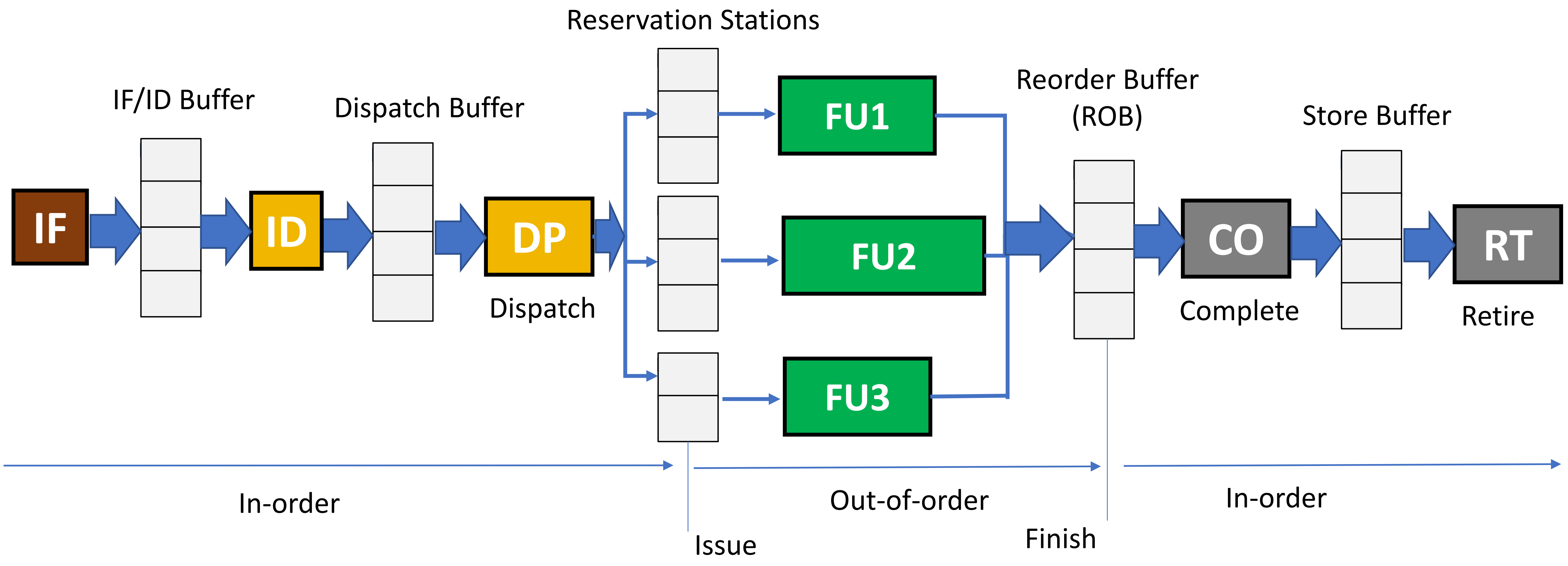
Superscalar

- Exploits Instruction Level Parallelism
- In-order: In order issue but pipeline (not compiler) selects issue bundles
- Out-of-order (OoO): dynamically scheduled

- Phases of instruction execution:
  Fetch – decode – rename – dispatch – issue – execute – complete – commit (retire)

Archetype of a OoO Superscalar Pipeline

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



Superscalar vs. VLIW

- Superscalar requires more complex hardware for instruction scheduling

➢issue buffers for OoO execution

➢complicated multiplexing between instruction issue structure & functional units

➢dependence checking logic between parallel instructions

➢functional unit hazard checking

➢VLIW requires a complex compiler and higher code size (e.g. slower due to less efficient use of instruction cache)

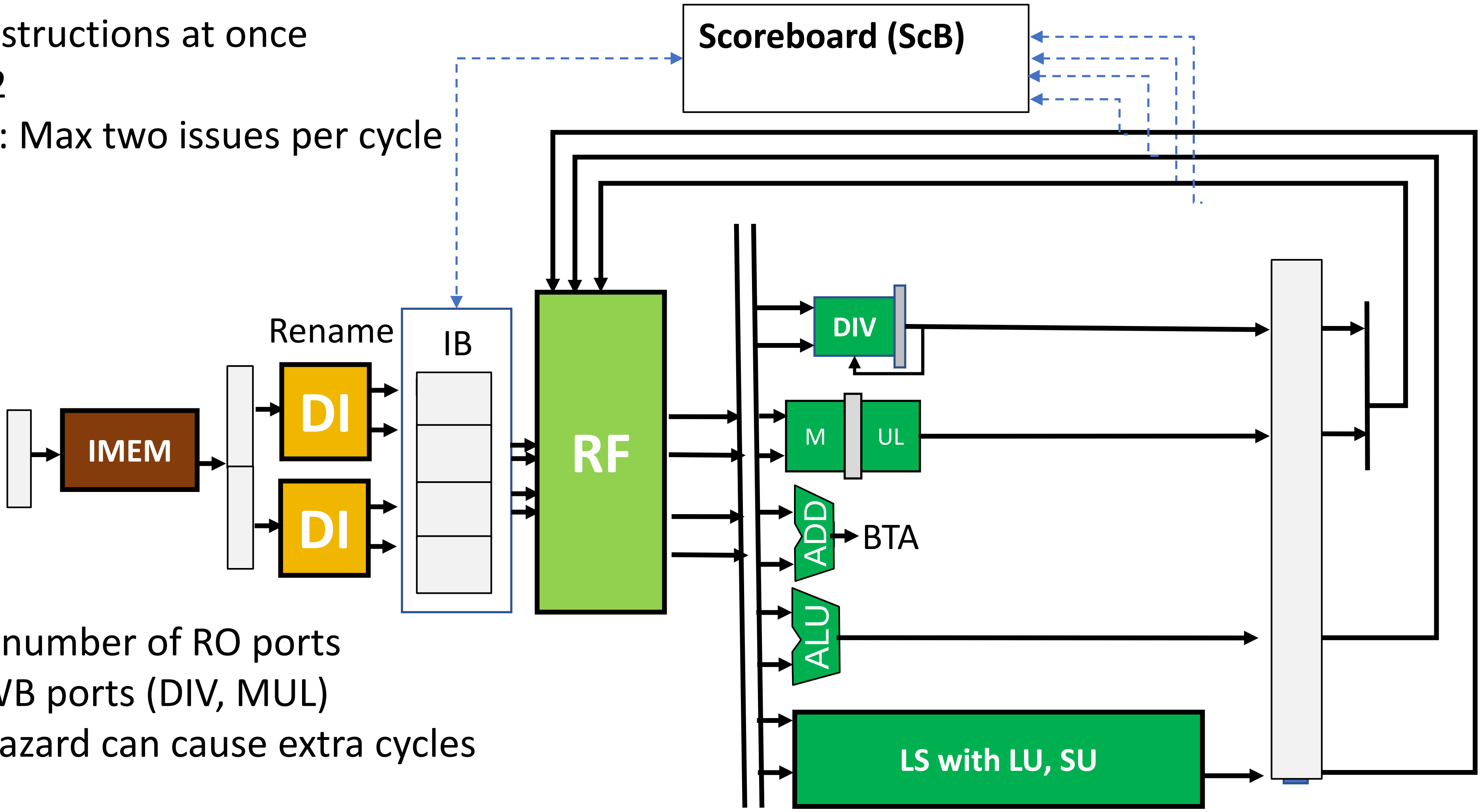
➢Superscalars can execute pipeline-dependent code more efficiently : don't have to recompile if binary is executed on different processors (pre-compiled libraries)

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

Wide instruction fetch can fetch two instructions at once

Ideal IPC = 2

Reduce HW: Max two issues per cycle



Reduce the number of RO ports and share WB ports (DIV, MUL)

Structural hazard can cause extra cycles

Unrolled Code Example

- Unroll factor = 4:

```
Loop: lw   x31, 0(x20)
      add  x31, x31, x21
      sw   x31, 0(x20)
      addi x20, x20, -4
      blt  x22, x20, Loop
```

```
lp:    lw     x28,0(x20)      # x28=array element
       Lw     x29,-4(x20)     # x29=array element
       lw     x30,-8(x20)     # x30=array element
       lw     x31,-12(x20)    # x31=array element
       add    x28,x28,x21     # add scalar in x21
       add    x29,x29,x21     # add scalar in x21
       add    x30,x30,x21     # add scalar in x21
       add    x31,x31,x21     # add scalar in x21
       sw     x28,0(x20)      # store result
       sw     x29,-4(x20)     # store result
       sw     x30,-8(x20)     # store result
       sw     x31,-12(x20)    # store result
       addi   x20,x20,-16     # decrement pointer
       blt    x22,x20,lp      # branch if x22 < x20
```

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	ALU	WB											
`lw   x28, 0(x20)`	IF	IS	IB	LU	LU	WB									
`Lw x29,12(x20)`		IF	IS	IB	LU	LU	WB								
`add x28,x28,x21`		IF	IS	IB	IB	ALU	WB								
`lw x30,8(x20)`			IF	IS	IB	LU	LU	WB							
`add x29,x29,x21`			IF	IS	IB	IB	ALU	WB							
`lw x31,4(x20)`				IF	IS	IB	LU	LU	WB						
`add x30,x30,x21`				IF	IS	IB	IB	ALU	WB						
`sw  x28,16(x20)`					IF	IS	IB	SU	SB						
`add x31,x31,x21`					IF	IS	IB	IB	ALU	WB					
`sw x29,12(x20)`						IF	IS	IB	SU	SB					
`sw  x30,8(x20)`						IF	IS	IB	IB	SU	SB				
`sw  x31,4(x20)`							IF	IS	IB	IB	SU	SB			
`blt x22,x20, Loop`							IF	IS	IB	BR					

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

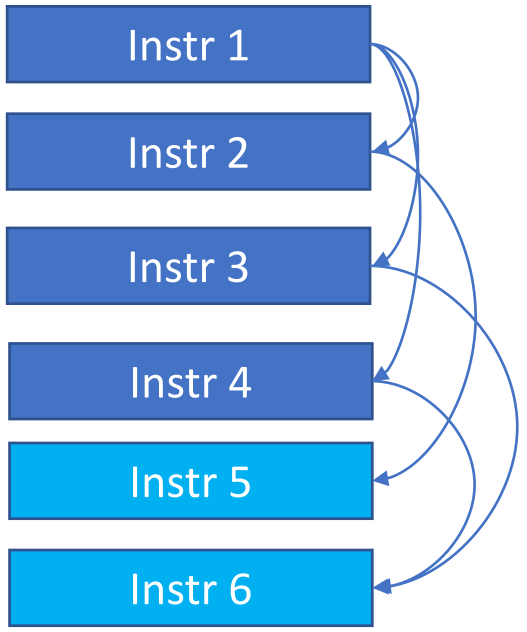
11-3=
8 cycles

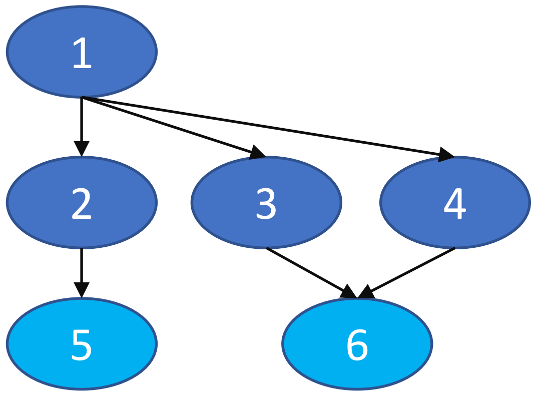
14 instructions

CPI = 0,57
IPC=1,75

Instruction Scheduling for Superscalar

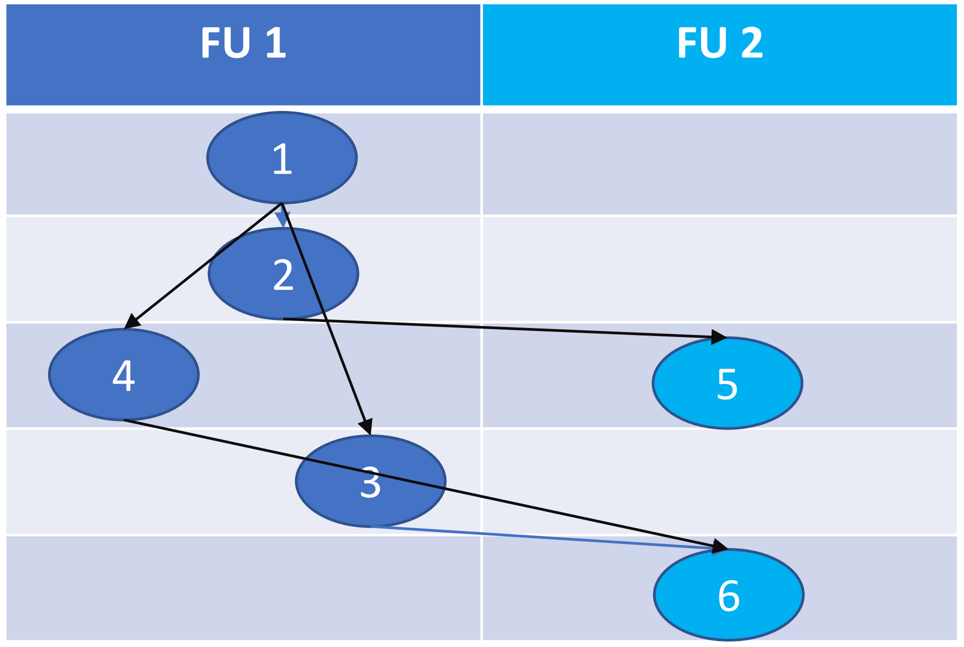
- The process of mapping a series of instructions into execution resources
- Decides when and where an instruction is executed





Dependence graph

1,2,3,4 can execute on FU1
5,6 can execute on FU 2



Derived from CA course of Mikko Lipasti-University of Wisconsin

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

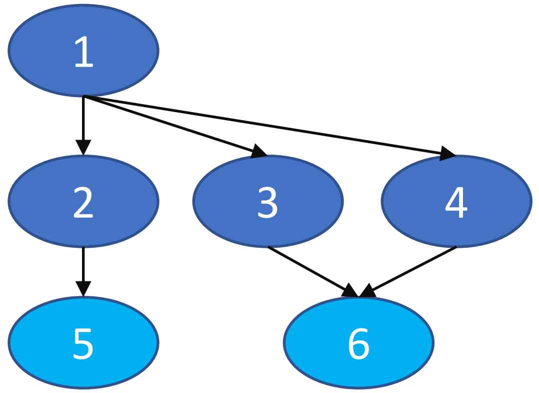
➢Resolves resource conflicts

➢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

Multithreading

Threads

- **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 Parallelism (TLP)** to improve uniprocessor throughput (IPC)

Thread-level parallelism (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.

Example: Use of FUs by Single Thread

Superscalar

Cycle	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	🟧			🟧

Time

Pattern for Superscalar Execution:

- Cycles that a certain instruction of the thread uses a specific FU (EX stage)
- Time now runs from top to bottom.
- We need to rotate the pipeline diagram by 90 deg.

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)

- **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)

Time

Coarse-grained MT

Cycle	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			🟦	

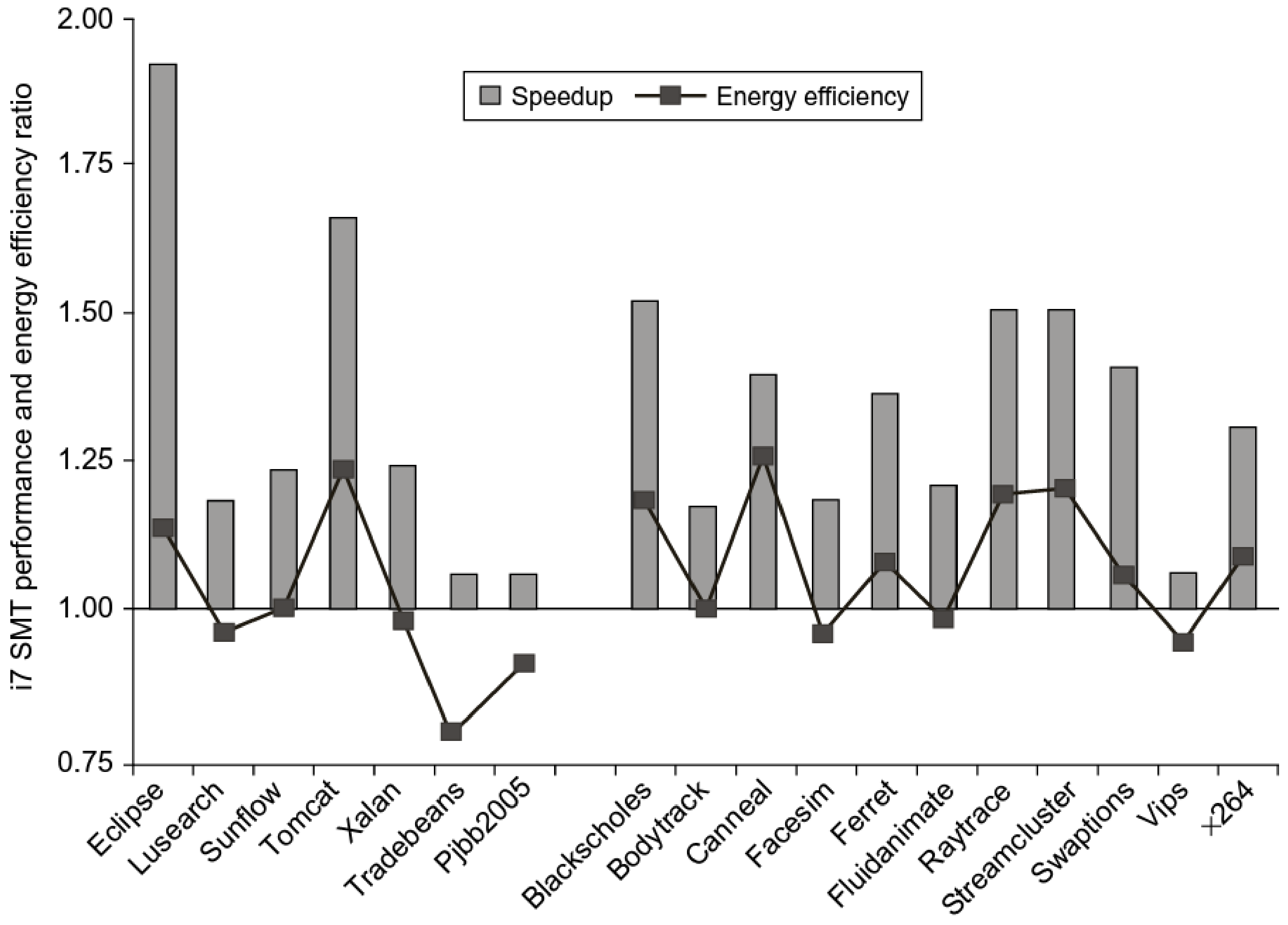
Fine-grained MT

Cycle	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		🟧		

Simultaneous MT (SMT)

Cycle	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		🟧		🟧

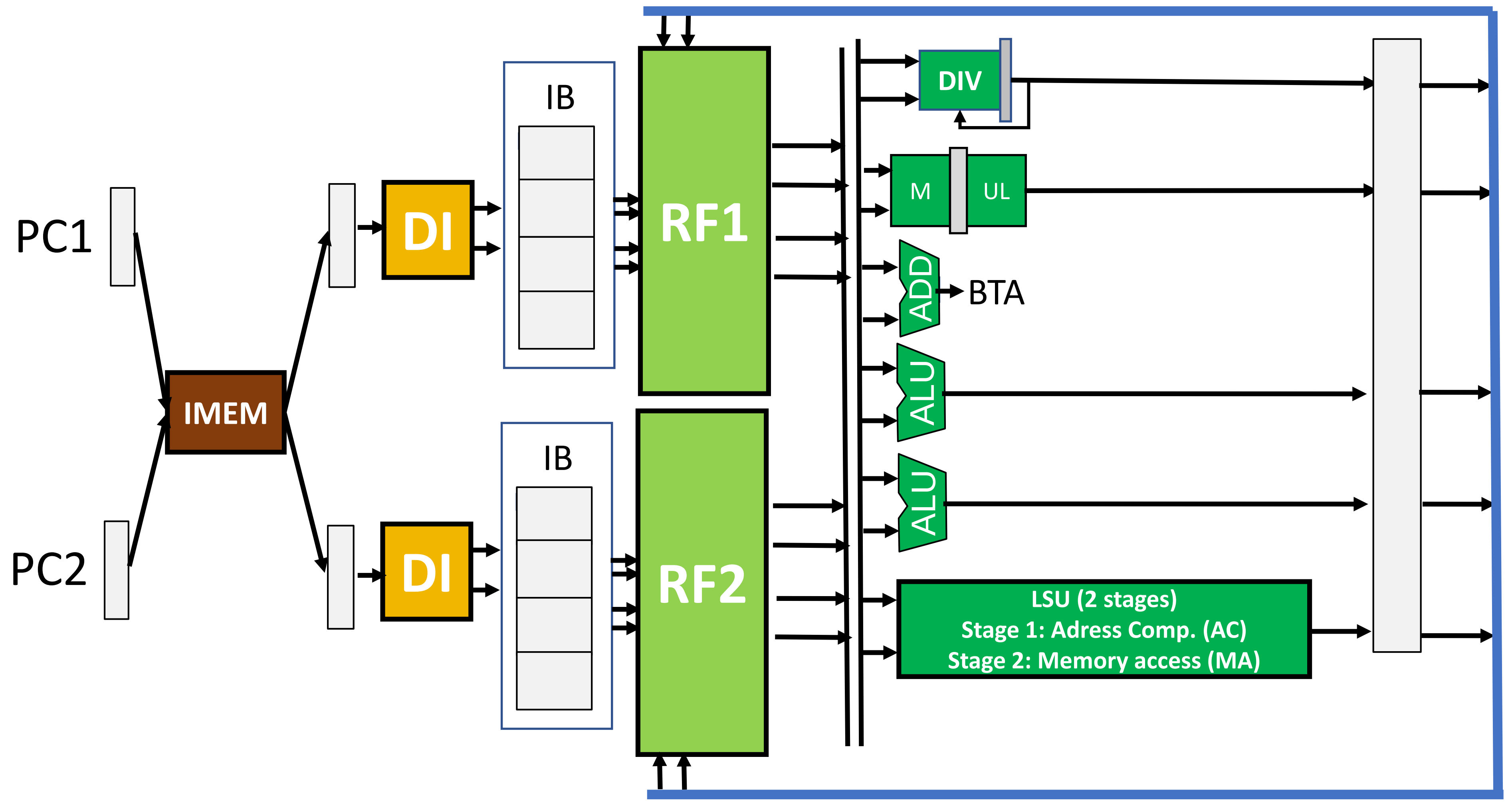
The speedup from using multithreading on one core on an i7 processor



Source: Computer Architecture – A Quantitative Approach 5th Edition Fig. 3.33

Example: Simple Dual Multi-threaded Processor

- EX has 1xDIV, 1xMUL, 1x Branch/ALU, 1xALU, 1xLSU



Example with Stall due to I-Cache Miss

```
Loop: lw   x31, 0(x20)
      add  x31, x31, x21
      sw   x31, 0(x20)
      addi x20, x20, -4
      blt  x22, x20, Loop
```

Cycle – i +	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
`lw x31,0(x20)`	IF	IS	LSU (AC)	LSU (MA)	WB										
`add x31,x31,x21`		IF	IS	IB	ALU	WB									
`sw x31,0(x20)`			IF	IS	IB	LSU (AC)	SB								
`addi x20,x20,-4`				IF	IS	ALU	WB								
`blt x22,x20,Loop`					IF	IS	BR								
`lw x31,0(p20)`						IF	IS	LSU (AC)	🟨	🟨	🟨	LSU (MA)			

Cache miss

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

Example with Stall due to I-Cache Miss

Cycle - i +	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
`lw x31,0(x20)`	IF	IS	LSU (AC)	LSU (MA)	WB										
`add x31,x31,x21`		IF	IS	IB	ALU	WB									
`sw x31,0(x20)`			IF	IS	IB	LSU (AC)	SB								
`addi x20,x20,-4`				IF	IS	ALU	WB								
`blt x22,x20,Loop`					IF	IS	BR								
`lw x31,0(x20)`						IF	IS	LSU (AC)		Cache miss		LSU (MA)			

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

FU USE - cycle i +	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
`ALU`					🟧	🟧	🟧								
`LSU (AC)`			🟧			🟧		🟧		Cache miss					
`LSU (MA)`				🟧								🟧			

Example with Stall due to I-Cache Miss

```
Loop: lw   x31, 0(x20)
      add  x31, x31, x21
      sw   x31, 0(x20)
      addi x20, x20, -4
      blt  x22, x20, Loop
```

Cycles run from top to bottom

Thread 1

Cycle	ALU	LSU (AC)	LSU (MA)
i+3		lw	
i+4			Lw
i+5	add		
i+6	addi	sw	
i+7	blt		
i+8		lw	
i+9		Cache	
i+10		Miss	
i+11		…	
i+12			lw
i+13	add		
i+14	addi	sw	

Thread 2

Cycle	ALU	LSU (AC)	LSU (MA)
i+3		lw	
i+4			lw
i+5	add		
i+6	addi	sw	
i+7	blt		
i+8		lw	
i+9			lw
i+10	add		
i+11	addi	sw	
i+12	blt		

Example with Stall due to I-Cache Miss

Thread 1

Cycle	ALU	LSU (AC)	LSU (MA)
i+3		lw	
i+4			Lw
i+5	add		
i+6	addi	sw	
i+7	blt		
i+8		lw	
i+9		Cache	
i+10		Miss	
i+11		…	
i+12			lw
i+13	add		
i+14	addi	sw	

Thread 2

Cycle	ALU	LSU (AC)	LSU (MA)
i+3		lw	
i+4			lw
i+5	add		
i+6	addi	sw	
i+7	blt		
i+8		lw	
i+9			lw
i+10	add		
i+11	addi	sw	
i+12	blt		

Multithreaded - SMT

Cycle	ALU	LSU (AC)	LSU (MA)
i+3		lw	
i+4		lw	lw
i+5	add		lw
i+6	add	sw	
i+7	addi	sw	
i+8	addi		
i+9	blt		
i+10	blt	lw	
i+11		lw	
i+12			lw
i+13	add		
i+14	addi	sw	Lw
i+15	add		
i+16	blt		
i+17	addi	sw	

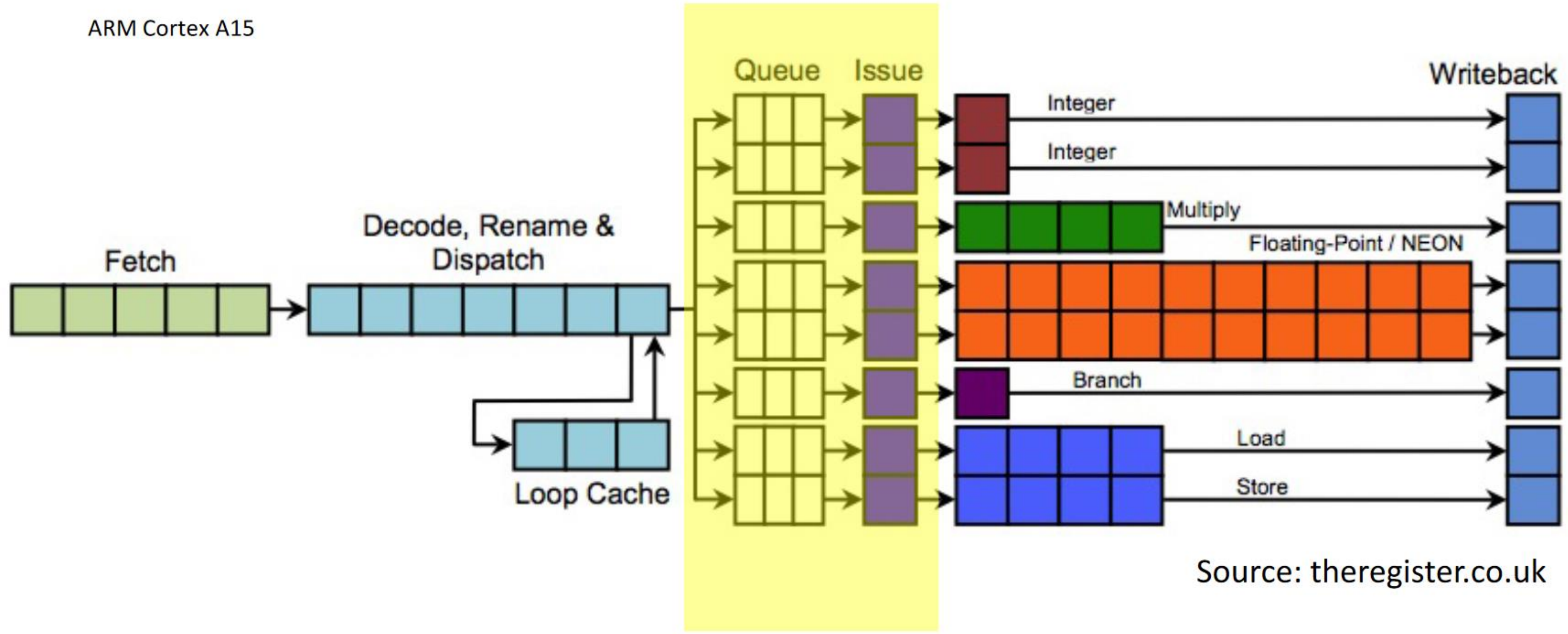
Optional, not relevant for exam

A Look at Real Processors

A15 and BOOM

ARM A15 Superscalar Core

- ARM A15 pipeline diagram:

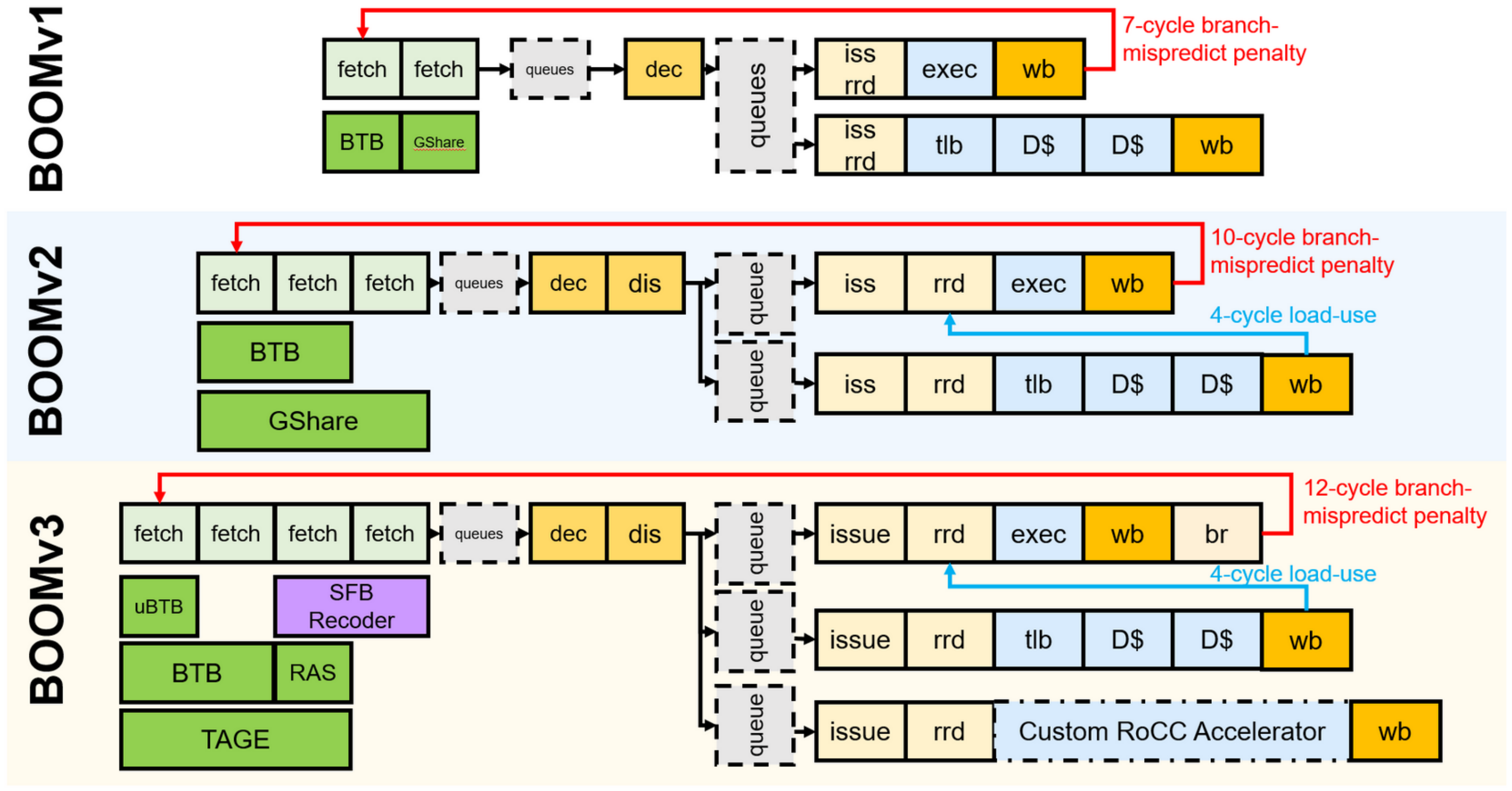


Source: theregister.co.uk

(Copied from from slides of CS course Mikko Lipasti-University of Wisconsin)

Berkeley Out-of-order Machine (BOOM)

- BOOM: an open-source out-of-order RISC-V core



Source: https://github.com/riscv-boom/riscv-boom

Summary

Where we are

- We covered the following features: **Branch prediction, Out of order execute, Scoreboard, Superpipelining, Multi-issue, Superscalar, VLIW, Multi-threading**

- Instruction Level Parallelism: VLIW, Superscalar
- Thread Level Parallelism: Multi-threaded Single Core Processor

- Upcoming:

➢Thread Level Parallelism: Multi-Core (MIMD)

➢Data level parallelism: Vector (SIMD)

Thank you for your attention!