Advanced Computer Architecture (191.019)

Exercises

Repository

Contents

2 RISC-V Assembly and Compilers

a) Can you mix C-Code and assembly code in one project? What do you need to take care about?

Solution: Yes, but you must adhere to the ABI

b) What is the RISC-V instruction to add register a1 and a2 and store the result in a0?

Solution: `ADDI a0, a1, a2`

c) What is the RISC-V instruction to subtract 4 from register a1 and save the result in register a2?

Solution: `ADDI a2, a1, -4`

d) What alternative instruction can you use to multiply a value by 4 if you do not want to use the MUL instruction?

Solution: `SLLI <rd>, <rs1>, 2`

e) Your current PC value is `0x00AA0000` What is the PC value after executing the instruction J 16?

Solution: `J 16` is pseudo instruction for `JAL zero, 16`;

$$\begin{aligned} \text{PC}_{\text{next}} &= \text{PC} + 16 \\ &= \texttt{0x00AA0000} + \texttt{0x10} \\ &= \texttt{0x00AA0010} \end{aligned}$$

PC value will be `0x00AA0010`

> ⚠ **Warning**
>
> Most tools do not allow to add immediates to J-Type instructions. Thus, it is not well defined. According to TUWEL question immediate will not be shifted and taken as it is.

f) Write the RISC V assembly that implements this C function. Please make sure to adhere to the RISC-V ABI.

```c
int acc ( int * x )
{
  int z;
  z = x [0] + x [1] + x [2] + x [3];
  return z ;
}
```

```asm
acc:
  LW t0, 0(a0)
  LW t1, 4(a0)
  LW t2, 8(a0)
  LW t3, 12(a0)
  ADD t0, t0 ,t1
  ADD t2, t2, t3
  ADD a0, t0, t2
  JALR zero, 0(ra) // ret
```

g) Generate the LLVM IR representation with no optimization and explain the generated LLVM IR code.

Solution: `> clang -S -emit-llvm acc.c -o acc noopt.ll --target=riscv32` will result in

```llvm
...
; Function Attrs: noinline nounwind optnone
define dso_local i32 @acc(i32* noundef %0) #0 {
  %2 = alloca i32*, align 4
  %3 = alloca i32, align 4
  store i32* %0, i32** %2, align 4
  %4 = load i32*, i32** %2, align 4
  %5 = getelementptr inbounds i32, i32* %4, i32 0
  %6 = load i32, i32* %5, align 4
  %7 = load i32*, i32** %2, align 4
  %8 = getelementptr inbounds i32, i32* %7, i32 1
  %9 = load i32, i32* %8, align 4
  %10 = add nsw i32 %6, %9
  %11 = load i32*, i32** %2, align 4
  %12 = getelementptr inbounds i32, i32* %11, i32 2
  %13 = load i32, i32* %12, align 4
  %14 = add nsw i32 %10, %13
  %15 = load i32*, i32** %2, align 4
  %16 = getelementptr inbounds i32, i32* %15, i32 3
  %17 = load i32, i32* %16, align 4
  %18 = add nsw i32 %14, %17
  store i32 %18, i32* %3, align 4
  %19 = load i32, i32* %3, align 4
  ret i32 %19
}
```

Explanations:

Line 4,5,6,7,10,11,14,22,24 Stack frame and load stores for input parameter and return value (for debug),

Line 8,11,15 `getelementptr` is pointer arithmetic. In this case fist `i32` is type, second `i32* %4` is the basepointer and third `i32 0` is the offset (first element 0, next 1, next 2, next 3). For integer variables with width 32 bit = 4 byte, in byte addressable memory these offsets should then be multiplied by 4 for address offsets. Attribute inbounds promises the pointer access will not be out of bounds (segmentation fault).

The rest of the code is straight forward, load, add and return.

h) Generate the LLVM IR representation with optimization level O2 and explain the generated LLVM IR code.

Solution: `> clang -S -emit-llvm acc.c -o acc opt.ll -O2 --target=riscv32` produces:

```llvm
...
; Function Attrs: mustprogress nofree norecurse
nosync nounwind readonly willreturn
define dso_local i32 @acc(i32* nocapture noundef
readonly %0) local_unnamed_addr #0 {
  %2 = load i32, i32* %0, align 4, !tbaa !4
  %3 = getelementptr inbounds i32, i32* %0, i32 1
  %4 = load i32, i32* %3, align 4, !tbaa !4
  %5 = add nsw i32 %4, %2
  %6 = getelementptr inbounds i32, i32* %0, i32 2
  %7 = load i32, i32* %6, align 4, !tbaa !4
  %8 = add nsw i32 %5, %7
  %9 = getelementptr inbounds i32, i32* %0, i32 3
  %10 = load i32, i32* %9, align 4, !tbaa !4
  %11 = add nsw i32 %8, %10
  ret i32 %11
}
...
```

Explanations:

- The lines Line 4,5,6,7,10,11,14,22,24: of the non-optimized code were removed.

i) Compile the optimized version to RISC-V assembler and compare the result to your written solution.

Solution: `>clang -O2 acc.c -o acc opt.S -S --target=riscv32` produces:

```asm
    .text
    .attribute 4, 16
    .attribute 5, "rv32i2p0_m2p0_a2p0_c2p0"
    .file "acc.c"
    .globl acc
    .p2align 1
    .type acc,@function
acc:
    lw a1, 0(a0)
    lw a2, 4(a0)
    lw a3, 8(a0)
    lw a0, 12(a0)
    add a1, a1, a2
    add a1, a1, a3
    add a0, a0, a1
    ret
// ...
```

3 Static Code Analysis

3.1 Data and Control Flow Analysis

Given is the program `solvequad` that computes the solutions x_1 and x_2 of the quadratic equation $ax^2 + bx + c = 0$ as:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

We assume that the solutions are always real. The following program is used to compute the solutions:

```c
void solvequad ( double a , double b , double c , double * x1 , double * x2 ) {
  * x1 = ( -1* b + mysqrt ( b *b -4* a * c ) ) /2* a ;
  * x2 = ( -1* b - mysqrt ( b *b -4* a * c ) ) /2* a ;
}
double mysqrt ( double x ) {
  double y =0.22222 + 0.888889 * x ;
  int i = 0;
  while (i <3)
  {
    y = 0.5 * ( y + x / y ) ;
    i = i +1;
  }
  return y ;
}
```

The three-address code is given as:

```
1   solvequad:
2     t1 := -1* b                          } (1) B1
3     t2 :=2* a
4     t3 := b * b
5     t4 :=4* a
6     t5 := t4 * c
7     t6 := t3 - t5
8     param t6
9     t7 := call mysqrt ,1
10    t8 := t1 + t7                        } (2) B2
11    t9 := t1 - t7
12    t10 := t8 / t2
13    t11 := t9 / t2
14    * x1 := t10
15    * x2 := t11
16    return
17  mysqrt:
18    t12 := 0.888889* x                   } (3) B3
19    y :=0.222222+ t12
20    i :=0
21  mysqrt_loop1:
22    t13 := x / y                         } (4) B4
23    t14 := y + t13
24    y :=0.5* t14
25    i := i +1
26    if i < 3 goto mysqrt_loop1
27    return y                             } (5) B5
```

a) Mark the basic blocks in the IR code.

Solution: Comments in code

b) Draw the control flow graph for this program.

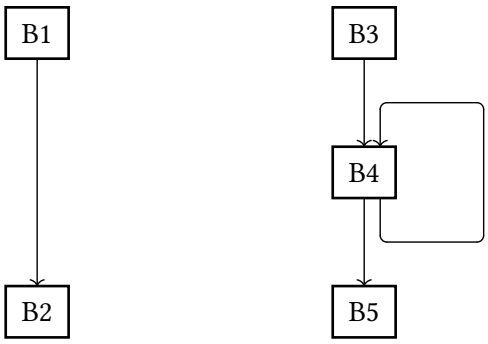


Figure 1: Control flow graph

c) Draw the data flow graph for each basic block of the IR code.

Solution:

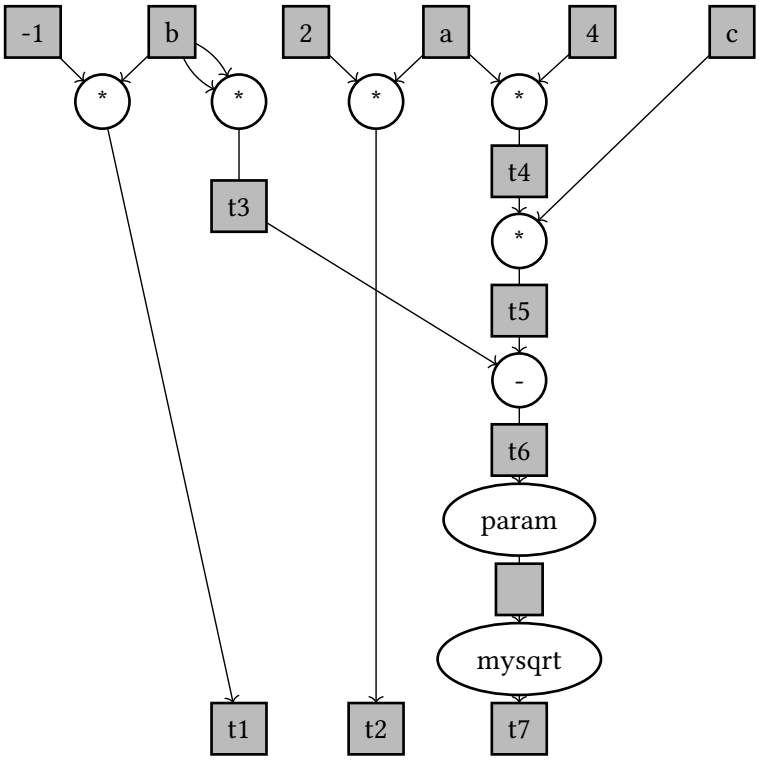


Figure 2: Data Flow graph for basic Block B1

3.2 Code optimization

The Taylor expansion can be used to compute an approximation of the sine function:

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} \ldots$$

The following sub-optimal implementation of this approximation is given as a C function:

```c
float sine_taylor(float x) {
    float sinx;
    sinx = x - (x*x*x) / (3*2*1) + (x*x*x*x*x) /(5*4*3*2*1);
    return sinx;
}
```

A straight-forward translation of this program into intermediate representation leads to the following three address code:

```
1     t1  := x * x
2     t2  := t1 * x
3     t3  := 1 * 2
4     t4  := t3 * 3
5     t5  := t2 / t4
6     t6  := x - t5
7     t7  := x * x
8     t8  := t7 * t7
9     t9  := t8 * x
10    t10 := 1 * 2
11    t11 := t10 * 3
12    t12 := t11 * 4
13    t13 := t12 * 5
14    t14 := t9 / t13
15    sinx := t6 + t14
16    return sinx
```

a) Optimize this IR code using classical compiler optimization steps one by one and name the optimization method.

Solution:

1) Constant propagation (calculate the constants and use them as such):

```
1     t1  := x * x
2     t2  := t1 * x
3     t5  := t2 / 6
4     t6  := x - t5
5     t7  := x * x
6     t8  := t7 * t7
7     t9  := t8 * x
8     t14 := t9 / 120
9     sinx := t6 + t14
10    return sinx
```

2) Common subexpression elimination (remove double calculation of x^2 and x^3):

```
1    t1  := x * x
2    t2  := t1 * x
3    t5  := t2 / 6
4    t6  := x - t5
5    t9  := t1 * t2
6    t14 := t9 / 120
7    sinx := t6 + t14
8    return sinx
```

3.3 Live variable analysis

a) Conduct a live variable analysis to compute the set of live variables at the entry and end of basic block `B1`. For this provide `USE[B1]`, `DEF[B1]`, `OUT[B1]` and `IN[B1]`.

The IR code for `B1` is given as:

```
1 B1:
2   t1 = x[i];
3   t2 = t1 * c ;
4   t2 = t2 -5;
5   j = j + 1;
6   z [j] = t2 ;
7   if t2 < 0 goto B3
8 B2:
9   ...
```

B2 and B3 are successors of B1 with current IN[B2]= $\{x, z, i, j\}$ and IN[B3]= $\{z, \mathrm{t2}\}$.

Solution:

- USE[B1] = $\{x, i, c, j, z\}$ (this does not include the variables from DEF)
- DEF[B1] = $\{\mathrm{t1}, \mathrm{t2}\}$
- IN[B1] = USE[B1] $\cup$ (OUT[B1] $-$ DEF[B1]) = $\{x, i, c, j, z\}$
- OUT[B1] = IN[B2] $\cup$ IN[B3] = $\{x, z, i, j, \mathrm{t2}\}$

b) How many registers do you need to use to store all values in this basic block?

Solution: Draw lines for each variables lifetime next to the code, this is probably the easiest way to check.

It is 6. 4 for the variables that are in IN and OUT ($\{x, i, j, z\}$), one for c in the beginning, and one shared register for t1 and t2.

c) *Bonus*: Given is the C-Code for the quicksort algorithm as well as the IR code and the control flow graph on the next pages. Run the complete live variable analysis for the quicksort algorithm.

> **i Note**
>
> This is solved in the solutions.

4 Five Stage Pipeline and Branch Prediction

4.1 Scalar in-order 5-stage Pipeline

Given are the following RISC-V Assembly codes for the same function.

```asm
acc:
  lw t0,0(a0)
  lw t1,4(a0)
  add t0,t0,t1
  lw t1,8(a0)
  add t0,t0,t1
  lw t1,12(a0)
  add t0,t0,t1
  mv a0,t0
  ret
```

```asm
acc:
  lw a1, 0(a0)
  lw a2, 4(a0)
  lw a3, 8(a0)
  lw a0, 12(a0)
  add a1, a1, a2
  add a1, a1, a3
  add a0, a0, a1
  ret
```

a) Determine the number of cycles it takes to execute this program on the scalar, in-order, 5-stage pipeline from the slides **without forwarding paths**.

Solution: Draw the data dependencies into the code and note down how many cycles delay they cause. Then count up the instructions and the added delay cycles.

> **i Note**
>
> The `ret` instructions takes 3 cycles, since it takes one for normal executions and the next 2 instructions get discarded because of the jump.

1) 19 cycles
2) 14 cycles

b) Determine the number of cycles it takes to execute this program on the scalar, in-order, 5-stage pipeline from the slides **with forwarding paths**.

Solution:
1) 14 cycles
2) 10 cycles

4.2 Branch Prediction

In machine learning a possible activation function is leaky ReLU. For each acti-vation output a_i scales the negatives linear accumulators z_i with a small factor c and the positive with a larger factor d.

The following integer implementation is given as c function:

```c
void act_leaky_relu_size64(int *z , int *a, int c, int d) {
  int i;
  for (i = 0; i < 64; i++) {
    if ( z[i] < 0) {
      a[i] = z[i] * c;
    } else {
      a[i] = z[i] * d;
    }
  }
  return;
}
```

A CLANG compilation of this program for RISCV32 leads to the following assembly code:

```asm
  (...)
act_leaky_relu_size64: // B0
  li a4, 0                          } (1) B0
  li a6, 256                        }
  j .LBB0_2                         }
.LBB0_1: // B1
  mul a7, a5, a7                    } (2) B1
  add a5, a1, a4                    }
  addi a4, a4, 4                    }
  sw a7, 0(a5)                      }
  beq a4, a6, .LBB0_4               }
.LBB0_2: // B2
  add a5, a0, a4                    } (3) B2
  lw a5, 0(a5)                      }
  mv a7, a2                         }
  bltz a5, .LBB0_1                  }
  mv a7, a3 // B3                   } (4) B3
  j .LBB0_1                         }
.LBB0_4: // B4
  ret                               } (5) B4
.Lfunc_end0:
  (...)
```

Assume z_0, z_1, z_3 to be positive and z_2 to be negative. Create a table for different branch predictors with the following contents:

- Code Line Nr. of Branch that is executed.
- Predictor State (with entry for branch instruction address, use code line number as substitute)
- Prediction Direction
- Direction
- Prediction Correct/Wrong

- Branch Target buffer entry (use Assembly Code Line Nr. as substitute for the branch instruction address and branch target address)

a) What is the basic block sequence of execution up to the computation of a_3? Use the assigned basic block nr. to indicate the basic block, e,g, a jump to label LBB0_2 is equal to a jump to B2.

Solution: Basic Block Sequence:
B0, B2, B3, B1, B2, B3, B1, B2, B1, B2, B3, B1

Start: B0
For positive z_0, z_1 : Two times B2, B3, B1
For negative z_2 : B2, B1
For positive z_3 : B2, B3, B1

b) Local Branch Predictor, 2-bit, start state undefined: Predictor always uses not taken (NT) as first prediction if no BTB entry exists, and then initializes to weakly not taken (WNT) state. No entries are cached in the BTB. The BTB has two slots for entries. Unconditional jumps and return statements are ignored by the predictor.

Solution: We assume if there is no predictor state entry yet, that we initialize with the weak state.

Basic Block	**Branch Line Nr.**	**Pred. Direct.**	**Direction**	**Wrong/ Correct**	**Pred. State 1**	**BTB Entry 1**	**Pred. State 2**	**BTB Entry 2**
B0	-	-	-	-	-	-	-	-
B2	L16	NT	NT	C	L16:WNT	L16:L6	-	-
B3	-	-	-	-	L16:WNT	L16:L6	-	-
B1	L11	NT	NT	C	L16:WNT	L16:L6	L11:WNT	L11:L19
B2	L16	NT	NT	C	L16:SNT	L16:L6	L11:WNT	L11:L19
B3	-	-	-	-	L16:SNT	L16:L6	L11:WNT	L11:L19
B1	L11	NT	NT	C	L16:SNT	L16:L6	L11:SNT	L11:L19
B2	L16	NT	T	W	L16:WNT	L16:L6	L11:SNT	L11:L19
B1	L11	NT	NT	C	L16:WNT	L16:L6	L11:SNT	L11:L19
B2	L16	NT	NT	C	L16:SNT	L16:L6	L11:SNT	L11:L19
B3	-	-	-	-	L16:SNT	L16:L6	L11:SNT	L11:L19
B1	L11	NT	NT	C	L16:SNT	L16:L6	L11:SNT	L11:L19

5 Out of Order Execution

5.1 Instruction Dependencies

The following integer implementation is given as c function:

```c
void act_leaky_relu_size64(int* z, int *a, int c, int d) {
  int i;
  for (i=0; i<64; i++) {
    if (z[i]<0) {
      a[i] = z[i] * c;
    } else {
      a[i] = z[i] * d;
    }
  }
  return;
}
```

A CLANG compilation of this program for RISCV32 leads to the following assem-bly code:

```asm
  (...)
act_leaky_relu_size64: // B0
  li a4, 0                    } (1) B0
  li a6, 256                  }
  j .LBB0_2                   }
.LBB0_1: // B1
  mul a7, a5, a7              } (2) B1
  add a5, a1, a4              }
  addi a4, a4, 4              }
  sw a7, 0(a5)                }
  beq a4, a6, .LBB0_4         }
.LBB0_2: //B2
  add a5, a0, a4              } (3) B2
  lw a5, 0(a5)                }
  mv a7, a2                   }
  bltz a5, .LBB0_1            }
  mv a7, a3 // B3             } (4) B3
  j .LBB0_1                   }
.LBB0_4: // B4
  ret                         } (5) B4
.Lfunc_end0:
  (...)
```

a) Mark all RAW, WAW, and WAR dependencies in the program. Only consider local dependencies within the basic blocks (not across basic blocks). *Solution:*

```asm
1    (...)
2  act_leaky_relu_size64: // B0
3    li a4, 0                                                    } (1) B0
4    li a6, 256                                                  }
5    j .LBB0_2                                                   }
6  .LBB0_1: // B1
7    mul a7, a5, a7
8    add a5, a1, a4 | WAR on line 7                              } (2) B1
9    addi a4, a4, 4 | WAR on line 8                              }
10   sw a7, 0(a5) | RAW on line 7,8                              }
11   beq a4, a6, .LBB0_4 | RAW on line 9                         }
12 .LBB0_2: //B2
13   add a5, a0, a4
14   lw a5, 0(a5) | WAW and RAW on line 13                       } (3) B2
15   mv a7, a2                                                   }
16   bltz a5, .LBB0_1 | RAW on line 13                           }
17   mv a7, a3 // B3                                             } (4) B3
18   j .LBB0_1                                                   }
19 .LBB0_4: // B4
20   ret                                                         } (5) B4
21 .Lfunc_end0:
22   (...)
```

5.2 Out of order processors

Given is the following basic block.

```asm
1   (...)
2 .LBB0_1: // B1
3   mul a7, a5, a7
4   add a5, a1, a4 | WAR on line 3
5   addi a4, a4, 4 | WAR on line 4
6   sw a7, 0(a5) | RAW on line 3,4
7   beq a4, a6, .LBB0_4 | RAW on line 5
8   (...)
```

a) Draw the pipeline diagram for the simple out of order pipeline from the slides (single instruction fetch, no renaming, no reorder buffer) with scoreboard.

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
`mul a7, a5, a7`	IF	ID	RO	MUL	MUL	WB							
`add a5, a1, a4`		IF	stall	stall	stall	ID	RO	ALU	WB				
`addi a4, a4, 4`						IF	stall	stall	ID	RO	ALU	WB	
`sw a7, 0(a5)`									IF	ID	RO	SU	SB
`beq a4, a6, .LBB0 4`										IF	ID	RO	ADD

i Note

We stall here because the pipeline does not allow an instruction with a `WAW` or `WAR` dependency to a previous one to enter.

b) Discuss, what is the problem with the scheme to block instructions with `WAW` and `WAR` dependencies to enter the pipeline.

Solution: They stall the pipeline for longer than necessary and prevent the following instructions from entering, even if they could be executed without any conflicts.

c) What alternative scheme could be used when not using register renaming for the `WAR` dependencies? What would be the resulting pipeline diagram?

Solution: The next instruction can be issued immediately after the WAR-dependency instruction has gone through the `RO` stage.

Cycle	1	2	3	4	5	6	7	8
`mul a7, a5, a7`	IF	ID	RO	MUL	MUL	WB		
`add a5, a1, a4`		IF	ID	RO	ALU	WB		
`addi a4, a4, 4`			IF	ID	RO	ALU	WB	
`sw a7, 0(a5)`				IF	ID	RO	SU	SB
`beq a4, a6, .LBB0 4`					IF	ID	RO	ADD

6 VLIW and Superscalar

6.1 VLIW

The following C code with corresponding RISC-V assembler code is given:

C-Code:

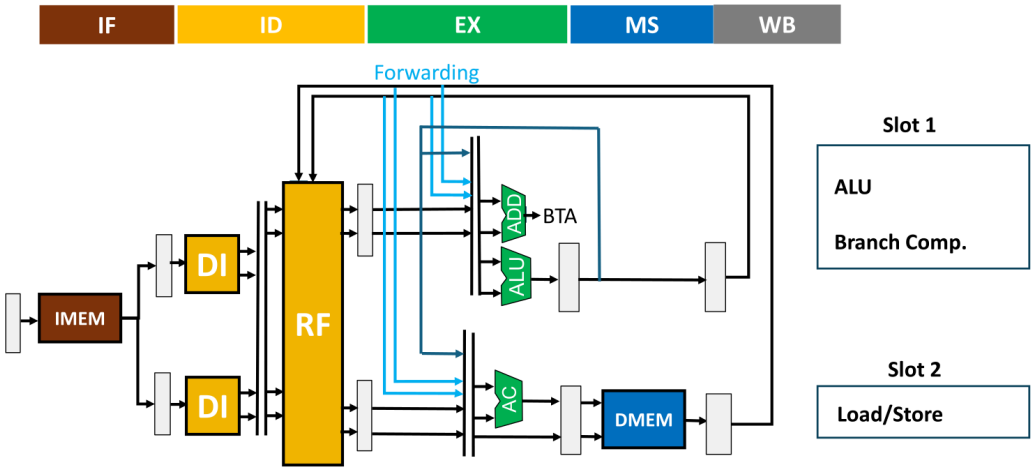
```c
#define N 2

void func(int *ad1, int *ad2,
          int *as1, int *as2, int s) {
  unsigned int i;
  for(i = 0; i < N; i++){
    ad1[i] = s+as1[i] + as2[i];
    ad2[i] = as1[i] - as2[i];
  }
}
```

RISC-V Assembly Code (scalar)

```asm
// a0: ad1, a1: ad2,
// a2: as1, a3: as2, a4: s
func:
  li t0, 0 // i=0
  li t1, 8
loop:
  lw t2, 0(a2)
  add t4, t2, a4
  lw t3, 0(a3)
  add t4, t4, t3
  sw t4, 0(a0)
  sub t4, t2, t3
  sw t4, 0(a1)
  addi a0, a0, 4
  addi a1, a1, 4
  addi a2, a2, 4
  addi a3, a3, 4
  addi t0, t0, 4
  bne t1, t0, loop
  ret
```

Given the following RISC-V VLIW pipeline with static dual issue, where slot-1 can execute ALU/branch instructions and slot-2 can execute load/store instructions:



a) Perform loop unrolling, optimizing the assembly code of the loop of the function for execution on the VLIW pipeline.

Solution: Remove loop, since N is statically 2. So the loop only runs two times.

1) Remove Loop:

```asm
// a0: ad1, a1: ad2, a2: as1, a3: as2, a4: s
func:
  lw t2, 0(a2)
  add t4, t2, a4
  lw t3, 0(a3)
  add t4, t4, t3
  sw t4, 0(a0)
  sub t4, t2, t3

  lw t2, 4(a2)
  add t4, t2, a4
  lw t3, 4(a3)
  add t4, t4, t3
  sw t4, 4(a0)
  sub t4, t2, t3
  sw t4, 4(a1)
  ret
```

2) Change registers to remove double use

```asm
// a0: ad1, a1: ad2, a2: as1, a3: as2, a4: s
func:
  lw t2, 0(a2)
  add t4, t2, a4
  lw t3, 0(a3)
  add t4, t4, t3
  sw t4, 0(a0)
  sub t4, t2, t3

  lw t5, 4(a2)
  add t6, t5, a4
  lw t7, 4(a3)
  add t6, t6, t7
  sw t6, 4(a0)
  sub t6, t5, t7
  sw t6, 4(a1)
  ret
```

3) Reorder instructions to remove data dependencies

```asm
// a0: ad1, a1: ad2, a2: as1, a3: as2, a4: s
func:
  lw t0, 0(a2)
  lw t1, 0(a3)
  lw t4, 4(a2)
  lw t5, 4(a3)

  add t2, t0, a4
  add t2, t2, t1
  sw t2, 0(a0)
  sub t3, t0, t1
  sw t3, 0(a1)

  add t6, t4, a4
  add t6, t6, t5
  sw t6, 4(a0)
  sub t7, t4, t5
  sw t7, 4(a1)
  ret
```

b) Schedule the assembly code and create the schedule for the static dual issue RISC-V pipeline specified above by filling in the table.

Solution:

Slot 1: ALU/Branch	Slot 2: Load/Store
nop	lw t0,0(a2)
nop	lw t1,0(a3)
add t4,t0,a4	lw t2,4(a2)
add t4,t4,t1	lw t3,4(a3)
sub t5,t0,t1	sw t4,0(a0)
add t4,t2,a4	sw t5,0(a1)
add t4,t4,t3	nop
sub t5,t2,t3	sw t4,0(a0)
ret	sw t5,0(a1)

6.2 Superscalar

Assume the simple superscalar RISC-V pipeline of the script with dual fetch, decode and issue, as well as forwarding. The size of the issue buffer is 8. There is a ROB and Commits (CO) should be in-order (multi-commits in same cycle allowed).

The following functional units are available:

- **ALU, ADD**: 1 cycle latency; 1 cycle interval
- **MUL**: 2 cycles latency; 1 cycle interval (pipelined)
- **DIV**: 4 cycles latency; 4 cycles interval (serial)
- **LSU**:
  - **LU**: 2 cycles latency; 1 cycle interval (non-blocking)
  - **SU**: 1 cycle latency, Store Buffer; 1 cycle interval

Dependencies are handled as follows **without** register renaming:

- RAW hazards are handled by the scoreboard. The instruction can be issued when all previous instructions with RAW dependency are at least in their finish state (last cycle of execute), hence, values are ready to be forwarded or available in the register file.
- WAR hazards are resolved by the scoreboard. The instruction can be issued when all previous instructions with WAR dependency are at least in their RO state (cycle before execute).
- WAW hazards are resolved by a ROB. Instructions can only be committed one cycle after all previous instructions with WAW dependency committed.

a) Execute the following code. Draw the pipeline diagram and calculate the achieved IPC value (instructions per clock cycle).

```asm
1 lw a1, 0(a0)
2 div a2, a4, a1  RAW to line 1
3 sw a2, 4(a0)  RAW to line 2
4 addi a3, a4, 5
5 slli a4, a3, 5  WAR and RAW to line 4
6 mul a4, a4, a7  RAW and WAW to line 5
```

Solution:

Cycle	1	2	3	4	5	6	7	8	9	10	11	12
`lw a1, 0(a0)`	IF	IS	RO	LU	LU	WB	CO					
`div a2, a4, a1`	IF	IS	IB	IB	RO	DIV	DIV	DIV	DIV	WB	CO	
`sw a2, 4(a0)`		IF	IS	IB	IB	IB	IB	IB	RO	SU	SB	
`addi a3, a4, 5`		IF	IS	RO	ADD	WB	ROB	ROB	ROB	ROB	CO	
`slli a4, a3, 5`			IF	IS	RO	ADD	WB	ROB	ROB	ROB	CO	
`mul a4, a4, a7`			IF	IS	IB	RO	MUL	MUL	WB	ROB	ROB	CO

i Note

The `ROB`s for the `addi` and `slli` are there because of interrupts. If the code gets interrupted/throws an exception in e.g. the `sw` instr., the value of `addi` and `slli` must not be written.

The `ROB` in the last line is because of the `WAW` hazard to the `slli` before it.

$$\text{IPC} = \frac{12}{6} = 2$$

7 Caches und Memories

7.1 Cache Structure

The cache system should be capable of addressing at least 42 MiB of memory. Each data word is 4 bytes in size, and the cache itself has a total capacity of 8 KiB, organized into blocks of 256 bytes each. Determine the number of address bits required with word addressing.

a) Analyze the following three cache configurations:
- Direct-Mapped Cache
- 4-Way Set Associative Cache
- Fully Set Associative Cache

For each configuration, determine the minimum number of bits required for the **Tag**, **Index**, and **Offset**, as well as the **total address length**.

Solution:

Type	Tag	Index	Offset	addr. length
Direct-mapped	13	5	6	24
4-Way set ass.	15	3	6	24
Fully ass.	18	0	6	24

Explanations:

addr. length Is always the same. The next power of 2 from 42 is $64 = 2^6$. 1 MiB takes 20 bit to address. Therefore we need 26 bit for byte addressing, and 24 bit for word addressing.

Offset Is always the same, depends on the cache blocksize. Used to index into the 256 byte blocks in the cache. To address 256 bytes, we need 8 bits, therefore we need 6 bit for word addressing.

Index Bits needed to address one "block of memory" in the cache.

Direct-Mapped Every 256 byte block gets its own address $\implies 13 - 8 = 5$ bit (13 to address the 8 KiB cache, remove 8 to address the 256 byte block)

4-Way set ass. 4 of the 256 byte blocks share one address $\implies 13 - 8 - 2 = 3$ bit

Fully ass. There is only one set of memory $\implies$ we need no address to select.

Tag the rest, so Tag+Index+Offset=addr. length

b) For the addresses 0x000DEADB and 0x000BE33F, compute the corresponding values of Tag, Index, and Offset for both the Direct-Mapped and 4-Way Set Associative cache configurations. Provide your answers in both binary and hexadecimal formats, omitting leading zeros where applicable.

Solution:

Type	Full	Tag	Index	Offset
Direct-mapped	`0b0000110111101|01011|011011`	`0x1BD`	`0xB`	`0x1B`
4-Way set ass.	`0b000011011110101|011|011011`	`0x6F5`	`0x3`	`0x1B`
Direct-mapped	`0b0000101111100|01100|111111`	`0x17C`	`0xC`	`0x3F`
4-Way set ass.	`0b000010111110001|100|111111`	`0x5F1`	`0x4`	`0x3F`

7.2 Caching Hierachy

The processor used in this task has the following specification:
- Clock Frequency: 2 GHz
- Access time to L1-Cache: 3 Cycles

- Access time to L2-Cache: 30 Cycles
- Access time to L3-Cache: 300 Cycles
- Access time to main memory: 1000 Cycles

Hint: The access time of the L2 cache already includes the access time incurred by an L1 cache miss. Similarly, the access time of the L3 cache includes the time taken to miss in both L1 and L2 caches. This same principle applies to main memory access time, which includes the cumulative delay of misses in all cache levels.

a) Assume the CPU performs 600,000 memory accesses. The hit rate (h) is 90% for the L1 cache, and 80% for both the L2 and L3 caches. Calculate the miss rate and the access time for each individual memory module.

Solution:

Layer	Miss rate	Misses	Access time
L1	10%	60000	1.5ns
L2	20%	12000	15ns
L3	20%	2400	150ns
main	0% xD	0	500ns

b) Discuss the impact of cache misses at different levels on overall system performance.

By what factor does the effective memory access time improve when using an L2 cache in addition to an L1 cache, compared to using only the L1 cache? How does the introduction of a third-level (L3) cache affect this?

To calculate this, determine $t_{\text{eff}_{\text{L1}}}$ for the scenario without an L2/L3 cache and $t_{\text{eff}_{\text{L2}}}$ for the scenario with a L1 and a L2 Cache, but no L3-Cache. $t_{\text{eff}_{\text{L3}}}$ represent the time of all 3 utilized cache levels.

To analyze this, calculate:
- $t_{\text{eff}_{\text{L1}}}$: the effective memory access time when only an L1 cache is used (no L2 or L3).
- $t_{\text{eff}_{\text{L2}}}$: the effective access time when both L1 and L2 caches are used, but no L3 cache.
- $t_{\text{eff}_{\text{L3}}}$: the effective access time when all three cache levels are utilized.

Use these values to compare performance improvements and the contribution of each cache level.

Hint: Use this formula to calculate the AMAT (average memory access time):

$$t_{\text{eff}} = h \cdot t_{\text{cache}} + (1 - h) \cdot t_{\text{main}}$$

Solution:

- $t_{\text{eff}_{\text{L1}}} = h_{\text{L1}} \cdot t_{\text{L1}} + (1 - h_{\text{L1}}) \cdot t_{\text{main}} = 0.9 \cdot 1.5 + 0.1 \cdot 500 = 51.35\text{ns}$
- $t_{\text{eff}_{\text{L2}}} = h_{\text{L1}} \cdot t_{\text{L1}} + (1 - h_{\text{L1}}) \cdot (h_{\text{L2}} \cdot t_{\text{L2}} + (1 - h_{\text{L2}}) \cdot t_{\text{main}}) = 12.55\text{ns}$
- $t_{\text{eff}_{\text{L3}}} = h_{\text{L1}} \cdot t_{\text{L1}} + (1 - h_{\text{L1}}) \cdot (h_{\text{L2}} \cdot t_{\text{L2}} + (1 - h_{\text{L2}}) \cdot (h_{\text{L3}} \cdot t_{\text{L3}} + (1 - h_{\text{L3}}) \cdot t_{\text{main}})) = 6.95\text{ns}$
- speed up with L1: $\frac{500}{51.35} = 9.737$
- speed up with L1+L2: $\frac{500}{12.55} = 39.841$
- speed up with L1+L2+L3: $\frac{500}{6.95} = 71.942$
- relative speed up L1 to L2: $\frac{t_{\text{eff}_{\text{L1}}}}{t_{\text{eff}_{\text{L2}}}} = \frac{51.35\text{ns}}{12.55\text{ns}} = 4.092$
- relative speed up L1 to L3: $\frac{t_{\text{eff}_{\text{L1}}}}{t_{\text{eff}_{\text{L3}}}} = \frac{51.35\text{ns}}{6.95\text{ns}} = 7.388$
- relative speed up L2 to L3: $\frac{t_{\text{eff}_{\text{L2}}}}{t_{\text{eff}_{\text{L3}}}} = \frac{12.55\text{ns}}{6.95\text{ns}} = 1.806$

From this we can see, that more cache levels are more effective (no shit sherlock...)

c) Assume a 2-level cache hierarchy where the L2 cache has significantly greater capacity than the L1 cache. Can the hit rate of the L1 cache even be higher than that of the L2 cache?

If so, explain elaborate multiple reasons.

Solution: Yes this is possible.

For example: Build a program, that runs in a loop and completely fits into the L1 cache. At some points in the program, it jumps to a random address and returns back into the loop.

This means, that the L1 cache has a hitrate of almost 1, except for the random jumps. If they are rare enough, they can be neglected.

The hitrate of the L2 cache only counts with the misses from the L1 cache. Since the jumps are to random positions, the data is also not in the L2 cache, which leads to a miss rate of 100%.

This example assumes, that the random jump does not kick any instruction from the L1 cache. But it can be safely assumed, that it is possible to construct such a program.

From solution document:
- Data patterns access far distant entries (Program fow)
- Sub-optimal Global Cache Configuration (Exclusive/Inclusive filling)
- Sub-optimal Local Cache Configuration (Direct/Full/Different associativity would be more beneficial)

8 RISC-V Vector (RVV) Instruction Set

8.1 Vector Pipeline Diagrams

A RISC-V Vector Unit is configured as follows:

- **VLEN** = 256 bits (register length)
- **LMUL** = 2 (number of registers per vector)
- **SEW** = 32 bits (selected element width – width of one element in the vector)
- **VL** = VLMAX (number of elements in one vector)

Vector Functional Units are designed with:

- **VADD**: 128 bits wide (4 32-bit lanes), pipelined, 3 cycles latency
- **VMUL**: 128 bits wide (4 32-bit lanes), pipelined, 5 cycles latency
- **VLSU**: 128 bits wide (4 32-bit lanes), pipelined, 4 cycles latency for loads, 2 cycles latency stores
- 1 Vector Register Write Port, 2 Vector Register Read Ports. On conflict, oldest instruction gets priority.

Given the following RISC-V Vector Code:

```asm
vle32.v v0, (a1)
vmul.vx v2, v0, t0
vadd.vv v4, v0, v2
vse32.v v4, (a2)
```

Draw the pipeline state of the vector unit for this code. Assume this vector unit allows for chaining results between vector functional units.

$$\text{VLMAX} = \frac{\text{LMUL} \cdot \text{VLEN}}{\text{SEW}} = 16$$

Solution:

Instruction	Register	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
`vle32.v v0, (a1)`	`v0[0-3]`	R	VLSU	VLSU	W															
`vle32.v v0, (a1)`	`v0[4-7]`		R	VLSU	VLSU	W														
`vle32.v v0, (a1)`	`v1[0-3]`			R	VLSU	VLSU	W													
`vle32.v v0, (a1)`	`v1[4-7]`				R	VLSU	VLSU	W												
`vmul.vx v2, v0, t0`	`v0[0-3]`					R	VMUL	VMUL	VMUL	W										
`vmul.vx v2, v0, t0`	`v0[4-7]`						R	VMUL	VMUL	VMUL	W									
`vmul.vx v2, v0, t0`	`v1[0-3]`							R	VMUL	VMUL	VMUL	W								
`vmul.vx v2, v0, t0`	`v1[4-7]`								R	VMUL	VMUL	VMUL	W							
`vadd.vv v4, v0, v2`	`v0[0-3]`										R	VADD	stall	W						
`vadd.vv v4, v0, v2`	`v0[4-7]`											R	stall	VADD	W					
`vadd.vv v4, v0, v2`	`v1[0-3]`												stall	R	VADD	W				
`vadd.vv v4, v0, v2`	`v1[4-7]`														R	VADD	W			
`vse32.v v4, (a2)`	`v0[0-3]`														stall	R	VLSU			
`vse32.v v4, (a2)`	`v0[4-7]`																R	VLSU		
`vse32.v v4, (a2)`	`v1[0-3]`																	R	VLSU	
`vse32.v v4, (a2)`	`v1[4-7]`																		R	VLSU

8.2 RISC-V Vector Configuration

The RISC-V vector unit configuration is as follows:

- Vector Register Parameters:
  - **VLEN**: 256 Bits
  - **LMUL**: 4
  - **SEW**: 32 Bits
- Input Data:
  - Array a
  - Data Type: int (32 Bits)
  - Data Length n: 122

Solve the following problems:

a) Calculate VLMAX

Solution:

$$\text{VLMAX} = \frac{\text{VLEN} \cdot \text{LMUL}}{\text{SEW}} = 32$$

b) Determine the number of iterations required to process 122 elements.

Solution:

$$\left\lceil \frac{122}{\text{VLMAX}} \right\rceil = 4$$

c) In the last iteration, illustrate the data layout in the vector registers based on the given configuration and dynamic vector length (**vl**). Label the key parameters (**SEW**, **LMUL**, **VLEN**, **VLMAX**, **vl**).

Solution:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
v0																																
v1																																
v2																																
v3																																

All the orange entries are filled with data, the white ones are not.

- **SEW** is 4, which is the width of one entry (one cell of the table)
- **LMUL** is 4, which is the number of bundled registers, here the number of lines in the table.
- **VLEN** is the register width, which is 256 bit or 32 bytes, here the width of the table
- **VLMAX** is the maximum number of entries that fit into one vector. Here the number of cells (including the not-colored ones)
- **vl** is the current number of entries that fit into one vector, which is 26. Here the number of colored cells.

8.3 Vectorized Assembly Code

Given the following assembly code:

```asm
    addi t0, x0, 100
  loop:
    lw t1, 0(a0)
    lw t2, 0(a1)
    mul t3, t1, t2
    sw t3, 0(a2)
    addi t0, t0, -1
    addi a0, a0, 4
    addi a1, a1, 4
    addi a2, a2, 4
    bne x0, t0, loop
    ret
```

Write a version of this code using scalable RISC-V Vector assembly instructions.

Solution:

```C
for (int i = 0; i < 100; ++i) {
  a2[i] = a0[i] * a1[i];
}
```

Therefore this can be parallelized with vectorized assembly instructions.

```asm
    addi t0, x0, 100
  loop:
    vsetvli t1, t0, e32, m2, ta, ma
    slli t3, t1, 2    // new loop handles VLMAXx2 elements
    vle32.v v0, (a0)
    vle32.v v2, (a1)
    vmul.vv v4, v0, v2
    vse32.v v4, (a2)
    sub t0, t0, t1
    add a0, a0, t3
    add a1, a1, t3
    add a2, a2, t3
    bne x0, t0, loop
    ret
```

How could this code be improved to make use of interleaving on the vector unit? Modify your written code to use this principle.

Solution: Yes, it can be improved:

```asm
    vsetvli t2, x0, 32, m2, ta, ma #Get VLMAX
    slli t2, t2, 1        // new loop handles VLMAXx2 elements
    addi t0, x0, 100
loop:
    blt t0, t2, tail
    vsetvli t1, t0, e32, m2, ta, ma
    slli t3, t1, 2
    vle32.v v0, (a0)
    add a0, a0, t3
    vle32.v v2, (a1)
    add a1, a1, t3
    vmul.vv v4, v0, v2
    vle32.v v6, (a0)      //interleave 2nd load with 1st multiply
    add a0, a0, t3
    vle32.v v8, (a1)
    add a1, a1, t3
    vmul.vv v10, v6, v8
    vse32.v v4, (a2)      // interleave 1st store with 2nd multiply
    add a2, a2, t3
    vse32.v v10, (a2)
    add a2, a2, t3
    sub t0, t0, t2
    bne x0, t0, loop
    tail: beq t0, x0, exit // if vector fits perfectly in loop
    vsetvli t1, t0, e32, m4, ta, ma // use old code here
    slli t3, t1, 2
    vle32.v v0, (a0)
    vle32.v v2, (a1)
    vmul.vv v4, v0, v2
    vse32.v v4, (a2)
    sub t0, t0, t1
    add a0, a0, t3
    add a1, a1, t3
    add a2, a2, t3
    bne x0, t0, tail
exit:
    ret
```

9 HLS Basics

Given is the following function to compute the estimate the square root of the input parameter x. If the input parameter is negative, then the function should first multiply the input x with −1 to make it positive.

```c
double mysqrt(double x) {
  if (x < 0) {
    x = x * -1;
  }
  double y = 0.22222 + 0.888889 * x;
  int i = 0;
  while (i < 3) {
    y = 0.5 * (y + x / y);
    i = i + 1;
  }
  return y;
}
```

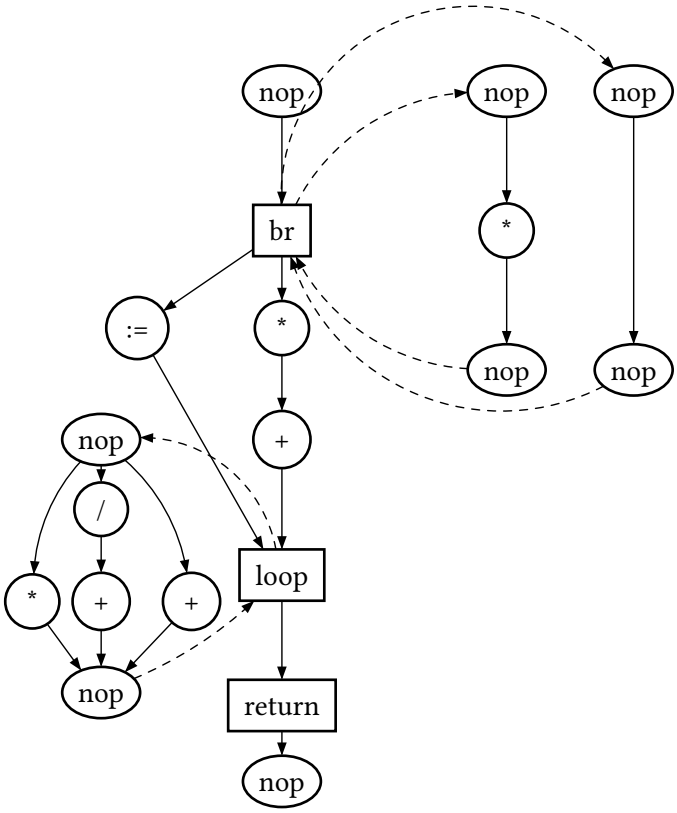
a) Give an intermediate code representation using three-address code notation.

Solution:

```
    if x>=0 goto endif1
    x := x * -1
endif1:
    t1 := 0.888889 * x
    y := 0.22222 + t1
    i := 0;
whilestart:
    t2 := x / y
    t3 := t2 + y
    y  := 0.5 * y
    i  := i + 1
    if i < 3 goto whilestart
    return y
```

b) Give the sequencing graph for your intermediate code representation.

Solution:


nop
nop
nop
br
*
:=
*
nop
nop
nop
+
nop
/
loop
*
+
+
nop
return
nop

10 HLS Scheduling

We look at the following code line of the XTEA encryption algorithm:

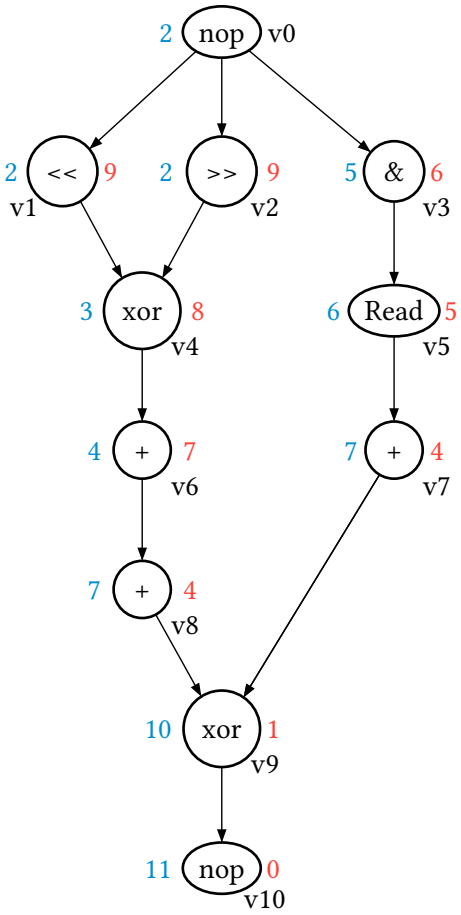
```c
y = y +(z<<4 ^ z>>5) + z ^ sum + k[sum & 3];
```

The code line should be implemented in hardware. An ALU is used, which can execute shift operations (`SL`,`SR`), logic operations (`XOR`,`AND`) and additions (`+`). The array `k` is stored in local memory. The access to the memory is done by a `READ` operation.

Given is the intermediate code representation using three-address code notation:

```
t1 := z << 4
t2 := z >> 5
t3 := sum & 3
t4 := t1 ^ t2
t5 := k[t3]
t6 := y + t4
t7 := sum + t5
t8 := t6 + z
y  := t7 ^ t8
```

The resulting Sequencing Graph is:



- red values are priorities (just bottom up times how long it takes, starting from 0)
- blue values are the ALAP times for the nodes

- The ALU supports multi-cycle operations.
- The shift and logical operations have a delay of one clock cycle
- The add operation (+) has a delay of three clock cycles
- The READ operations takes the array index as input to compute a memory address and outputs the value of the array element after one clock cycle on the memories' read bus.

a) Schedule the sequencing graph for a resource constrained (2 ALUs, 1 decoder) with the list scheduling method.

Solution:

	$2 \times$ ALU			1 decoder (for `READ`s)		Start time
t_{act}	$U_{\text{akt,alu}}$	$T_{\text{akt,alu}}$	$S_{\text{akt,alu}}$	$U_{\text{akt,dec}}$	$S_{\text{akt,dec}}$	t_i
1	$\{\text{v1}, \text{v2}, \text{v3}\}$	$\{\}$	$\{\text{v1}, \text{v2}\}$	$\{\}$	$\{\}$	$t_1 = t_2 = 1$
2	$\{\text{v3}, \text{v4}\}$	$\{\}$	$\{\text{v3}, \text{v4}\}$	$\{\}$	$\{\}$	$t_3 = t_4 = 2$
3	$\{\text{v6}\}$	$\{\}$	$\{\text{v6}\}$	$\{\text{v5}\}$	$\{\text{v5}\}$	$t_6 = t_5 = 3$
4	$\{\text{v7}\}$	$\{\text{v6}\}$	$\{\text{v7}\}$	$\{\}$	$\{\}$	$t_7 = 4$
5	$\{\}$	$\{\text{v6}, \text{v7}\}$	$\{\}$	$\{\}$	$\{\}$	
6	$\{\text{v8}\}$	$\{\text{v7}\}$	$\{\text{v8}\}$	$\{\}$	$\{\}$	$t_8 = 6$
7	$\{\}$	$\{\text{v8}\}$	$\{\}$	$\{\}$	$\{\}$	
8	$\{\}$	$\{\text{v8}\}$	$\{\}$	$\{\}$	$\{\}$	
9	$\{\text{v9}\}$	$\{\}$	$\{\text{v9}\}$	$\{\}$	$\{\}$	$t_9 = 9$
10	$\{\text{v10}\}$	$\{\}$	$\{\text{v10}\}$	$\{\}$	$\{\}$	$t_{10} = 10$

b) Schedule the sequencing graph with the ASAP schedule and give the number of required resources.

Solution:
- $t_1 = t_2 = t_3 = 1$
- $t_4 = t_5 = 2$
- $t_6 = t_7 = 3$
- $t_8 = 6$
- $t_9 = 9$
- $t_{10} = 10$

$\Lambda^S = 9$ (latency)

Resources: 3 ALUs, 1 decoder

c) Schedule the sequencing graph with the ALAP schedule for a latency of 10 clock cycles.

Solution:
- $t_{10} = 11$
- $t_9 = 10$
- $t_8 = t_7 = 7$
- $t_5 = 6$
- $t_3 = 5$
- $t_6 = 4$
- $t_4 = 3$
- $t_1 = t_2 = 2$
- $t_0 = 2$

d) Give the mobility of the operations and the critical path for a latency of 10 clock cycles.

Solution:

- $\mu_3 = \mu_5 = \mu_7 = 4$
- $\mu_0 = \mu_1 = \mu_2 = \mu_4 = \mu_6 = \mu_8 = \mu_9 = \mu_{10} = 1$

Critical path: v0 $\longrightarrow$ v1 | v2 $\longrightarrow$ v4 $\longrightarrow$ v6 $\longrightarrow$ v8 $\longrightarrow$ v9 $\longrightarrow$ v10

e) Schedule the sequencing graph with the list scheduling method and a timing constraint (maximal 10 clock cycles latency).

Solution:

	1 ALU						1 decoder (for `READ`s)					Start time
t_{act}	$U_{\text{akt,alu}}$	Slack	$T_{\text{akt,alu}}$	$S_{\text{akt,alu}}$	$R_{\text{akt,alu}}$	a_{alu}	$U_{\text{akt,dec}}$	Slack	$S_{\text{akt,dec}}$	$R_{\text{akt,alu}}$	a_{dec}	t_i
1	$\{\text{v1}, \text{v2}, \text{v3}\}$	$s_1 = s_2 = 1; s_3 = 4$	$\{\}$	$\{\}$	$\{\text{v1}\}$	1	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$t_1 = 1$
2	$\{\text{v2}, \text{v3}\}$	$s_2 = 0; s_3 = 3$	$\{\}$	$\{\text{v2}\}$	$\{\}$	1	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$t_2 = 2$
3	$\{\text{v3}, \text{v4}\}$	$s_3 = 2, s_4 = 1$	$\{\}$	$\{\}$	$\{\text{v4}\}$	1	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$t_4 = 3$
4	$\{\text{v3}, \text{v6}\}$	$s_3 = 1, s_6 = 0$	$\{\}$	$\{\text{v6}\}$	$\{\}$	1	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$t_6 = 4$
4	$\{\text{v3}\}$	$s_3 = 0$	$\{\text{v6}\}$	$\{\text{v3}\}$	$\{\}$	2	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$t_3 = 5$

The last line is not possible with one ALU, since v6 is running and we need to add v3. By the algorithm, we would now restart with 2 ALUs. We already did that in point a) and proved that it is possible to remain below 10 cycles of latency, so we do not have to continue here.

- t_{act} current time
- $U_{\text{akt,alu}}$ Candidates to schedule
- Slack: $t_{\text{ALAP}} - t_{\text{act}}$
- $T_{\text{akt,alu}}$ currently running
- $S_{\text{akt,alu}}$ now starting (with **slack 0**)
- $R_{\text{akt,alu}}$ now starting (with **not slack 0**)
- a_{alu} number of availible ALUs
- t_i starting time of vi

11 HLS Binding and RTL Generation

Given is the following c-program:

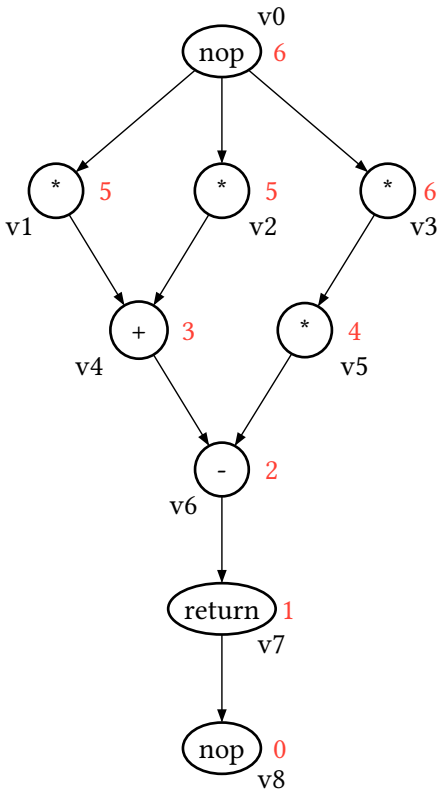
```c
int32_t calc2(int32_t a, int32_t b, int32_t c) {
  int32_t d;
  d = a * a + b * b - 3 * a * c;
  return d;
}
```

The three address code for this code line is given as follows:

```
calc2:
  t1:=a*a
  t2:=b*b
  t3:=a*c
  t4:=3*t3
  t5:=t1+t2
  d:=t5-t4
  return d
```

a) Draw the sequencing graph for this three address code.

Solution:



b) Compute the start times for all operations using the list schedule method for a resource constraint of one ALU that can execute additions and subtractions and two multipliers that can execute multiplications. The delay of the ALU is one clock cycle and the delay of the multiplier is two clock cycles.

Solution:

	2 MUL			1 ALU		Start time
t_{act}	$U_{\text{akt,mul}}$	$T_{\text{akt,mul}}$	$S_{\text{akt,mul}}$	$U_{\text{akt,alu}}$	$S_{\text{akt,alu}}$	t_i
1	{v1, v2, v3}	{}	{v1, v3}	{}	{}	$t_1 = t_3 = 1$
2	{v2}	{v1, v3}	{}	{}	{}	
3	{v2, v5}	{}	{v2, v5}	{}	{}	$t_2 = t_5 = 3$
4	{}	{v2, v5}	{}	{}	{}	
5	{}	{}	{}	{v4}	{v4}	$t_4 = 5$
6	{}	{}	{}	{v6}	{v6}	$t_6 = 6$
7	{}	{}	{}	{v7}	{v7}	$t_7 = 7$
8	{}	{}	{}	{}	{}	$t_8 = 8$

c) Find a valid binding of the multiplications to the multipliers using the left edge algorithm.

Solution: Find the multiplication intervals:

- $I_1 = [1, 2]$
- $I_2 = [3, 4]$
- $I_3 = [1, 2]$
- $I_5 = [3, 4]$

Sort by left sides of the intervals (the starting points): I_1, I_3, I_2, I_4

- $a_{\text{act}} = 0 \longrightarrow l_i \geq a_{\text{act}}$: choose I_1
- $a_{\text{act}} = r_1 = 2 \longrightarrow l_i \geq a_{\text{act}}$: choose I_2
- $a_{\text{act}} = r_2 = 4 \longrightarrow l_i \geq a_{\text{act}}$: there is none left

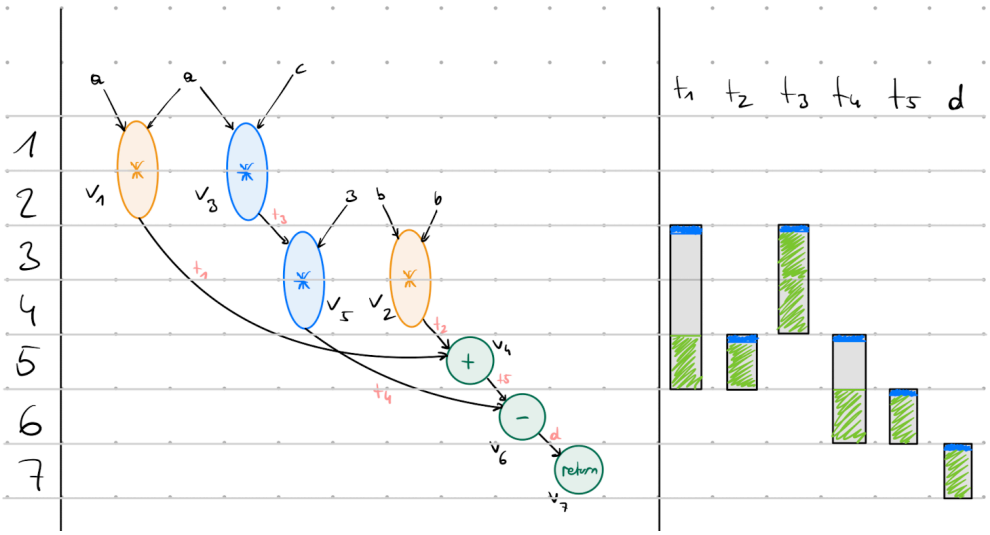
Try again for the other multiplier with set $\{I_3, I_4\}$:

- $a_{\text{act}} = 0 \longrightarrow l_i \geq a_{\text{act}}$: choose I_3
- $a_{\text{act}} = r_3 = 2 \longrightarrow l_i \geq a_{\text{act}}$: choose I_4
- $a_{\text{act}} = r_4 = 4 \longrightarrow l_i \geq a_{\text{act}}$: there is none left

So the bindings are $\{I_1, I_2\}$ on the first multiplier and $\{I_3, I_4\}$ on the second one.

d) Give the lifetime of all compiler temporary variables `tx`. Draw the register conflict graph for these variables `tx`.

Solution:



Reg 1 t_1, T_5, d
Reg 2 t_2, t_3
Reg 3 t_4

e) Find a valid binding of the compiler temporary variables to registers using the left edge algorithm. How many registers are required? Color the register conflict graph accordingly.

Solution:

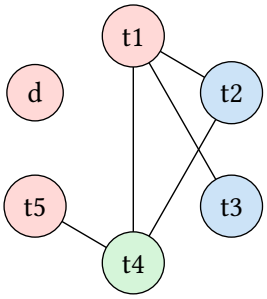
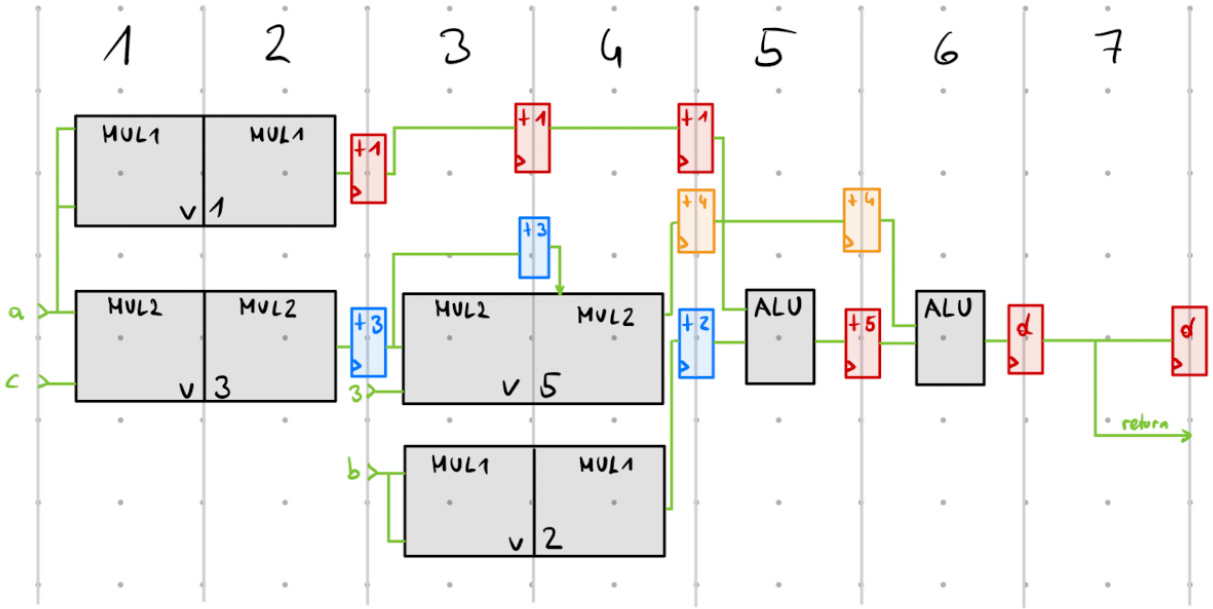
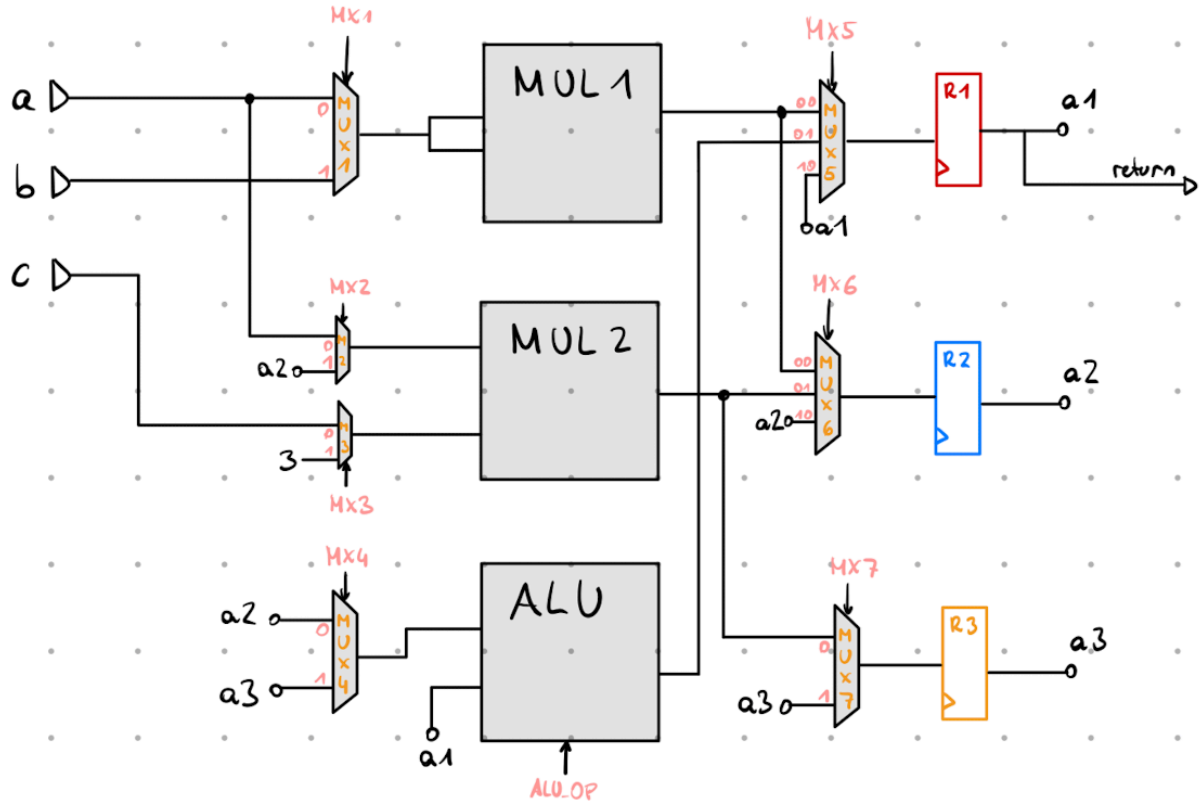


Figure 8: Conflict graph

f) Draw the data flow graph with schedule and binding.



g) Draw a structural view of the data path.



h) Give the activation signals for all operations.

Solution:

Note	Operation	Binding	Cycle	MX1	MX2	MX3	MX4	MX5	MX6	MX7	ALU_OP
v1	t1 := a*a	MUL1	1	0	-	-	-	-	-	-	
v1	t1 := a*a	MUL1	2	0	-	-	-	00	-	-	
v2	t2 := b*b	MUL1	1	1	-	-	-	-	-	-	
v2	t2 := b*b	MUL1	2	1	-	-	-	-	00	-	
v3	t3 := a*c	MUL2	1	-	0	0	-	-	-	-	
v3	t3 := a*c	MUL2	2	-	0	0	-	-	01	-	
v4	t5 := t1+t2	ALU	1	-	-	-	0	01	-	-	+
v5	t4 := 3*t3	MUL2	1	-	1	1	-	-	-	-	
v5	t4 := 3*t3	MUL2	2	-	1	1	-	-	-	0	
v6	d := t5-t4	ALU	1	-	-	-	1	01	-	-	−

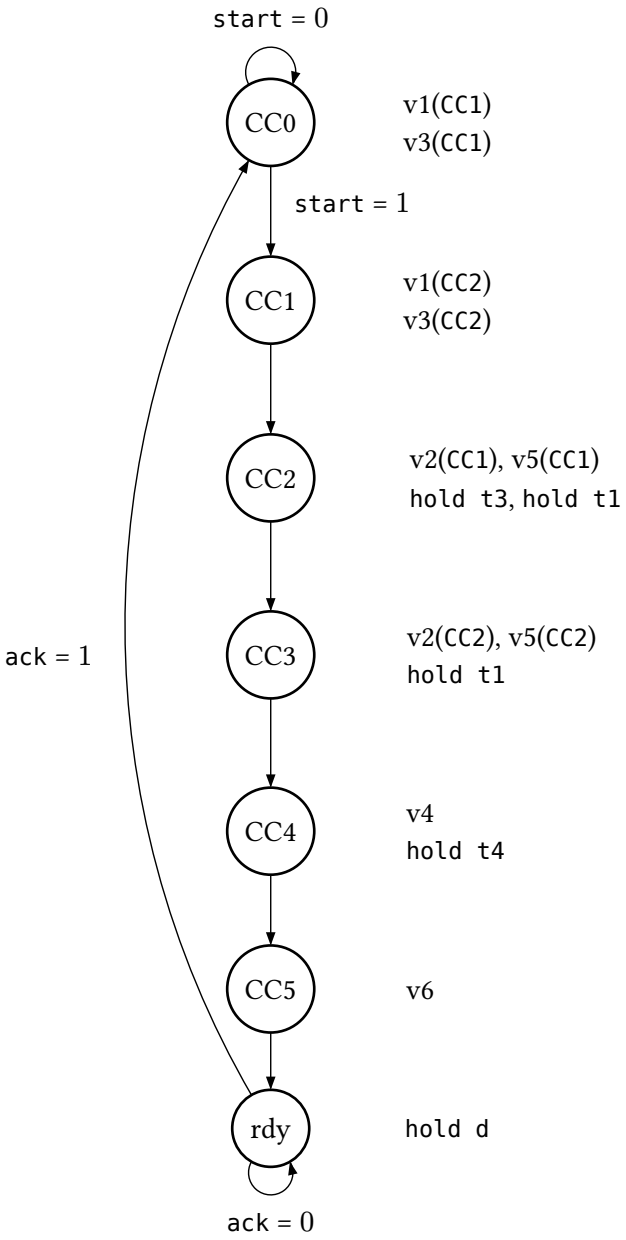
i) Give the hold signals for all variables, for which they are required.

Solution:

Operation	Binding	MX1	MX2	MX3	MX4	MX5	MX6	MX7
hold t1	R1	-	-	-	-	10	-	-
hold t3	R2	-	-	-	-	-	10	-
hold t4	R3	-	-	-	-	-	-	1
hold d	R1	-	-	-	-	10	-	-

j) Draw the FSM with data specification to control the data path. Apply a start, ready, acknowledge control scheme. The value of `d` should be kept at its output port for readout until the acknowledge signal is high.

Solution:



i Note

- The state transitions that are not given explicitly are done on every clock edge.
- There should be a state transition from each state to `CC0` if `reset` is high.
- The syntax v1(`CC1`) means that in this state, the multiplexer signals from state v1 in clock cycle 1 should be applied.

k) Conduct a state assignment for your FSM applying binary encoding.

Solution:

State	Binary encoding	One-hot encoding	Almost-one-hot encoding
CC0	000	0000001	000000
CC1	001	0000010	000001
CC2	010	0000100	000010
CC3	011	0001000	000100
CC4	100	0010000	001000
CC5	101	0100000	010000
rdy	110	1000000	100000

l) Give the truth table for the output logic of your FSM.

Solution:

State	MX1	MX2	MX3	MX4	MX5	MX6	MX7	ALU_OP	ready
CC0 (000)	0	0	0	-	-	-	-		0
CC1 (001)	0	0	0	-	00	01	-		0
CC2 (010)	1	1	1	-	10	10	-		0
CC3 (011)	1	1	1	-	10	00	0		0
CC4 (100)	-	-	-	0	01	-	1	+	0
CC5 (101)	-	-	-	1	01	-	-	−	0
rdy (110)	-	-	-	-	10	-	-		1

m) Give the truth table for the next state logic of your FSM. Include a reset signal that brings the FSM from any state to its initial state.

Solution:

State	start	ack	reset	next State
000	0	X	0	000
000	1	X	0	001
001	X	X	0	010
010	X	X	0	011
011	X	X	0	100
100	X	X	0	101
101	X	X	0	110
110	X	0	0	110
110	X	1	0	000
XXX	X	X	1	000

12 HLS Loop Optimization

A FIR filter with an input sample vector x, coefficient vector c and output vector y should be designed. Given is the following C-function that implements the FIR filter:

```c
void fir(int x[703], int c[64], int y[640]) {
  int i, j;
  for (j = 0; j < 640; j++){
    y[j] = 0;
    for (i=0;i<64;i++){
      y[j] = y[j] + c[i] * x[i + j];
    }
  }
}
```

The arrays x,c and y are all stored in local dedicated memories. Read operations to the memories take two cycles, write operations one cycle. The multiplication takes six cycles. All other operations take one cycle. *Attention: the first cycle of a read operation can be chained in one cycle together with an add operation! The read operation is pipelined, such that one can use the read port every cycle, but the result is returned in the next cycle.*

12.1 Solution 1:

At first we implement the FIR filter without any loop optimization: Its intermediate representation in three-address code notation is as follows:

```
fir:
  j:=0
Loop1:
  y[j]:=0
  i:=0
Loop2:
  t1:=i+j
  t2:=x[t1]
  t3:=c[i]
  t4:=t2*t3
  t5:=y[j]
  t6:=t5+t4
  y[j]:=t6
  i:=i+1
  if (i<64) goto Loop2
  j:=j+1
  if(j<640) goto Loop1
  return
```

a) Can this three-address code further be optimized? If yes, give the optimized code.
b) Draw the sequencing graph for this intermediate representation of the fir function.
c) Schedule the Sequencing Graph using ASAP Scheduling. Compute the latency for the whole fir function. Give the number of adders, multipliers, read ports for x and c as well as write ports for y.

Solution:

12 Solution - HLS Loop Optimization

a) Optimized code:

```
1  fir:
2    j:=0 // v1
3  Loop1:
4    t1:=j // v6
5    t2:=0 // v7
6    i:=0 // v8
7  Loop2:
8    t3:=i+t1 //v15
9    t4:=x[t3] //v17
10   t5:=c[i] //v16
11   t6:=t4*t5 //v20
12   t2:=t2+t6 //v21
13   i:=i+1 //v18
14   if (i<64) goto Loop2 //v19
15   y[t1]:=t2 //v10
16   j:=j+1 //v11
17   if(j<640) goto Loop1 //v12
18   return //v3
```

b) Sequencing Graph

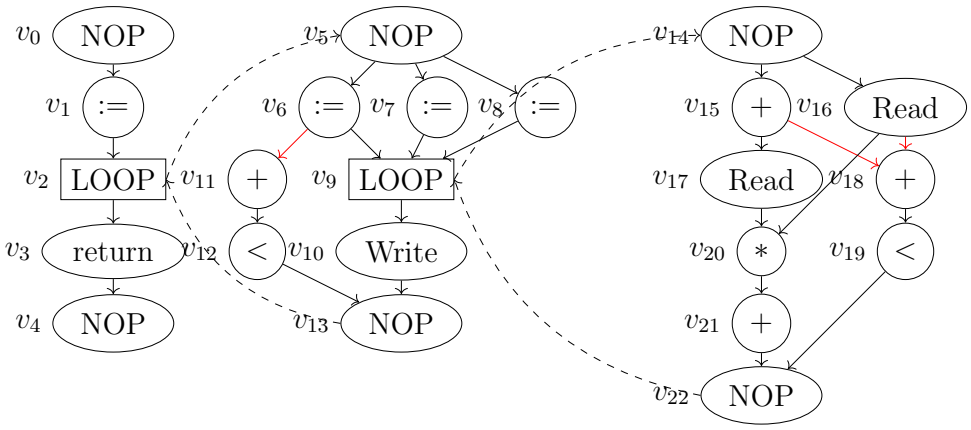


Figure 16: Sequencing graph

c) ASAP Scheduling (Most Inner Loop 2)
$\hat{t}^S_{14}$=1
$\hat{t}^S_{15}$=1, $\hat{t}^S_{16}$=1
$\hat{t}^S_{17}$=1 (chained with v_{15})
$\hat{t}^S_{20}$=3, $\hat{t}^S_{18}$=3
$\hat{t}^S_{19}$=4
$\hat{t}^S_{21}$=9
$\hat{t}^S_{21}$=10

$\Lambda_{Loop2} = 9$
Execution time of Loop node: $d_9 = \Lambda_{Loop2} \cdot$ #iterations $= 9 \cdot 64 = 576$

ASAP Scheduling (Most Inner Loop 1)
$\hat{t}^S_5$=1
$\hat{t}^S_6$=1, $\hat{t}^S_7$=1, $\hat{t}^S_8$=1
$\hat{t}^S_{11}$=2, $\hat{t}^S_9$=2
$\hat{t}^S_{12}$=3
$\hat{t}^S_{10}$=578
$\hat{t}^S_{13}$=579
$\Lambda_{Loop1} = 578$
Execution time of Loop node: $d_9 = \Lambda_{Loop1} \cdot$ #iterations $= 578 \cdot 640 = 369920$

ASAP Scheduling of top FIR SGU
$\hat{t}^S_0$=1
$\hat{t}^S_1$=1
$\hat{t}^S_2$=2
$\hat{t}^S_3$=369922
$\hat{t}^S_4$=369923
Λ_{FIR} =369922

We need Λ_{FIR} =369922 cycles and one resource of each type (two ADD as v_{11} and v_{15} would run in same cycle in this schedule).

12.2 Solution 2

We unroll the inner loop completely to optimize the FIR filter for a resource constrained case: at maximum two read ports may be used for x and two read ports may be used for c.

```c
void firUnrolled1(int x[703], int c[64], int y[640]) {
  int j;
  for (j = 0; j < 640; j++) {
    y[j] = 0;
    y[j] = y[j] + c[0] * x[0 + j];
    y[j] = y[j] + c[1] * x[1 + j];
    y[j] = y[j] + c[2] * x[2 + j];
    (...)
    y[j] = y[j] + c[63] * x[63 + j];
  }
}
```

d) Draw the sequencing graph for this case. Assume the same optimization as in the previous solution.
e) Compute the latency of the `fir` function.

Solution:

d) Unroll the inner loop: Three-address code

```
1  fir:
2    j:=0 // v1
3  Loop1:
4    t1:=j // v6
5    // block: i=0,i=1
6    t2:=x[t1] //v7
7    t3:=c[0] //v8
8    t4:=t2*t3 //v9
9    t5:=1+t1 //v10
10   t6:=x[t5] //v11
11   t7:=c[1] //v12
12   t8:=t6*t7 //v13
13   t9:=t4+t8 //v14
14   // block i=2,i=3
15   t10:=2+t1 //v15
16   t11:=x[t10] //v16
17   t12:=c[2] //v17
18   t13:=t11*t12 //v18
19   t14:=3+t1 //v19
20   t15:=x[t14] //v20
21   t16:=c[3] //v21
22   t17:=t15*t16 //v22
23   t18:=t13+t17 //v23
24   t19:=t9+t18 //v24
25   // block: i=4,i=5
26   (...)
27
28   y[t1]:=t319 //v332
29   j:=j+1 //v333
30   if(j<640) goto Loop1 //v334
31   return //v3
```

Simplified Sequencing Graph for only four iterations (i=0,1,2,3) of the inner loop The real sequencing graph has 32 blocks for i = 0..64!

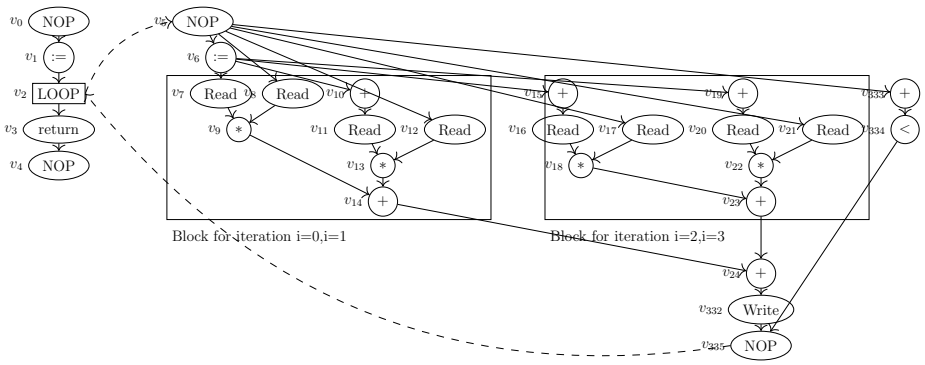


Figure 17: Sequencing graph

12.3 Solution 3

In the following, the inner loop should be pipelined. There needs to be only a single ramp-down phase at the end of the last iteration of the outer loop NOT at every iteration of the outer loop.

f) Draw the sequencing graph including the cross-carried data dependency edges.
g) What is the possible initialization interval T_p?
h) What is the latency for this schedule for the complete function with the smallest initialization interval? What are the number of required resources?

Solution:

e) Scheduling with max. two read ports for `x` and two read ports for `y`. We can schedule two blocks at the same time with the available read ports.
Hence, there are 32 blocks for the 64 inner loop iterations.
Each block has a latency of Λ_{Block}=9 cycles (2 for read, 6 for mult, 1 for add).
As the read port is pipelined, we can start each block one cycle after the previous one, leading to a quasi-pipelined schedule: The latency for all 32 blocks is hence 32 cycles to start all blocks plus 8 cycles to finsh the last block (ramp down):

$\Lambda_{AllBlocks}$ = #blocks + (Λ_{Block}-1) = 32 + 9 -1 = 40

Finally, these is one more cycle required to add the output of the last two blocks and one cycle for the write operation.

Λ_{Loop}=$\Lambda_{AllBlocks}$ +2 = 42

With this we can compute the latency of the LOOP node v_2:
$d_2 = \Lambda_{Loop} \cdot$ #iterations = $42 \cdot 640 = 26880$

The latency of the FIR filter adds 2 for node v_1 and node v_3.
$\Lambda_{FIR,unrolled}$= 26882

f) Add the cross-loop dependencies in the sequencing graph

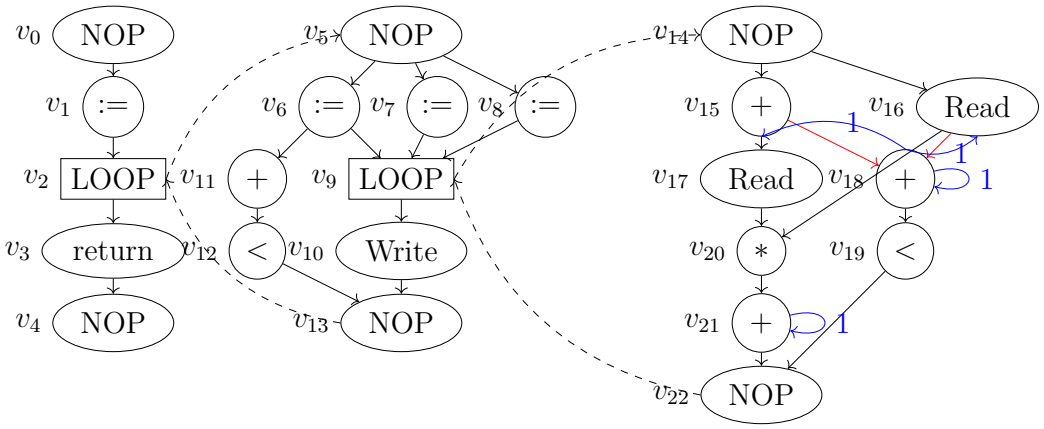


Figure 18: Sequencing graph

The pipelining starts each operation again after the initialization interval:

$$t_i^{(l+1)} = t_i^{(l)} + T_p$$

The cross-loop dependencies at new constraints:

$$t_{21}^{(l+1)} \geq t_{21}^{(l)} + d_{21}$$
$$t_{21}^{(l)} + T_p \geq t_{21}^{(l)} + d_{21}$$
$$T_p \geq d_{21}$$
$$T_p \geq 1$$

$$t_{18}^{(l+1)} \geq t_{18}^{(l)} + d_{18}$$
$$t_{18}^{(l)} + T_p \geq t_{18}^{(l)} + d_{18}$$
$$T_p \geq d_{18}$$
$$T_p \geq 1$$

$$t_{15}^{(l+1)} \geq t_{18}^{(l)} + d_{18}$$
$$t_{15}^{(l)} + T_p \geq t_{18}^{(l)} + d_{18} \text{ with } t_{15}^{(l)} = 1, t_{18}^{(l)} = 3$$
$$1 + T_p \geq 3 + 1$$
$$T_p \geq 3$$

$$t_{16}^{(l+1)} \geq t_{18}^{(l)} + d_{18}$$
$$t_{16}^{(l)} + T_p \geq t_{18}^{(l)} + d_{18} \text{ with } t_{16}^{(l)} = 1, t_{18}^{(l)} = 3$$
$$1 + T_p \geq 3 + 1$$
$$T_p \geq 3$$

So with the schedule and three address code of a) + b), we can only select an initialization interval of $T_p \geq 3$.

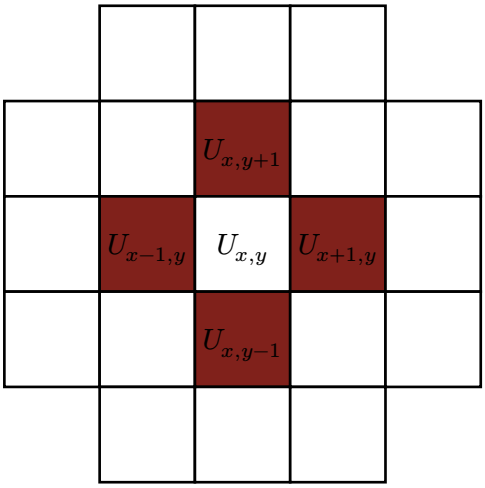
13 Multicore Synchronization Challenges

13.1 Iterative Stencil Loops (ISLs)

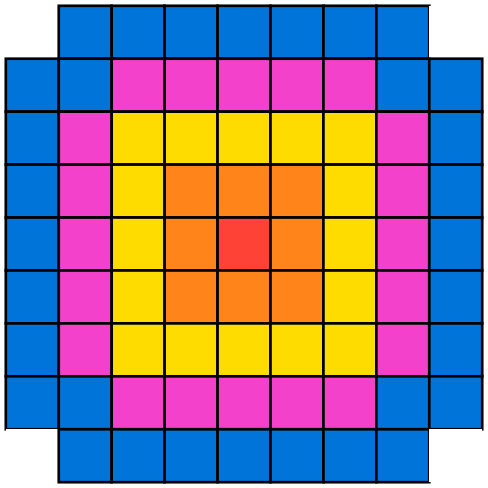
Numerical data processing with a fixed pattern, called a stencil, is most commonly used in computer simulations. ISLs perform their calculations in so-called timesteps within a given array, where each element or cell is updated. The stencil describes the access pattern to the neighboring cells utilized. We investigate in the following task the 5-Point 2D stencil to calculate the average over 4 neighbors + cell itself:

$$Z_{x,y} = \frac{1}{5} \times \left(U_{x,y} + U_{x-1,y} + U_{x+1,y} + U_{x,y-1}, U_{x,y+1}\right)$$

Figure 3a depicts the pattern for each cell in each timestep. Figure 3b illustrates a simple application designed solely for visualization purposes, demonstrating why such an access pattern might be reasonable. We assume for the following questions, that **U** and **Z** are both implemented as 2-D arrays (x and y-axis).



Subfigure 18.1: 5 Point 2D Stencil Pattern



Subfigure 18.2: HeatMap for visualization

Figure 18: 5-Point 2D stencil pattern and a simplified application example

a) Is this a do-all or do-across loop? Is there any data shared between computations?

Solution: This is an do-all loop with `U` and its elements as shared data.

b) Write pseudo-code for this task and indicate which parts can be parallelized?

```python
offsets = [(-1, 0), (1, 0), (0, -1), (0, 1), (0, 0)]
for x in range(width): # can be parallelized
  for y in range(height): # can be parallelized
    sum = 0
    count = 0
    for offset in offsets:
      sum += U[x + offset.x, y + offset.y]
      count += 1
    Z[x, y] = sum / count
```

c) Considering multi-core synchronizations, is a parallelization always correct?

Solution: Yes, since `U` and `Z` are different buffers. Therefore there are no race conditions.

d) Can there be performance bottlenecks when doing the parallelization? How can these be mitigated?

Solution: No. Since each thread/processor writes to its own memory address without having to fetch the data that is in there, there are no cache problems like false sharing.

from solution document: There can be a performance bottleneck as caches operate on blocks of data. Hence two threads on different cores writing to nearby indices in Z may both invalidate the cache line requiring many cache refills. Ideally parallelization is done in a way that threads on different cores work on output data that is located in different cache lines, e.g, thread 1: x = 1 to X/2 and thread 2: x = X/2 + 1 to X.

13.2 SoC Memory Hierarchies

Modern multi-core processors feature increasingly specialized architectures. Typically, each core has its own dedicated L1 and L2 caches, while the L3 cache is shared among all cores. Smaller chips often use different internal structures.

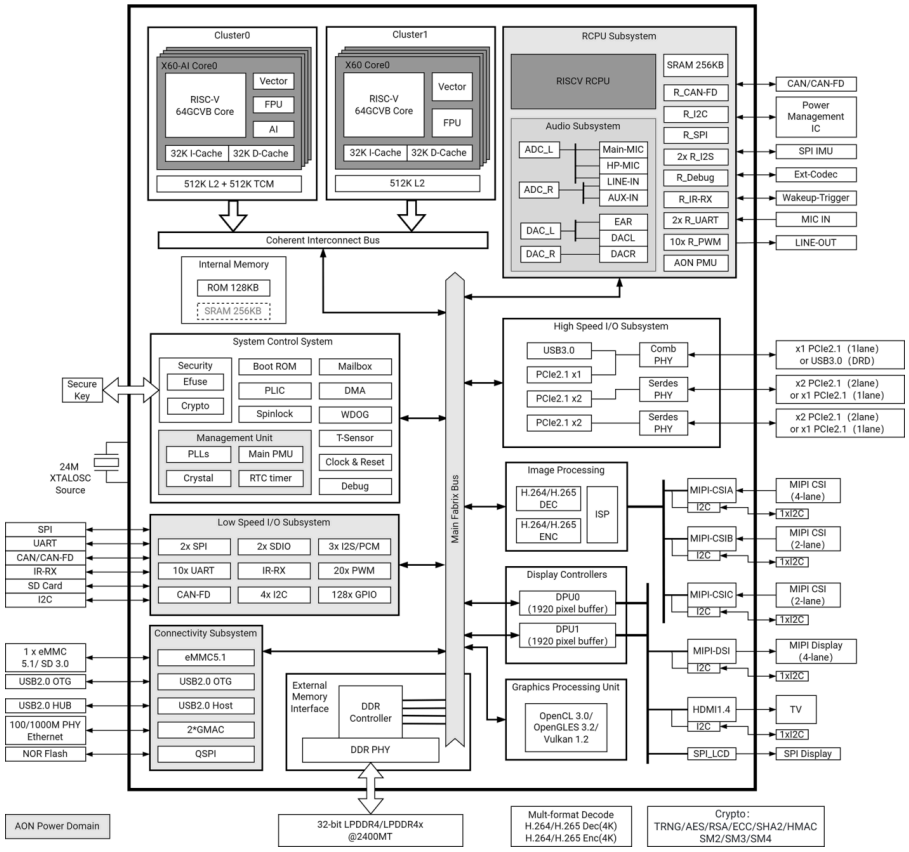


Figure 19: RISC-V K1 CPU diagram

a) Examine the cache hierarchy in the diagram (see Figure 19) for the SPacemit K1 RISC-V octa-core processor (see https://www.spacemit.com/en/key-stone-k1/) and explain the overall structure.

Solution: Each of the processor (the X60-AI cores) has its own dedicated 32K L1 instruction and 32K L1 data cache. Each compute cluster (consisting of 4 cores) then has its own 512K cache, not split into instr. and data. There is no L3 cache here.

from solution document:

- The 16 GB eMMC storage is accessible only through the connectivity subsystem.
- A shared L2 cache per cluster can cause contention → serializing eMMC memory accesses.
- Inter-cluster communication incurs additional latency due to the coherent interconnect bus.

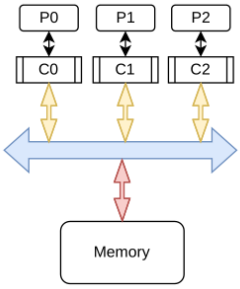
b) What might be an undesired effect of having a shared vector unit per cluster?

Solution: from solution document:

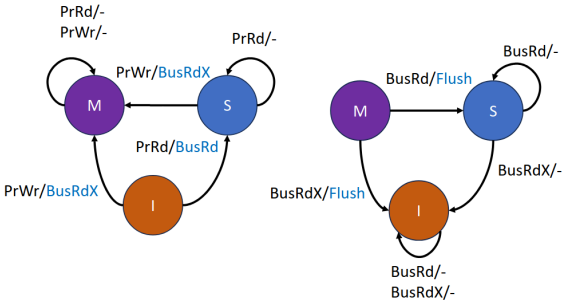
Since vector units typically transfer much larger amounts of data than scalar cores, they generate significantly higher bandwidth demands, which can monopolize the L2 cache and interconnect bus, leading to resource starvation for the simpler cores.

14 Multicore Cache Coherency

3 CPUs with **3 write-back** Caches **(C0/C1/C2)** have a shared bus system and a snooping-based Cache Coherency protocol, which is depicted in Subfigure 20.1. Assume for the task that each memory access uses the **same** memory address and each cache entry is invalid at the beginning.



Subfigure 20.1: Memory structure



Subfigure 20.2: MSI Protocol **without** BusUpgr

Figure 20: Memory system and MSI Protocol for task 1

14.1 MSI Protocol

a) Fill the following table with all corresponding states, bus requests and data transfer using the **MSI** protocol **w/o** BusUpgr(ade). Fill each line with the states after the operation is executed.

t	Operation	C0	C1	C2	Bus Request	data transfer
	Initial	-	-	-		
1	R2	-	-	S	BusRead	M → C2
2	W2	-	-	M	BusRdX	M → discard
3	R1	-	S	S	BusRead	C2 → M → C1
4	W0	M	I	I	BusRdX	M → C0
5	R2	S	I	S	BusRead	C0 → M →C2
6	R1	S	S	S	BusRead	M → C1
7	W0	M	I	I	BusRdX	M → discard

Table 6: Cache after (R/W)-operations for the MSI protocol

i Note

Without the BusUpgr(ade) protocol, each BusRead and BusRdX request initiates a data transfer, but the data from BusRdX requests gets discarded.

b) Would BusUpgr(ade) impact the data transfer?

Solution: Yes, it would save the two data transfers that get discarded, which saves data bandwidth.

14.2 MESI Protocol

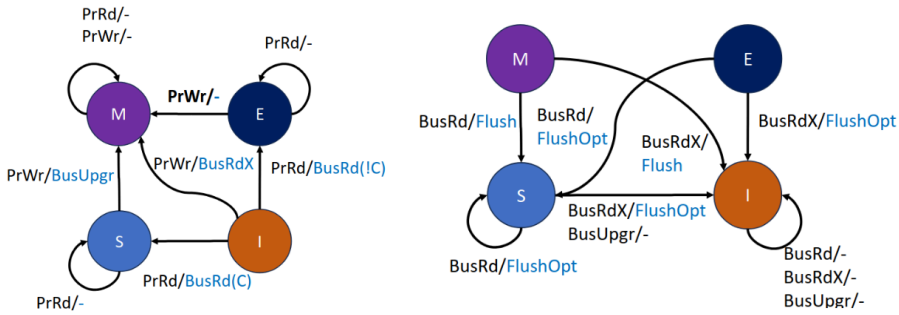


Figure 21: MESI Protocol

a) Now fill the table for the **MESI** protocol: Distinguish explixitly between **Flush** and **FlushOpt** and its impact on the data transfer. This also includes the direction from which it is flushed (e.g., Flush (C0)).

t	Operation	C0	C1	C2	Bus Request	data transfer
	Initial	-	-	-		
1	R2	-	-	E	BusRead !C	M → C2
2	W2	-	-	M	-	
3	R1	-	S	S	BusRead C	C2 → M → C1
4	W0	M	I	I	BusRdX	C1/C2 → C0
5	R2	S	I	S	BusRead C	C0 → M → C2
6	R1	S	S	S	BusRead C	C0/C2 → C1
7	W0	M	I	I	BusUpgrade	-

Table 7: Cache states after (R/W)-operations with the MESI protocol

14.3 MOESI Protocol

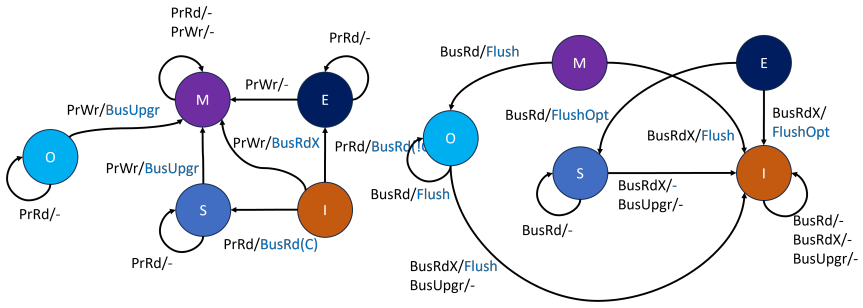


Figure 22: MOESI Protocol

a) Now fill the table for the **MOESI** protocol: Distinguish explixitly between **Flush** and **FlushOpt** and its impact on the data transfer. This also includes the direction from which it is flushed.

t	**Operation**	**C0**	**C1**	**C2**	**Bus Request**	**data transfer**
	Initial	-	-	-		
1	R2	-	-	E	BusRead !C	M → C2
2	W2	-	-	M	-	-
3	R1	-	S	O	BusRead C	C2 → C1 (with `Flush`)
4	W0	M	I	I	BusRdX	C2 → C0 (with `Flush`)
5	R2	O	I	S	BusRead C	C0 → C2 (with `Flush`)
6	R1	O	S	S	BusRead C	C0 → C1 (with `Flush`)
7	W0	M	I	I	BusUpgrade	-

Table 8: Cache states after (R/W)-operations with the MOESI protocol

15 Atomic Data Types and Memory Models

15.1 Synchronisation – Blocking und Nonblocking

Given the following program fragment that updates two variables `A` and `B` with new values. Access is protected from concurrent access using `Lock`/`Unlock`.

```
1 : Lock(S);
2 : A := A + 1;
3 : B := B - 1;
4 : Unlock(S);
```

Write alternative code with the same functionality, but which should be **non-blocking**, using:

a) **RMW (Read-Modify-Write)** operations.

Solution:

```pseudo
// initialize S with 0

while_start:
  if !read-modify-write(S, 0, 1) goto while_start; // acquire lock
  A := A + 1;
  B := B - 1;
  S := 0; // release lock
```

> **i Note**
>
> `read-modify-write(A, B, C)`:
> - `A`: variable to change
> - `B`: what we expect
> - `C`: what to set `A` to, if `A` matches `B`
> - returns true if set was successful

b) **LL/SC (Load-Linked / Store-Conditional)**

In addition to the statements already given above, you may also use `if`-statements, loops, labels, and `goto`s. The function `Address(V)` returns the address at which the variable `V` is stored.

What problems can occur in the **RMW** and **LL/SC** implementations?

Solution:

> **i Note**
>
> `LL(addr)`:
> - loads data from `addr`
> - remembers that this thread loaded the variable
> - returns the read value
>
> `SC(addr, data)`:
> - stores `data` to `addr`, if nobody wrote something since the corresponding `LL(addr)` call
> - returns if write was successful

```pseudo
    // initialize S with 0
    s_addr = Address(S)

  check_start:
    if LL(s_addr) != 0 then goto check_start;
    if not SC(s_addr, 1) then goto check_start;

    // lock acquired
    A := A + 1;
    B := B - 1;

    S = 0;
```

- Problems with RMW implementation:
  - Needs hardware support
  - is vulnerable to ABA problem
- Problems with LL/SC:
  - Needs hardware support

15.2 Release-Acquire Model

Given are the two threads Th_1 und Th_2 with their respective instructions to be executed. The initial variable assignment before the execution of the threads is

```
B := 24
X := 4
Y := 2
Z := 3
V1 := 0
V2 := 0
E := 0
```

Th_1:

```
  1a: V1 := atomic_load(B, Acquire);
  1b: X := X + 1;
  1c: V1 := V1 * X;
  1d: V1 := V1 - 1;
  1e: atomic_store(B, V1, Release);
  1f: atomic_store(E,1, Release);
  1g: ift1: if atomic_load(E, Acquire) = 1
    then goto ift1;
  1h: X := X * X;
  1i: Y := Z + 1;
  1j: V1 := V1 + 1;
  1k: Z := atomic_load(B, Acquire);
```

Th_2:

```
1 2a: ift2: if atomic_load(E, Acquire) != 1
2   2 then goto ift2
3 2b: V2 := atomic_load(B, Acquire);
4 2c: V2 := V2 / 7;
5 2d: X := X * Z
6 2e: Z := Z + 1;
7 2f: Y := Y + V2;
8 2g: atomic_store(B, V2, Release);
9 2h: atomic_store(E, 0, Release);
```

a) Calculate the **final** values of **each** variable after execution of both threads. Justify your answer.

Solution:

Th_1	Th_2	B	X	Y	Z	V1	V2	E
–	–	24	4	2	3	0	0	0
1a	2a	24	4	2	3	24	0	0
1b	2a	24	5	2	3	24	0	0
1c	2a	24	5	2	3	120	0	0
1d	2a	24	5	2	3	119	0	0
1e	2a	119	5	2	3	119	0	0
1f	2a	119	5	2	3	119	0	1
1g	2b	119	5	2	3	119	119	1
1g	2c	119	5	2	3	119	17	1
1g	2d	119	15	2	3	119	17	1
1g	2e	119	15	2	4	119	17	1
1g	2f	119	15	19	4	119	17	1
1g	2g	17	15	19	4	119	17	1
1g	2h	17	15	19	4	119	17	0
1h	–	17	225	19	4	119	17	0
1i	–	17	225	5	4	119	17	0
1j	–	17	225	5	4	120	17	0
1k	–	17	225	5	17	120	17	0

b) Specify all *release* and *acquire* operations in the example above that can be made *relaxed* without affecting the final values of the variables. Justify your answer.

Solution:

- `1a`: can be Relaxed as `1c` has RAW dependency
- `1e`: can be Relaxed as it has RAW dependency on `1d`, `1f` is Release such that it cannot move behind `1f`
- `1k`: can be Relaxed because `1g` is Acquire such that it cannot move before `1g`
- `2b`: can be Relaxed as `2a` is Acquire such that it cannot move before `2a`, `2c` has RAW dependency
- `2g`: can be Relaxed as it has RAW dependency on `2c`, `2h` is Release such that it cannot move behind `2h`

Correct synchronization is provided by variable `E`, here we need to keep Acquire and Release to fence the critical sections.

c) Is it necessary that the variable `B` is *atomic* qualified? Justify your answer.

Solution: No, `B` does not have to be atomic. Since Th_1 and Th_2 each access `B` exclusively through synchronization via `E`, the accesses to `B` are exclusive anyway.

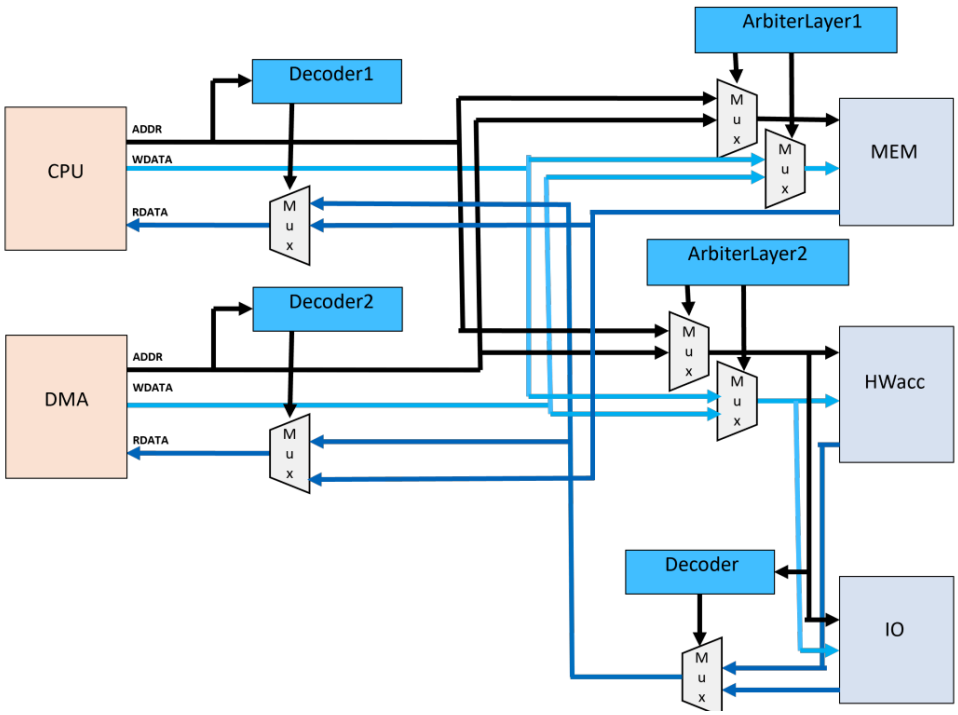
16 On-chip Buses

Given is the following architecture for a shared layered bus:

- There are two initiator components, CPU and DMA.
- There are three target components, MEM, HWacc and IO.
  - The MEM, is on layer 1
  - The Hwacc and IO component is on layer 2.

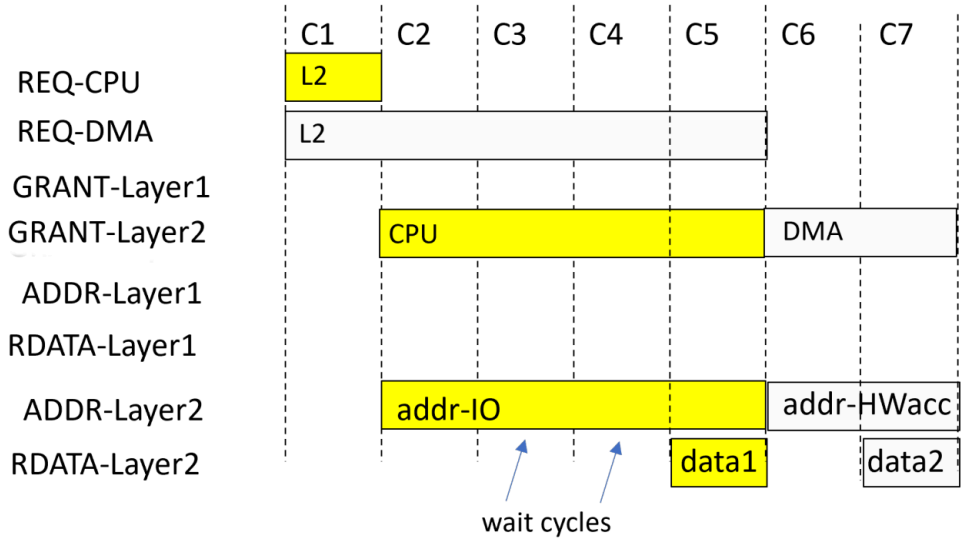
a) Draw the bus architecture. It is sufficient to show the directed connections between components and arbiters/decoders.

Solution:



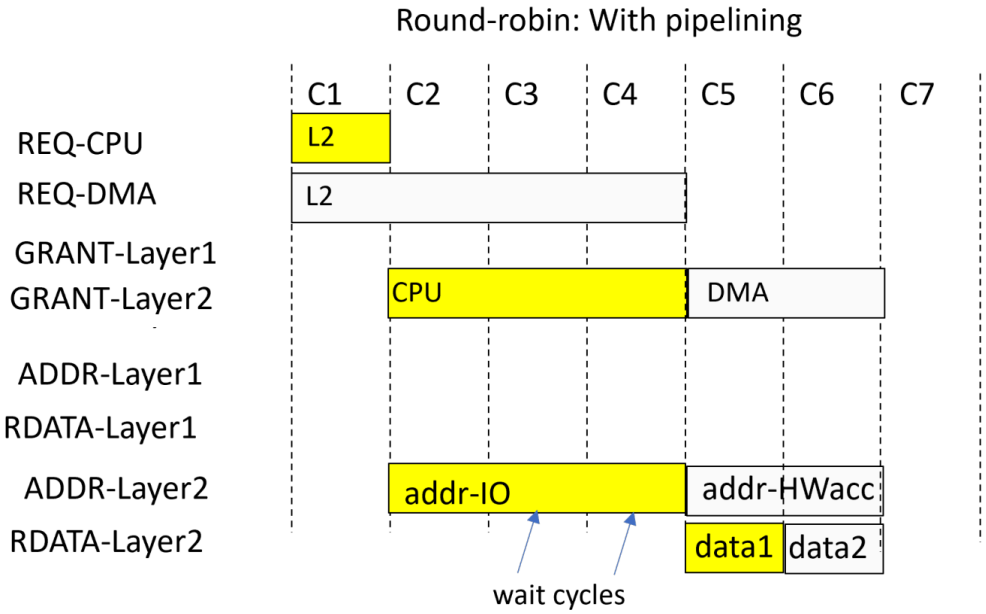
b) Assume that the CPU wants to read access the IO slave component in the bus cycle 1 and that the DMA wants to read access the HWacc in the same bus cycle 1. Draw the bus access diagram for the data and address bus of the two bus masters as well as the control request and grant signals for the two layers assuming that the bus does not support pipelining. The IO component inserts two wait cycles. The HWacc component inserts no wait cycles. The arbitration order is CPU first, then DMA. There is no pipelining.

Solution:



c) Now assume there is pipelining. Draw the bus access diagram again.

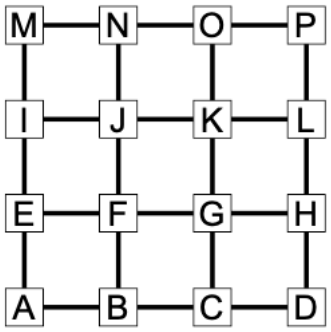
Solution:



17 Network-on-chip

17.1 NoC - Routing

Given is the following on-chip interconnection network (4-ary 2-cube):



a) Show a path from E to P using Dimension-order Routing for
   1) the variant XY

      i Note

      Dimension-order routing:
      - XY: first go in X, then go in Y direction
      - YX: first go in Y, then go in X direction

      Solution: E → F → G → H → L → P

   2) the variant YX

      Solution: E → I → M → N → O → P

b) Are there other minimal paths from E to P? If yes, provide an example, if no give a reason.

   Solution: Yes, all paths that go up and left in any combination have equal length and are minimal.

   e.g.: E → I → J → N → O → P

c) Give an example of a minimum path from `E` to `P` that is not used in the minimum version of Valiant's Algorithm in conjunction with XY dimension-order routing, even if all possible nodes are considered as intermediate nodes d'. Justify your solution.

   i Note

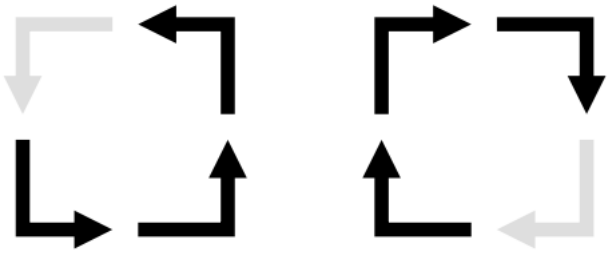
   Valiant's algorithm:
   - To diversify the routes, choose any random node that has to be visited before the goal.

   Minimum version:
   - chose this random node to be in the bounding box of the start and end point

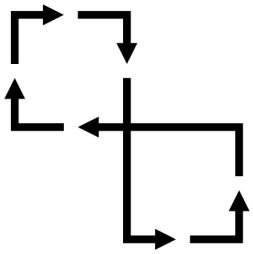
   Solution: E → F → J → K → O → P

   No matter which point of those to chose as d', it is not possible to get this path with XY routing.

d) To avoid deadlocks, one option is to prohibit turning in certain directions during routing, thereby preventing abstract cycles (see “turn model”). Is the selection of prohibited direction changes (gray) listed below promising in this regard? Justify your assessment.

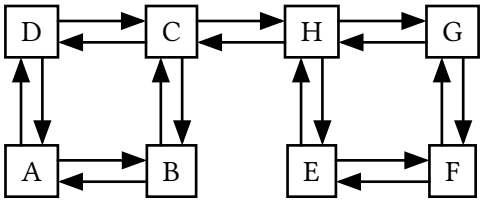


Solution: No, since it is possible to get a cycle once again with this rules by doing the following:



17.2 NoC - Channel Dependency Graph

Given is the following on-chip interconnection network:



a) Compared to previously presented topologies (such as meshes), not all neighboring routers are connected to each other here:

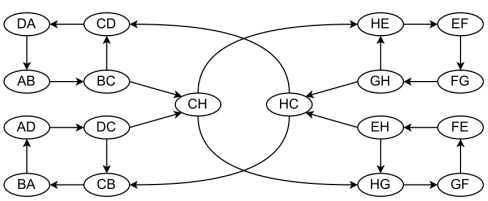
What arguments could be made in favor of choosing this topology?

Solution:

- less hardware needed
- This is almost equally effective as the mesh if the two clusters rarely communicate with each other, but there is much traffic in each of the clusters.

b) Create the CDG (Channel Dependence Graph) for the network (assume that 180° turns are not allowed).

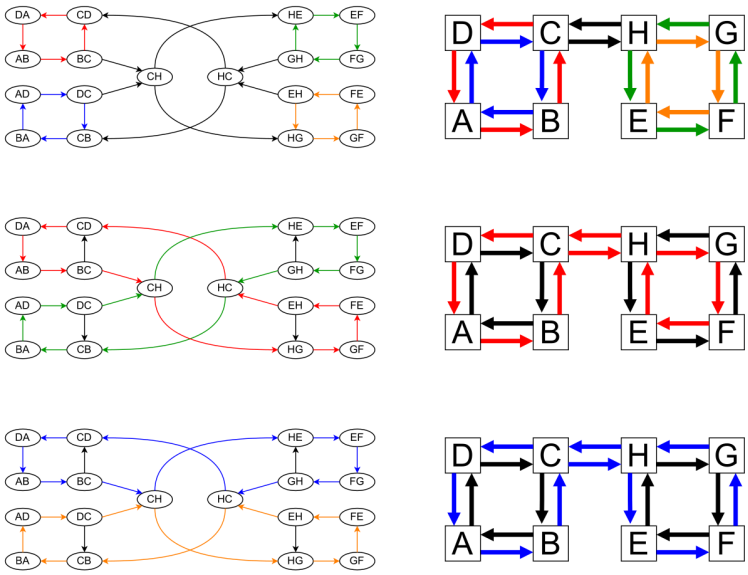
Solution:



c) Based on the CDG: Is there a possibility of deadlocks here, and how can this be determined using the CDG? Justify your assessment based on the CDG.

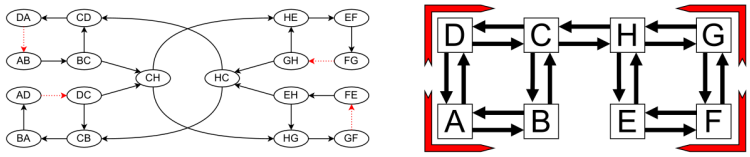
> **i Note**
>
> If there are cycles in the CDG, there is a possibility for deadlocks.

Solution: There are cycles in the CDG:



d) If deadlocks can occur: Propose a solution using the CDG and the results from task part (c). Show the resulting CDG.

Solution: Remove a few edges (prohibit turns) to remove all the cycles in the CDG:



18 ML on GPUs, Systolic Arrays and CGRAs

a) What is the challenge for executing ANNs with nonlinear operators, on Edge compute platforms?

Solution: Many nonlinear operators require to call math library functions with complex algorithms to compute results (e.g. tanh, softmax needs exp() computations). As this has to be done many times, it is computational expensive. Fast implementations may, e.g., use look up tables or specialized hardware blocks.

b) Why are MAC units a key building block of AI systems/ML platforms?

Solution: Fully-connected layers and convolution layers require to multiply activation with weights and accumulate the result (fully connected: row times column, convolution: all values in kernel window) , which basically results in many required MAC operations.

c) Assume an embedded ML platform that can execute 24 parallel MAC operations per cycle on 4-byte integer inputs (activations and weights). Assume that activations and weights are streamed in a straight forward fashion from memory. How many bytes have to be loaded per cycle (read memory bandwidth)? Is storing data a challenge (write memory bandwidth)?

Solution: $24\ \text{MAC} \rightarrow 24 \times 4$ byte for activation $+24 \times 4$ byte for weights $= 48 \times 4$ byte $= 192$ byte / cycle. Storing data usually not a challenge as we accumulate many values first locally and only store the result, not the intermediate sum.