ECE4700J Lab 4
Introduction of RISC-V

Haoyang Zhang

UM-SJTU Joint Institute

zhy-sjtu-jc@sjtu.edu.cn

June 9, 2022

Overview

Administrivia

- Lab 3 Assignment's live demo will be on 8:00pm - 10:30pm, Friday Jun. 10th (Beijing Time). Lab 3 Assignment code submission will be due on 23:59pm, Jun. 10th (Beijing Time).
- Homework 2 will be due on 02:59am, Jun. 13rd (Beijing Time).
- Lab 4 Assignment will be released soon (in this week). It will be due on Friday Jun. 24th (Beijing Time).
- Homework 3 will come soon.

Administrivia

- Start early on Lab 4!!!
- The Lab 4 Assignment will be significantly harder than former Lab assignments, and you will learn a lot.
- We will cover hints in lab lectures.

For Online Live Demo

- Please put yourself in the demo queue: `https://sjtu.feishu.cn/sheets/shtcn5u6ycQejXPQ2dQpLq24nhb`.
  - We will invite students into a discussion room to do the demo.
  - Others please wait in the main room.

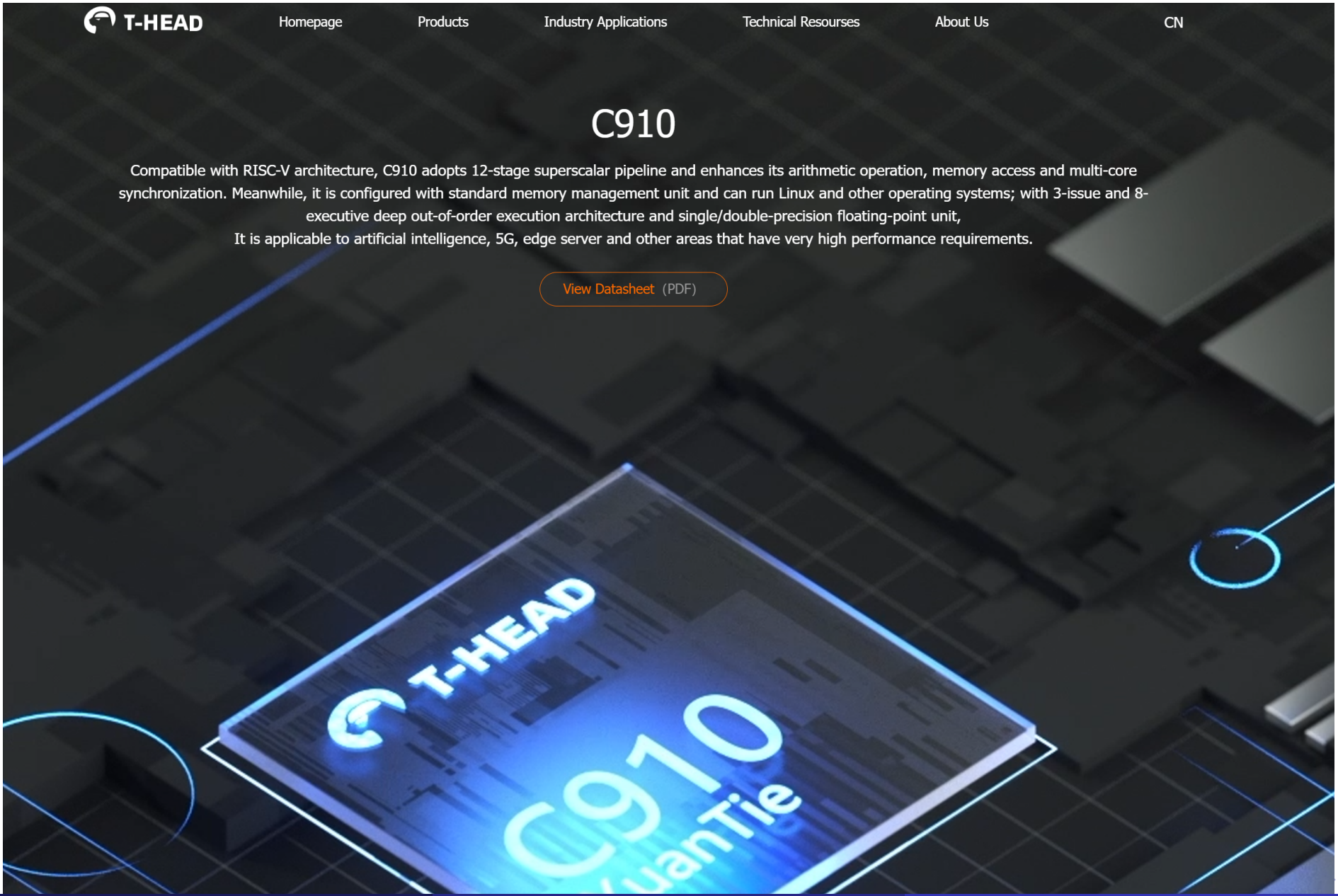
What is RISC-V

- RISC-V (pronounced “risk-five”) is an open, free ISA enabling a new era of processor innovation through open standard collaboration. Born in academia and research, RISC-V ISA delivers a new level of free, extensible software and hardware freedom on architecture, paving the way for the next 50 years of computing design and innovation. - RISC-V Foundation
  - Take the last part of with a huuuuuuge grain of salt
- It originated in UC Berkeley, but now it has own its foundation with a large number of contributors
- Spawned quite a few startups

Why RISC-V

- A completely open ISA that is freely available to academia and industry
- A real ISA suitable for direct native hardware implementation, not just simulation or binary translation
- An ISA that avoids "over-architecting" for a particular microarchitecture style (e.g., microcoded, in-order, decoupled, out-of-order) or implementation technology (e.g., full-custom, ASIC, FPGA), but which allows efficient implementation in any of these
- An ISA separated into a small base integer ISA, usable by itself as a base for customized accelerators or for educational purposes, and optional standard extensions, to support general-purpose software development
    - Most important part for us in particular, is good software support

T-Head's XuanTie 910



Why RISC-V

- Lets us explore more layers of the computing stack, mainly compilers and systems
- Can arbitrarily generate test cases, since we can just write in C now!
  - Easier for you to test
  - Easier for the staff to shuffle around test cases
  - Easier to generate large test cases that can actually benefit from additional features and properly reward those who worked on extra features

ISA Overview - Base ISA

- Base ISA + many extensions including privileges mode
- 32, 64 and 128-bit address space
  - We only use 32-bit for now, the other two only add a few instructions
- 32 integer registers
- Byte level addressing for memory, little endian
- Instructions must align to 32-bit addresses (unless they are compressed)
- No condition codes or carry out bits to detect overflow
  - Intentional, these can be achieved in software
- Comparisons are built in for branches
  - e.g. beq x1, x2, offset

ISA Overview - Instruction Formats

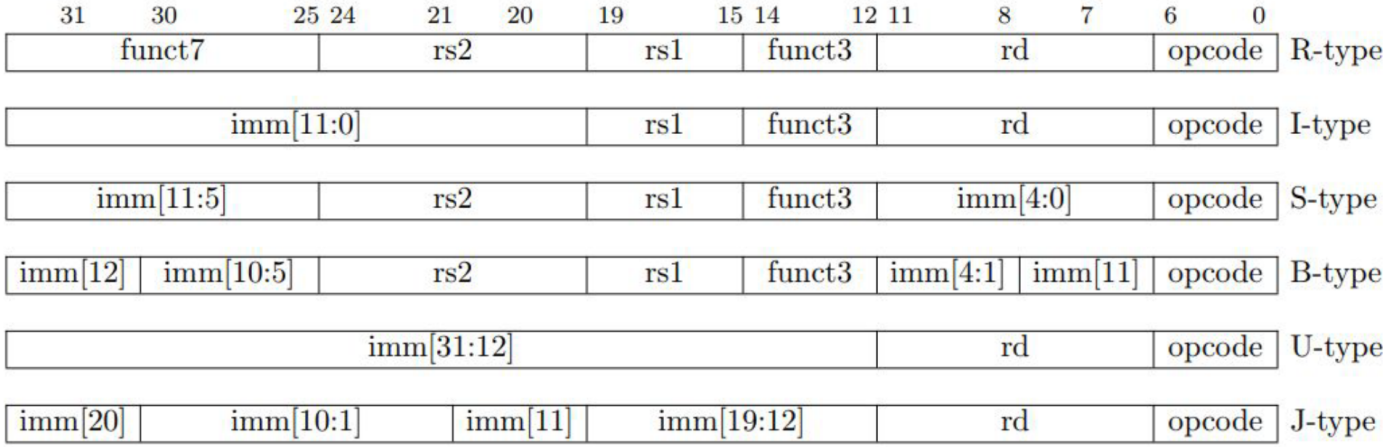


Figure 2.3: RISC-V base instruction formats showing immediate variants.

ISA Overview - More Extensions

- V - Has a vector extension as well, if staff in the future wants to spice things up
- A - The atomic extension will be partially used to implement locks in the future
- F, D, Q, L- Floating point extensions can be supported for people's own interest
- C - Compressed extension to increase code density
- E - for embedded systems; reduced number of registers (only 16), can be combined with C to save ROM
- T - RISC-V has plans to support transactional memory in the future (omegalul)
- Z series, basically all future extensions since they ran out of letters

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends

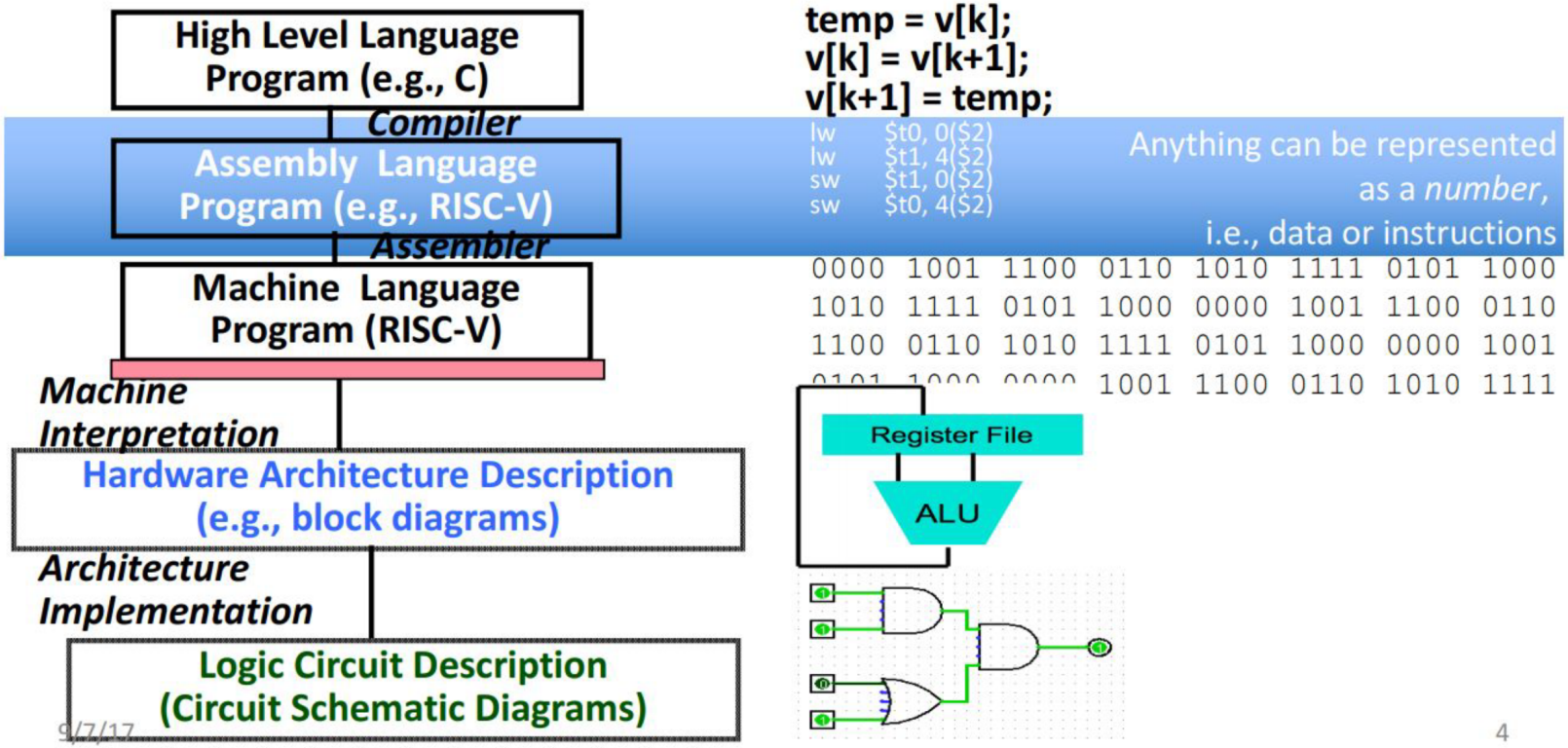
RV32I Base Integer Instructions

lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

RV32M Multiply Extension

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
`mul`	MUL	R	`0110011`	`0x0`	`0x01`	`rd = (rs1 * rs2)[31:0]`
`mulh`	MUL High	R	`0110011`	`0x1`	`0x01`	`rd = (rs1 * rs2)[63:32]`
`mulsu`	MUL High (S) (U)	R	`0110011`	`0x2`	`0x01`	`rd = (rs1 * rs2)[63:32]`
`mulu`	MUL High (U)	R	`0110011`	`0x3`	`0x01`	`rd = (rs1 * rs2)[63:32]`
`div`	DIV	R	`0110011`	`0x4`	`0x01`	`rd = rs1 / rs2`
`divu`	DIV (U)	R	`0110011`	`0x5`	`0x01`	`rd = rs1 / rs2`
`rem`	Remainder	R	`0110011`	`0x6`	`0x01`	`rd = rs1 % rs2`
`remu`	Remainder (U)	R	`0110011`	`0x7`	`0x01`	`rd = rs1 % rs2`

Software Environment



Software Environment

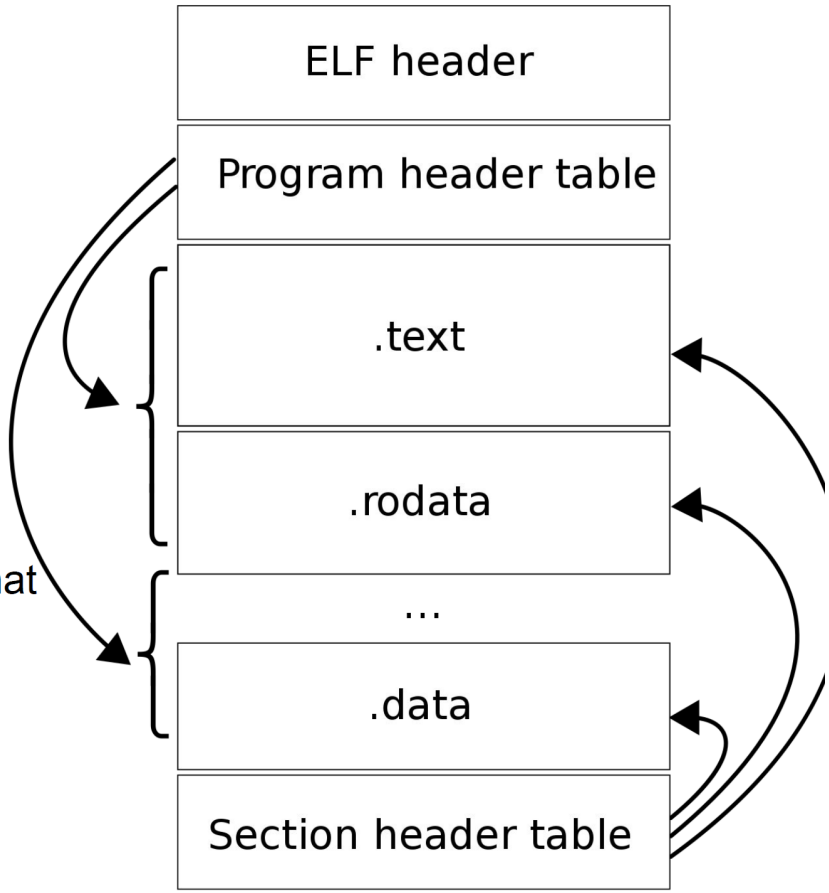
- Why do I remotely even care about software in a hardware class
  - Believe me, it's important
  - Architecture is the bridge between the two (insert preaching)
- For RISC-V, we will have both C programs and assembly programs to test
- At the same time, you also need to have a grasp of how C works at a very low level
  - It doesn't affect your implementation for sure, but knowing this will make your life easier
  - You will also learn A LOT

Software Environment - GNU Tools

- There's a full suite of GNU tools for RISC-V
  - `gcc` - compiler
  - `as` - assembler
  - `ld` - linker
  - `objdump` - dissassembler
  - `objcopy` - don't really need but cool
  - `g++` - don't use this
  - `gdb` - no idea if this actually works
  - a lot more that you can explore yourself...

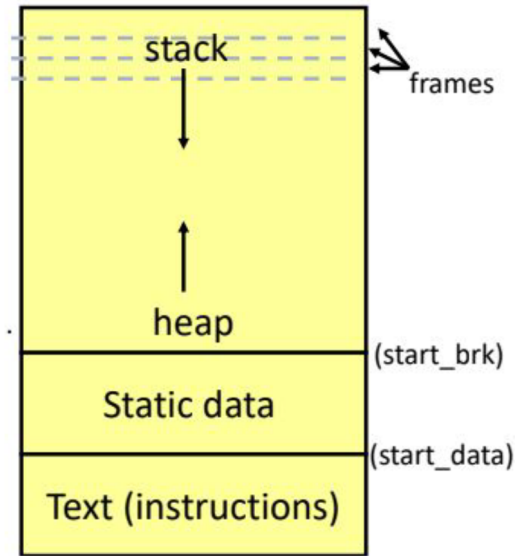
Software Environment - ELF

- What happens when you compile a program?
  - You generate an ELF, not Legolas though
  - But rather Executable and Linkable Format
- What is actually inside an ELF?
  - ELF/program header
    - Usually tells what OS it's for
    - Where in memory to put the program in
  - .text: the actual instructions of the program
  - .rodata: read-only data, but we don't enforce that
  - .data: modifiable program data
  - Section header table: where's what



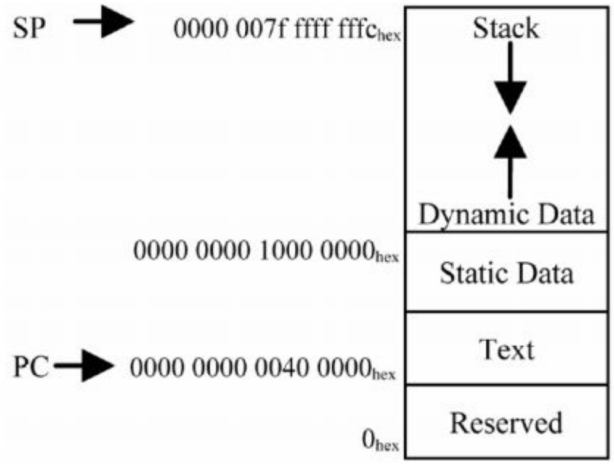
Software Environment - Program Space

- Flashback to 370 or whatever computer organization class you had
- What does the memory space for a program look like?
- Stack for statically allocated variables, pointer decrements
- Heap for dynamic memory, pointer increments



Software Environment - Program Space

- Example of program space allocation for arm processors->
  - The linker allocate the memory space
- In general memory addresses around 0x0 are precious
  - Some peripherals on the serial buses can only talk to limited addresses
- In our case, the text section starts at 0x0 to simplify loading
- The stack pointer starts at 0x10000
  - The end of the testbench memory space
  - This means any program that you write, text+data+stack < 64KiB



Software Environment - Program Space

```c
extern void exit();
#include <stdlib.h>

int** avg_pooling(int image [][4]) {
        int down_sample [2][2];
        int avg = 0;
        for (int i = 0; i < 2; ++i) {
                down_sample[0][i] = (image[0][i*2] + image[1][i*2] + image[0][i*2+1] + image[1][i*2+1]) >> 2;
                down_sample[1][i] = (image[3][i*2] + image[4][i*2] + image[3][i*2+1] + image[4][i*2+1]) >> 2;
        }
        return down_sample;
}



int main() {
        int image [4][4];
        for (int i = 0; i < 4; ++i) {
                for (int j = 0; j < 4; ++j) {
                        image[i][j] = rand();
                }
        }
        int** worse_image;
        worse_image = avg_pooling(image);

        return 0;
}
```

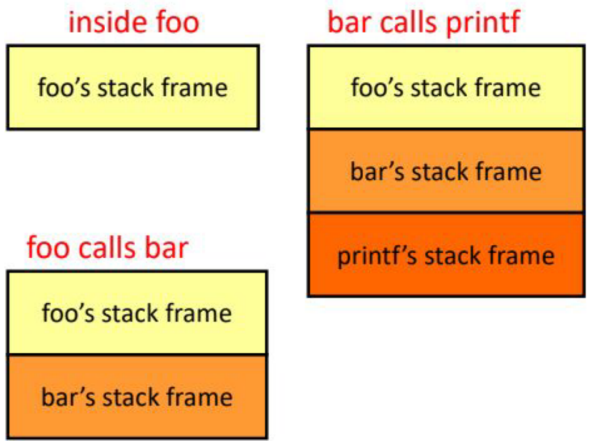
```
program.debug.elf:     file format elf32-littleriscv
Disassembly of section .text:
000000c8 <avg_pooling>:
        for (int i = 0; i < 2; ++i) {
                down_sample[0][i] = (image[0][i*2] + image[1][i*2] + image[0][i*2+1] + image[1][i*2+1]) >> 2;
                down_sample[1][i] = (image[3][i*2] + image[4][i*2] + image[3][i*2+1] + image[4][i*2+1]) >> 2;
        }
        return down_sample;
}
  c8:   00000513                li      a0,0
  cc:   00008067                ret

000000d0 <main>:
int main() {
  d0:   ff010113                addi    sp,sp,-16
  d4:   00812423                sw      s0,8(sp)
  d8:   00112623                sw      ra,12(sp)
  dc:   00400413                li      s0,4
        int image [4][4];
        for (int i = 0; i < 4; ++i) {
                for (int j = 0; j < 4; ++j) {
                        image[i][j] = rand();
  e0:   00000097                auipc   ra,0x0
  e4:   054080e7                jalr    84(ra) # 134 <rand>
  e8:   00000097                auipc   ra,0x0
  ec:   04c080e7                jalr    76(ra) # 134 <rand>
  f0:   00000097                auipc   ra,0x0
  f4:   044080e7                jalr    68(ra) # 134 <rand>
  f8:   fff40413                addi    s0,s0,-1
  fc:   00000097                auipc   ra,0x0
 100:   038080e7                jalr    56(ra) # 134 <rand>
        for (int i = 0; i < 4; ++i) {
 104:   fc041ee3                bnez    s0,e0 <main+0x10>
        }
        int** worse_image;
        worse_image = avg_pooling(image);

        return 0;
}
 108:   00c12083                lw      ra,12(sp)
 10c:   00812403                lw      s0,8(sp)
 110:   00000513                li      a0,0
 114:   01010113                addi    sp,sp,16
 118:   00008067                ret

0000011c <srand>:
 11c:   00000797                auipc   a5,0x0
 120:   52478793                addi    a5,a5,1316 # 640 <_impure_ptr>
 124:   0007a783                lw      a5,0(a5)
 128:   0aa7a423                sw      a0,168(a5)
 12c:   0a07a623                sw      zero,172(a5)
 130:   00008067                ret
```

```
Disassembly of section .data:

00000200 <impure_data>:
 200:   0000
 202:   0000
 204:   04ec
 206:   0000
 208:   0554
 20a:   0000
 20c:   05bc
        ...
 2a6:   0000
 2a8:   0001
 2aa:   0000
 2ac:   0000
 2ae:   0000
 2b0:   330e
 2b2:   abcd
 2b4:   1234
 2b6:   e66d
 2b8:   deec
 2ba:   0005
 2bc:   0000000b
        ...

Disassembly of section .srodata:

000001e0 <_global_impure_ptr>:
 1e0:   0200
        ...

Disassembly of section .sdata:

00000640 <_impure_ptr>:
 640:   0200
```

Software Environment - Function Calls

- Every time there's a function call, have a frame pointer that saves the previous stack pointer
- Caller/callee save the variables

```
void foo()
{
   int x, y[2];
   bar(x);
}

void bar(int x)
{
   int a[3];
   printf();
}
```



Software Environment - ABI

- Registers aren't just registers, each of them has a meaning
- Such concept is called 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	t0	Temporary/alternate link register	Caller
x6–7	t1–2	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
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Code Example

```
add     1   2   3     ; reg 3 = reg 1 + reg 2
nand    3   4   5     ; reg 5 = reg 3 ~& reg 4
add     6   3   7     ; reg 7 = reg 6 + reg 3
lw      3   6   10    ; reg 6 =  Mem[reg 3 + 10]
sw      6   2   12    ; Mem[reg6+12] = reg 2
```

Code Example

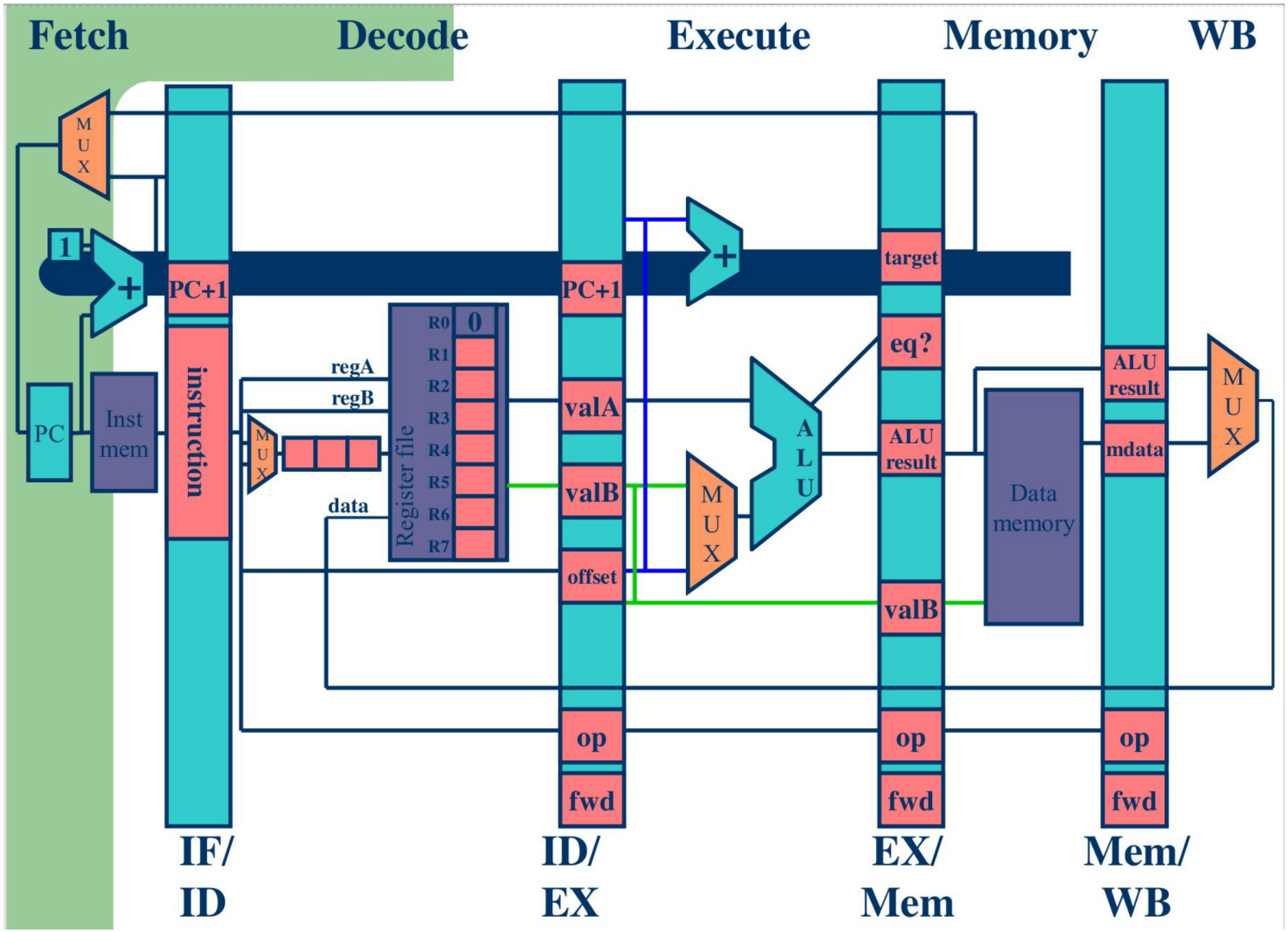
test_progs > rv32_evens.s

```
/*
    TEST PROGRAM #2: compute even numbers that are less than 16


    long output[16];

    void
    main(void)
    {
      long i,j;

      for (i=0,j=0; i < 16; i++)
        {
          if ((i & 1) == 0)
            output[j++] = i;
        }
    }
*/
    data = 0x1000
    li  x3, 0
    li  x4, data
loop1: andi x31, x3, 1
    bne x31,    x0, loop2 #
    sw  x3, 0(x4)
    addi    x4, x4, 0x8 #
loop2:  addi    x3, x3, 0x1 #
    slti    x2, x3, 16 #
    bne x2, x0, loop1 #
    wfi


```

Lab4 Assignment - VeriSimple Pipeline

- Same basic pipeline as in class
  - More info at Appendix A of the textbook
    - Starting with the 8th Edition of Hennessy and Patterson, all examples should be using RISC-V as well
  - Supports RV32IM minus divide, remainder and system instructions
- Need to add hazard logic to a simple 5 stage pipeline
- Given pipeline without hazard detection or forwarding logic
- Programs still run because only one instruction is allowed in the pipeline at a time

Lab4 Assignment - VeriSimple Pipeline



Lab 4 Assignment Overview

- Forwarding
  - Like what we covered in class
  - Need to forward results from later stages to EX
- Structural Hazards
  - Only one memory port for fetching and memory accesses
  - Memory gets priority over fetch
    - You need this to guarantee forward progress
- Control Hazards
  - Predict not taken, resolved in MEM stage
  - Flush IF/ID, ID/EX, EX/MEM if incorrect

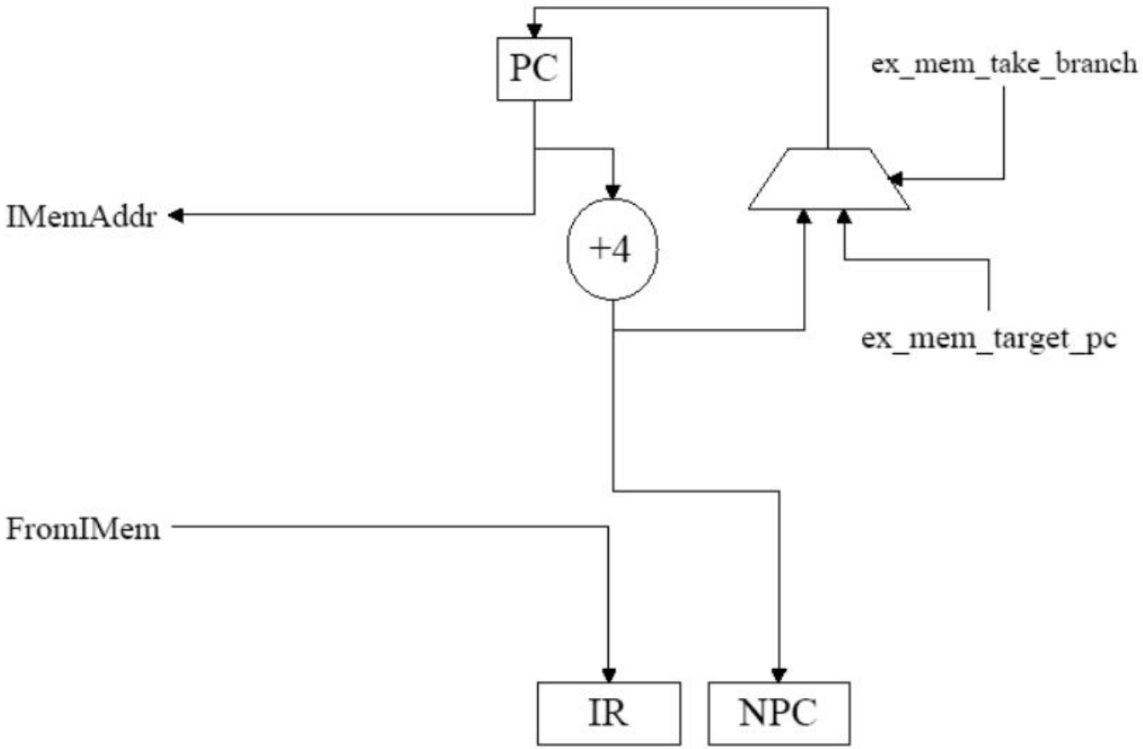
Testing Environment Setup

- This will be very complicated
- After we finish designing the assignment, a Lab4-part2 slide will be released to talk about this part.

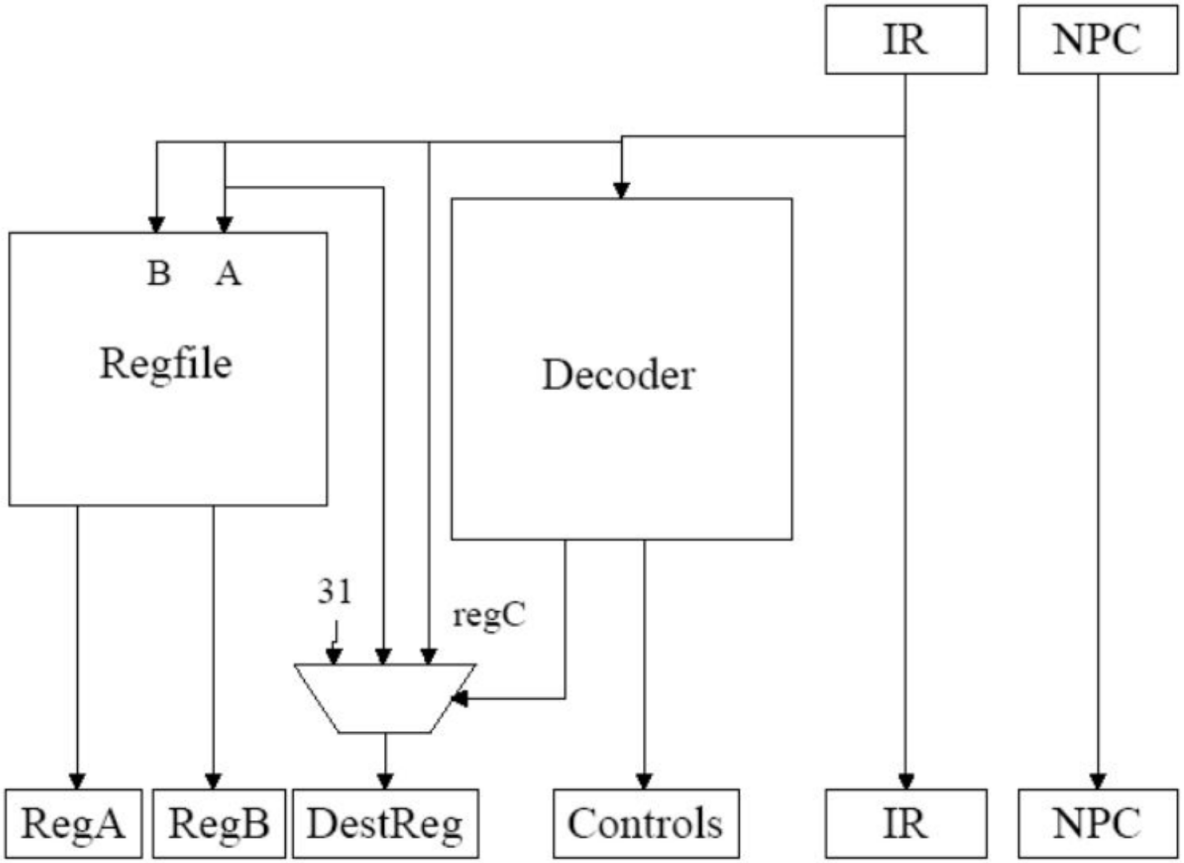
Testcases Writing

- Feel free to write your own test programs, C or assembly
- Most of the glibc library functions should work, as long as they don't use any system calls
- We do have our own version of `malloc()`, `calloc()`, `free()`
  - `tj_malloc()`, `tj_calloc()`, `tj_free()`
- Avoid using the division ("/") and modulus operator ("%"), since they will generate unsupported instructions like DIV and REM

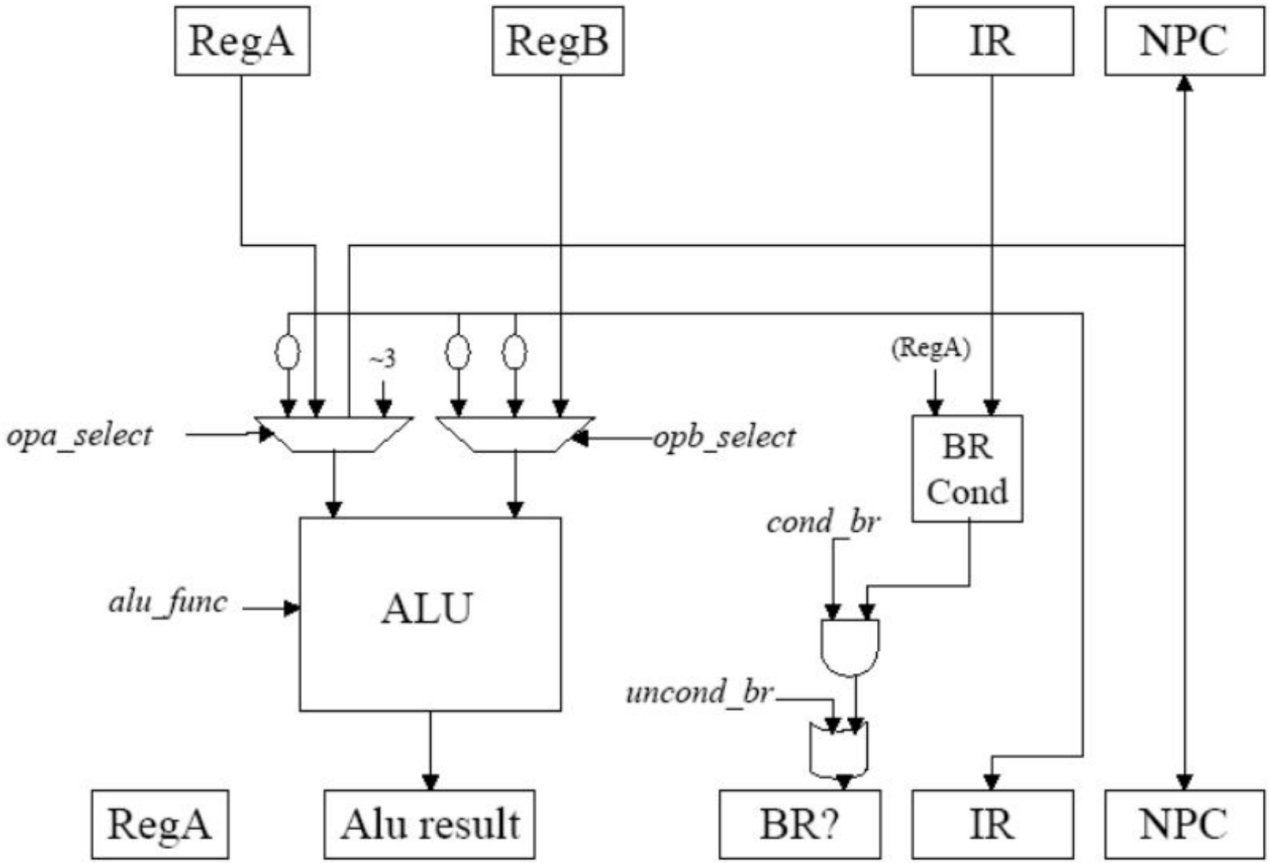
Pipeline Specifics - Fetch



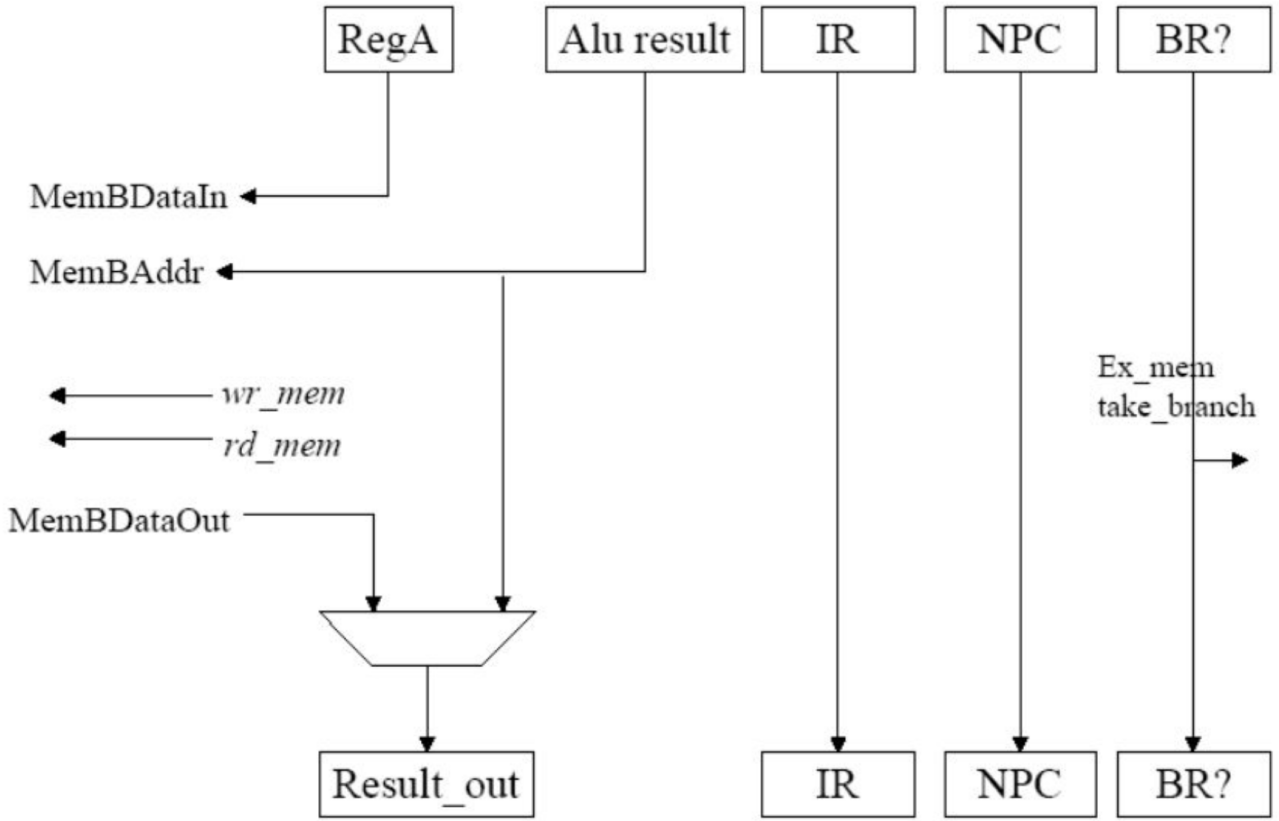
Pipeline Specifics - Decode



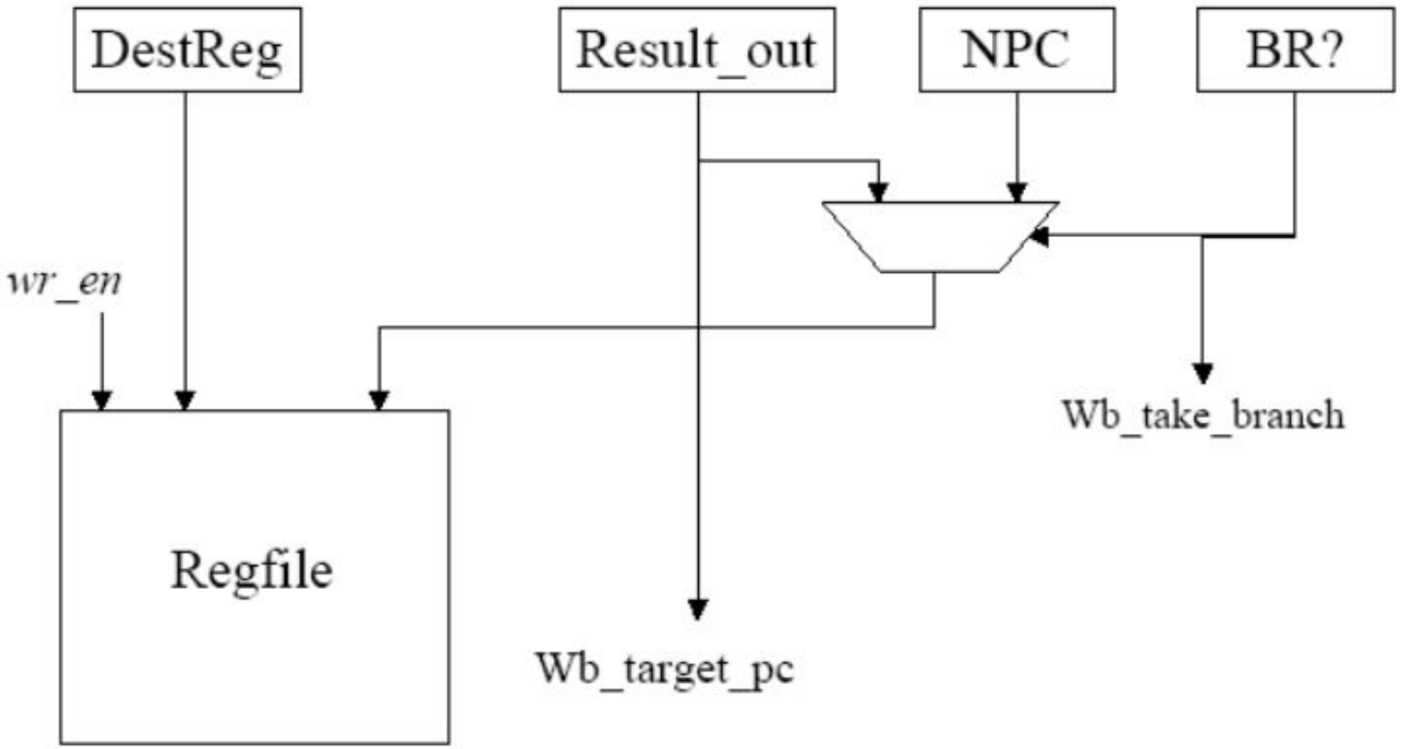
Pipeline Specifics - EX



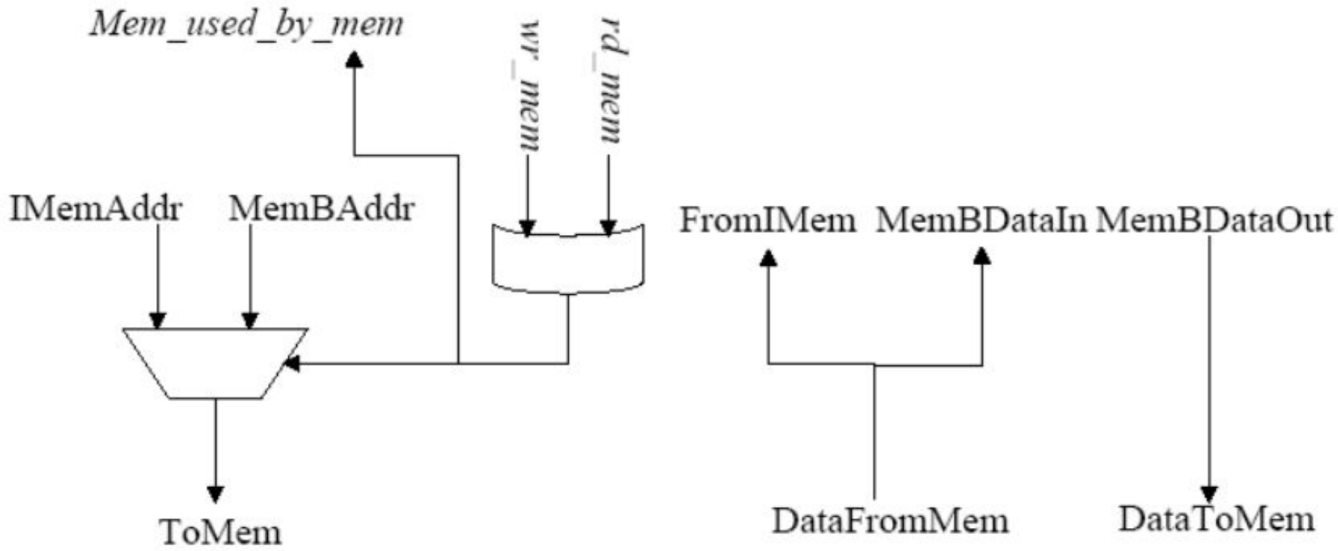
Pipeline Specifics - MEM



Pipeline Specifics - WB



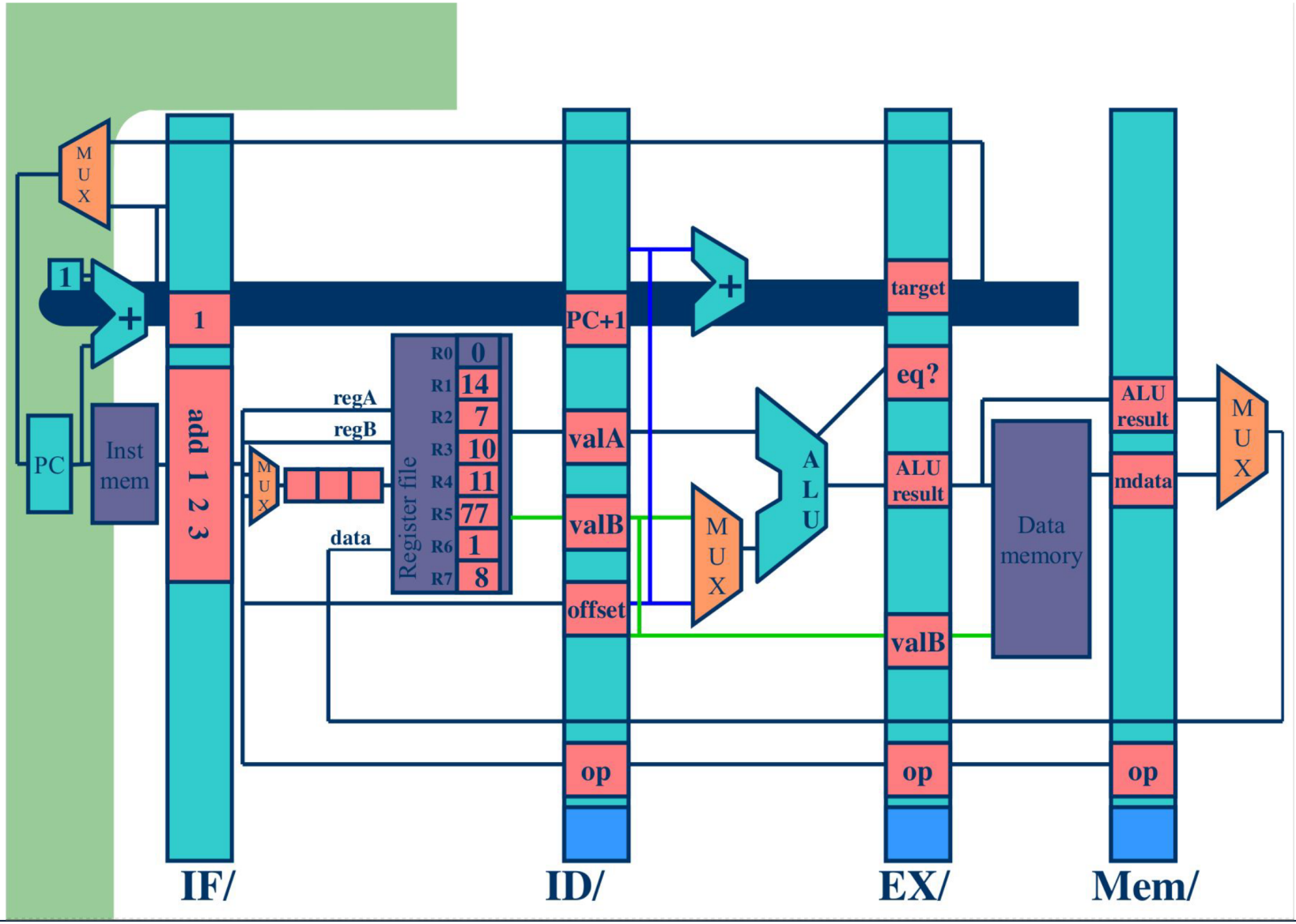
Pipeline Specifics - Memory Arbitation



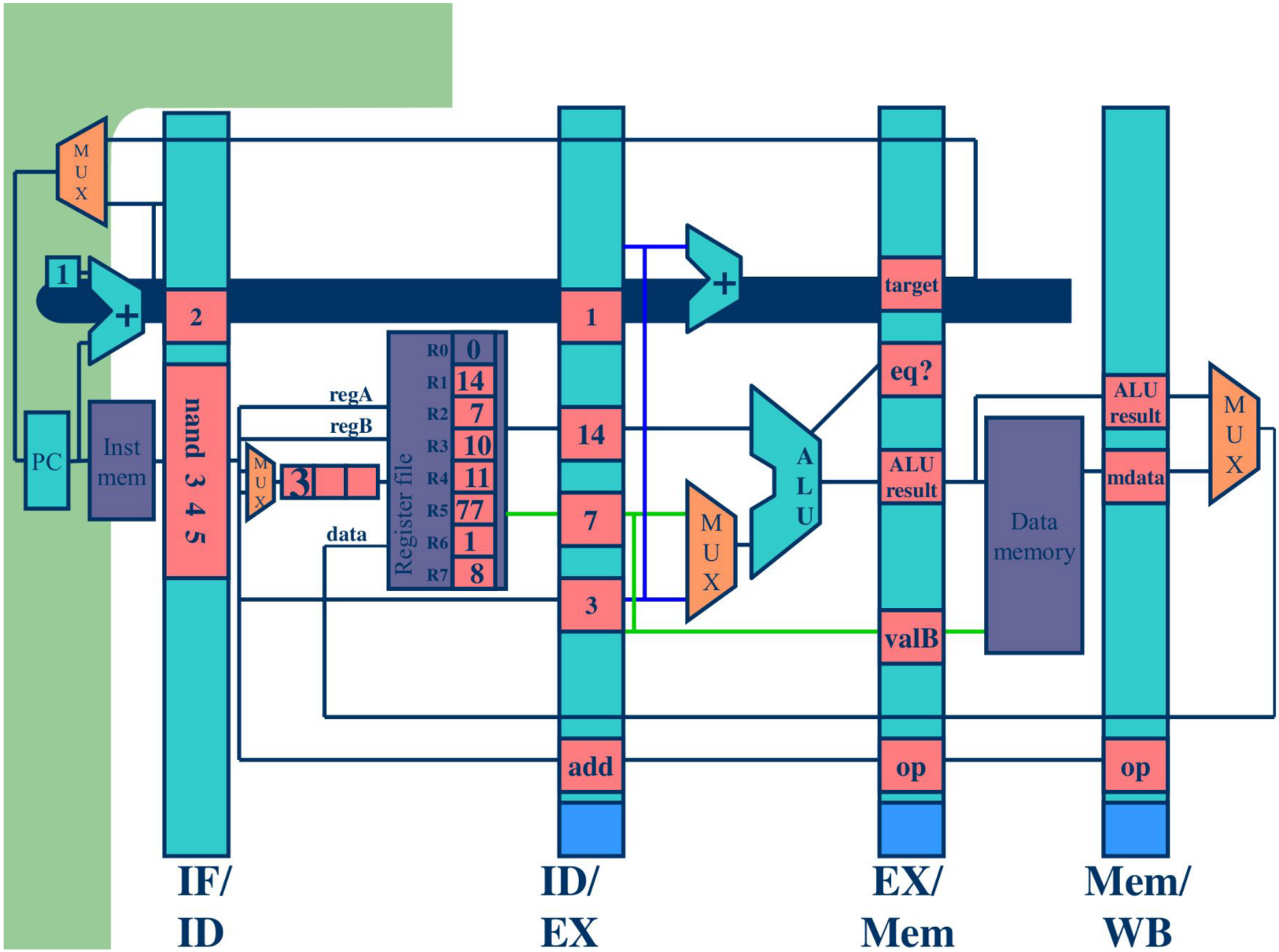
Sample Code

```
add     1   2   3     ; reg 3 = reg 1 + reg 2
nand    3   4   5     ; reg 5 = reg 3 ~& reg 4
add     6   3   7     ; reg 7 = reg 6 + reg 3
lw      3   6   10    ; reg 6 =  Mem[reg 3 + 10]
sw      6   2   12    ; Mem[reg6+12] = reg 2
```

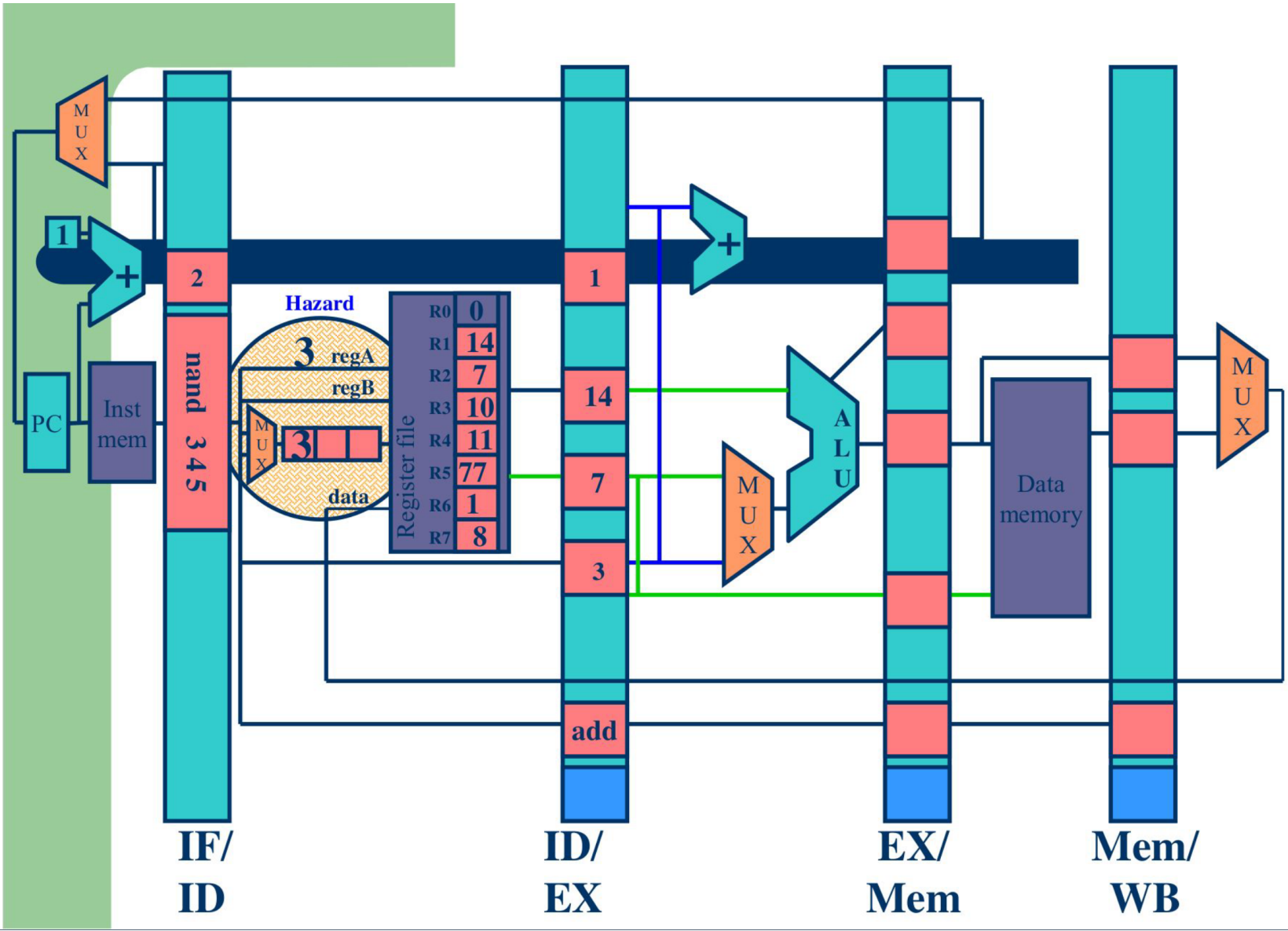
Lab4 Hints


M
U
X
1
+
1
add 1 2 3
PC
Inst
mem
regA
regB
data
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
Register file
PC+1
valA
valB
offset
op
+
target
eq?
ALU
result
valB
op
A
L
U
Data
memory
ALU
result
mdata
op
IF/
ID/
EX/
Mem/

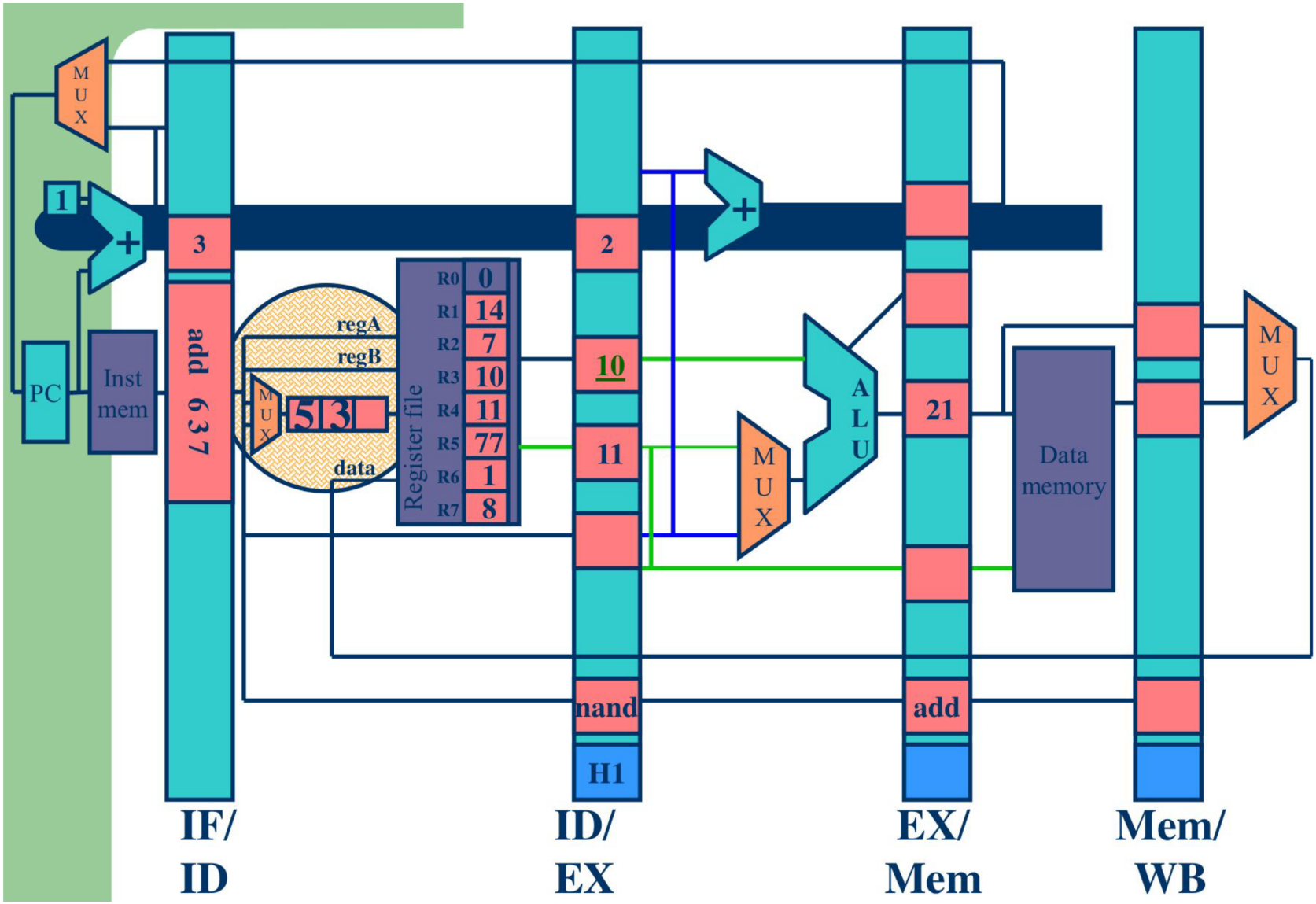
Lab4 Hints


M
U
X
1
+
2
PC
Inst
mem
nand 3 4 5
regA
regB
M
U
X
3
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
1
14
7
3
add
+
target
eq?
ALU
result
valB
op
M
U
X
A
L
U
Data
memory
ALU
result
mdata
op
M
U
X
IF/
ID
ID/
EX
EX/
Mem
Mem/
WB

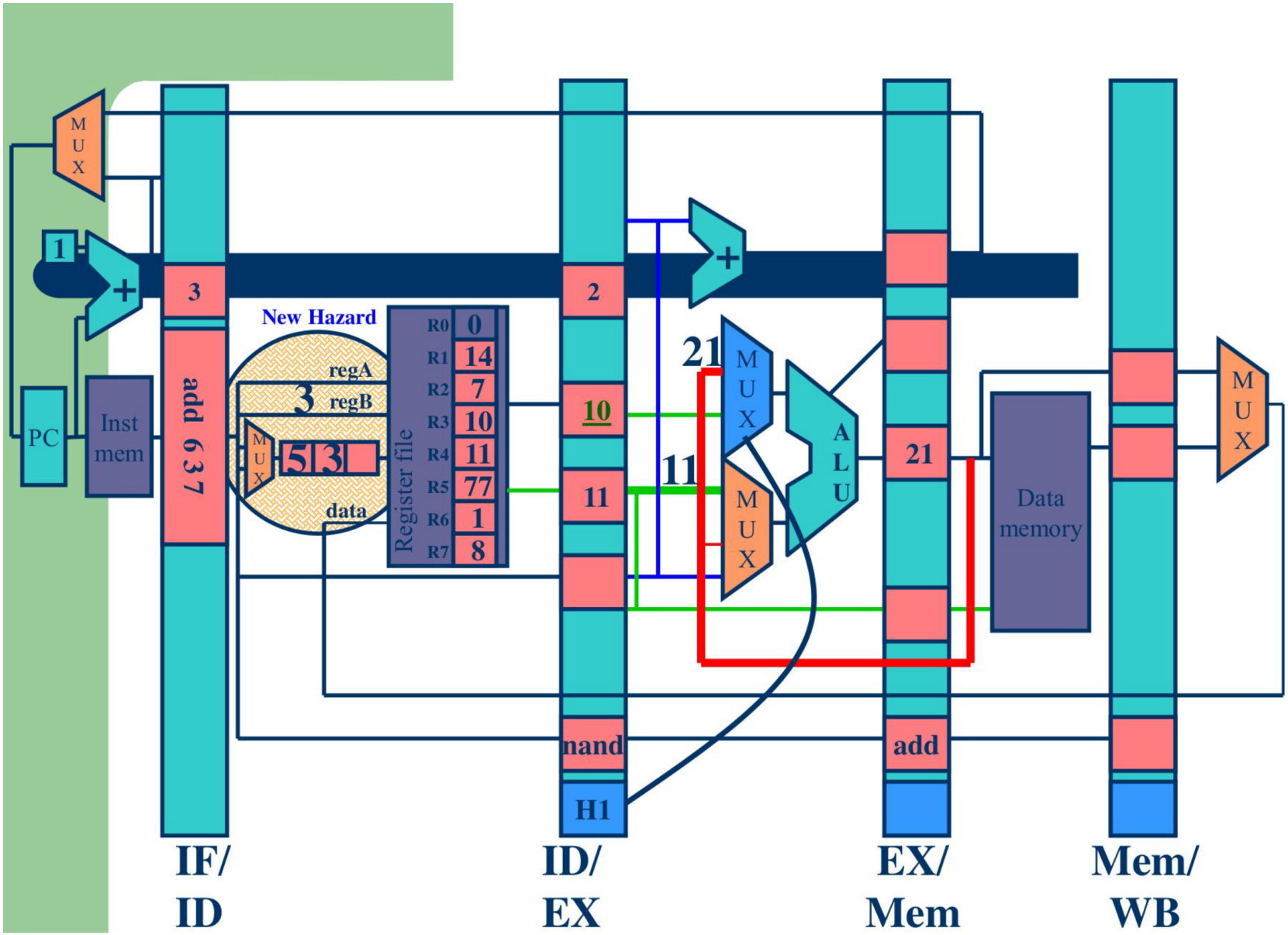
Lab4 Hints


MUX
1
+
2
PC
Inst mem
nand 3 4 5
Hazard
3
regA
regB
MUX
3
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
1
14
7
3
add
+
MUX
ALU
Data memory
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

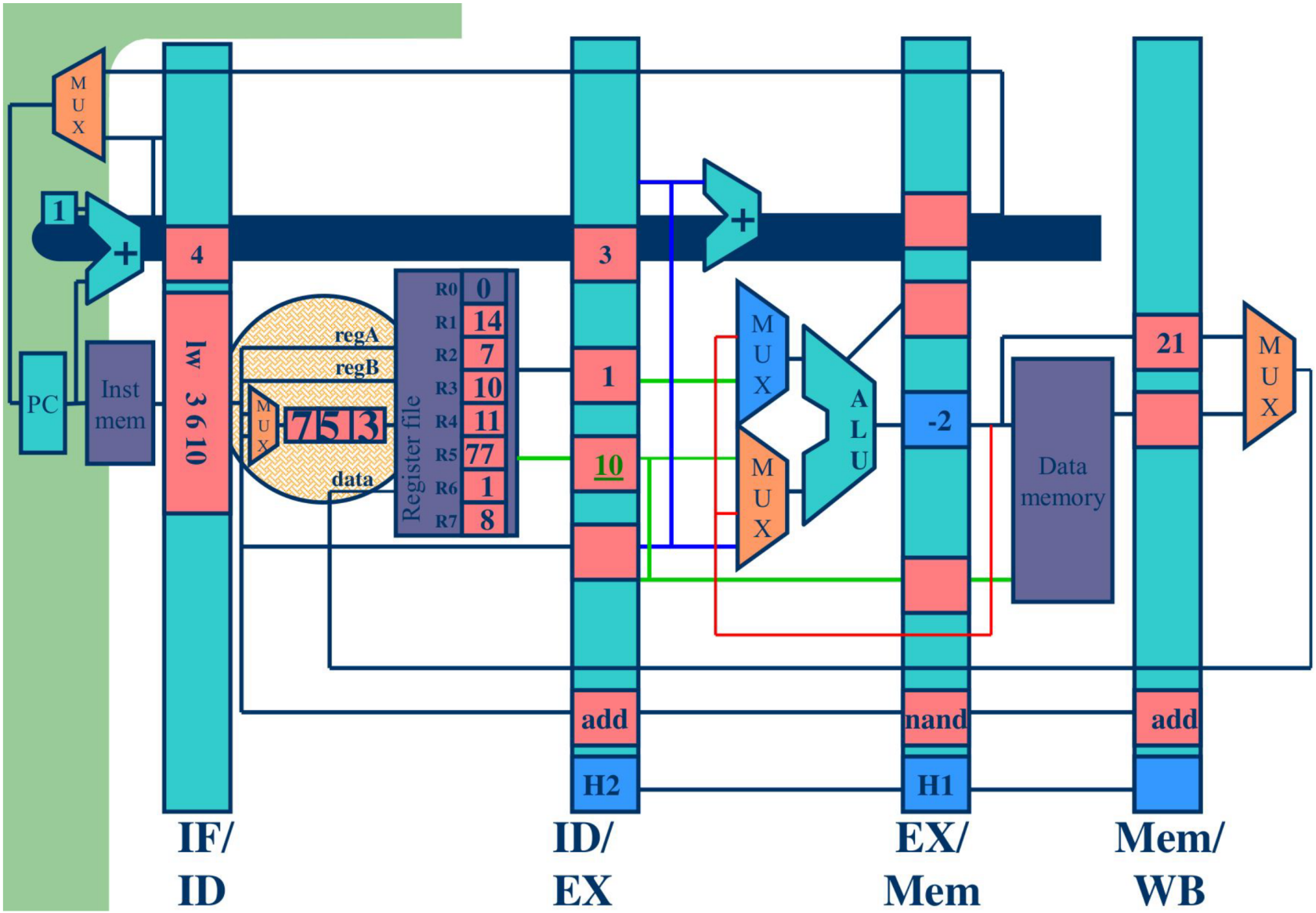
Lab4 Hints


M
U
X
1
+
3
add 6 3 7
PC
Inst
mem
regA
regB
M
U
X
5 3
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
2
10
11
nand
H1
+
M
U
X
A
L
U
21
add
Data
memory
M
U
X
IF/
ID
ID/
EX
EX/
Mem
Mem/
WB

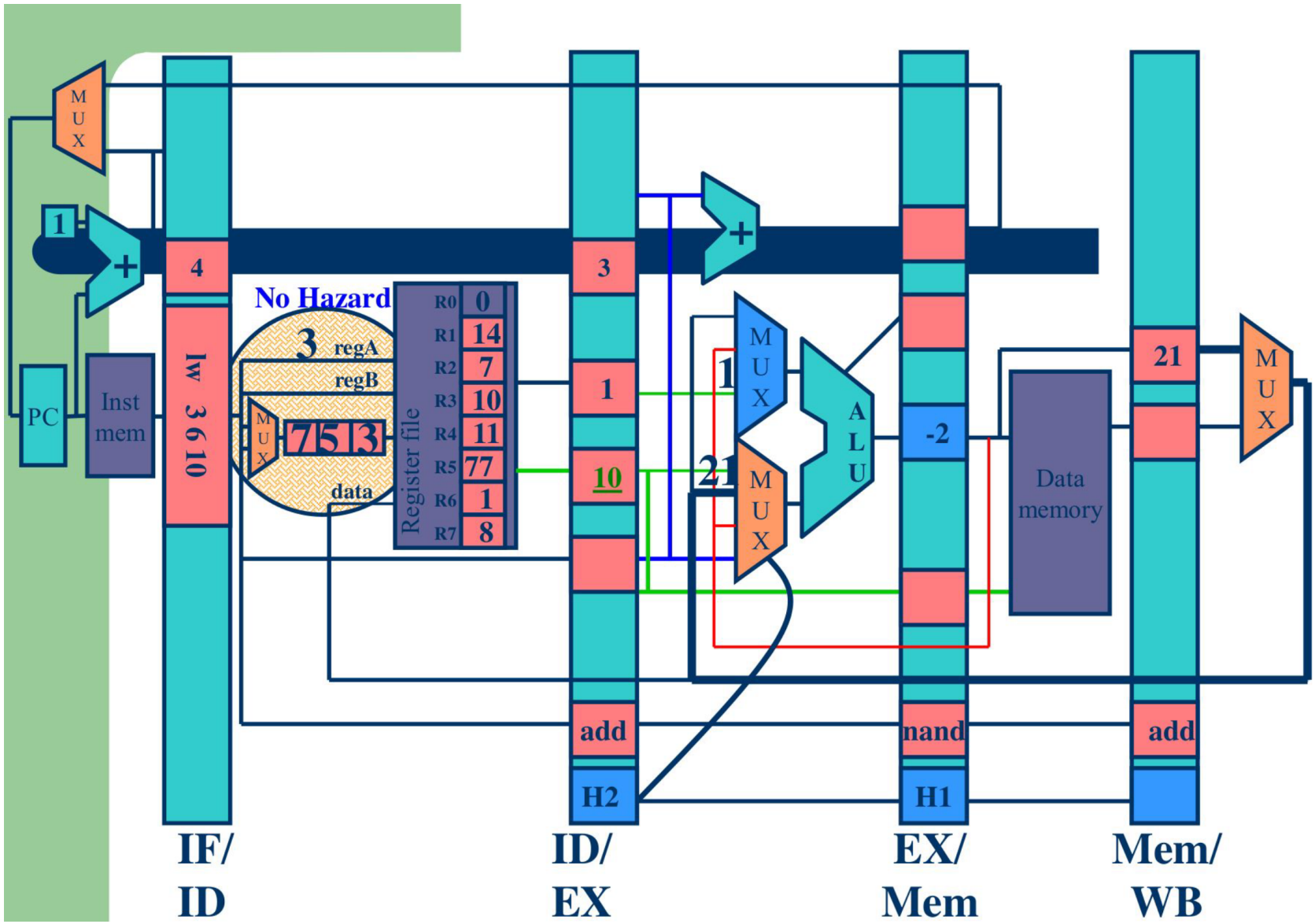
Lab4 Hints


MUX
1
+
3
add 6 3 7
PC
Inst mem
New Hazard
regA
3
regB
5 3
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
2
10
11
nand
H1
+
21
11
MUX
MUX
ALU
21
add
Data memory
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

Lab4 Hints


MUX
1
+
4
PC
Inst mem
lw 3 6 10
regA
regB
MUX
7 5 1 3
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
3
1
10
add
H2
MUX
MUX
ALU
-2
nand
H1
Data memory
21
add
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

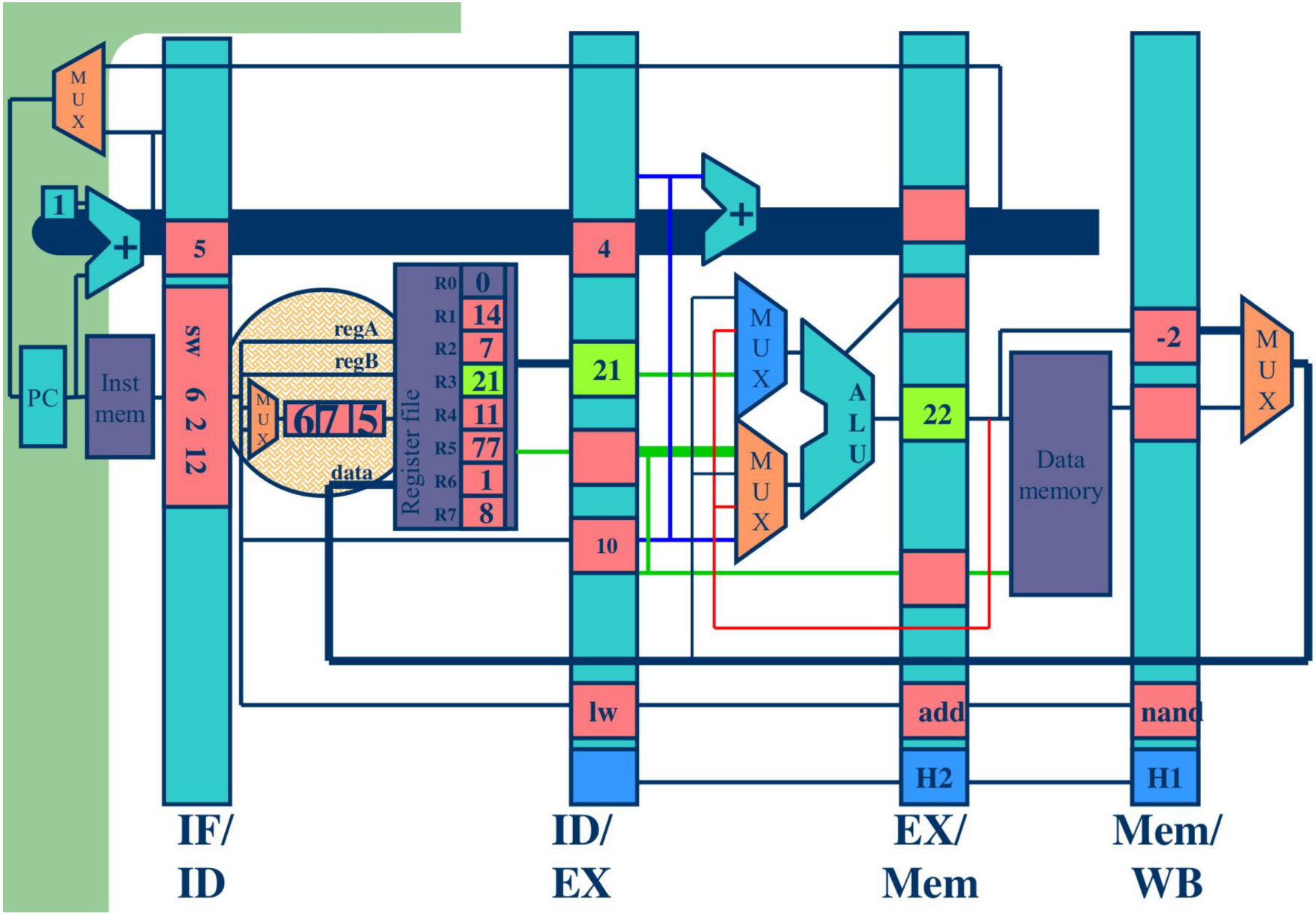
Lab4 Hints


MUX
1
+
PC
Inst mem
4
lw 3 6 10
No Hazard
3 regA
regB
MUX
7 5 3
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
3
1
10
add
H2
MUX
1
21
MUX
ALU
-2
nand
H1
Data memory
21
add
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

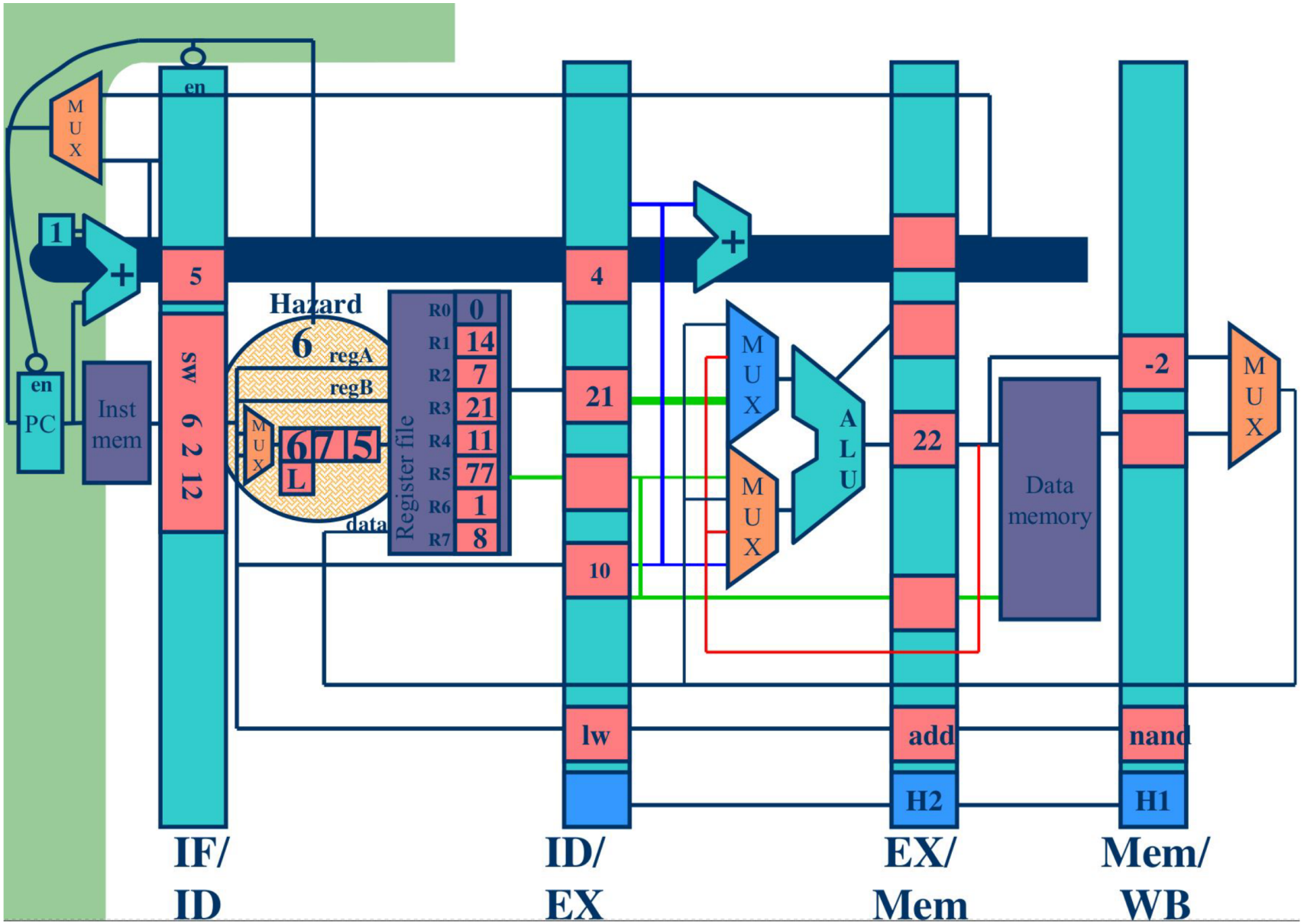
Lab4 Hints

- Branches should resolve in the same stage they are currently resolved in
- All forwarding must be to the EX stage, even if the data isn't needed until a later stage
- Any stalling due to data hazards must occur in the decode stage. (That is, if stalling is required the dependent instruction should stall in the decode stage.) Obviously, instructions following the stalling instruction in the IF stage will have to stay in the IF stage. Put another way, if you need to insert an invalid instruction, it should be inserted in the EX stage

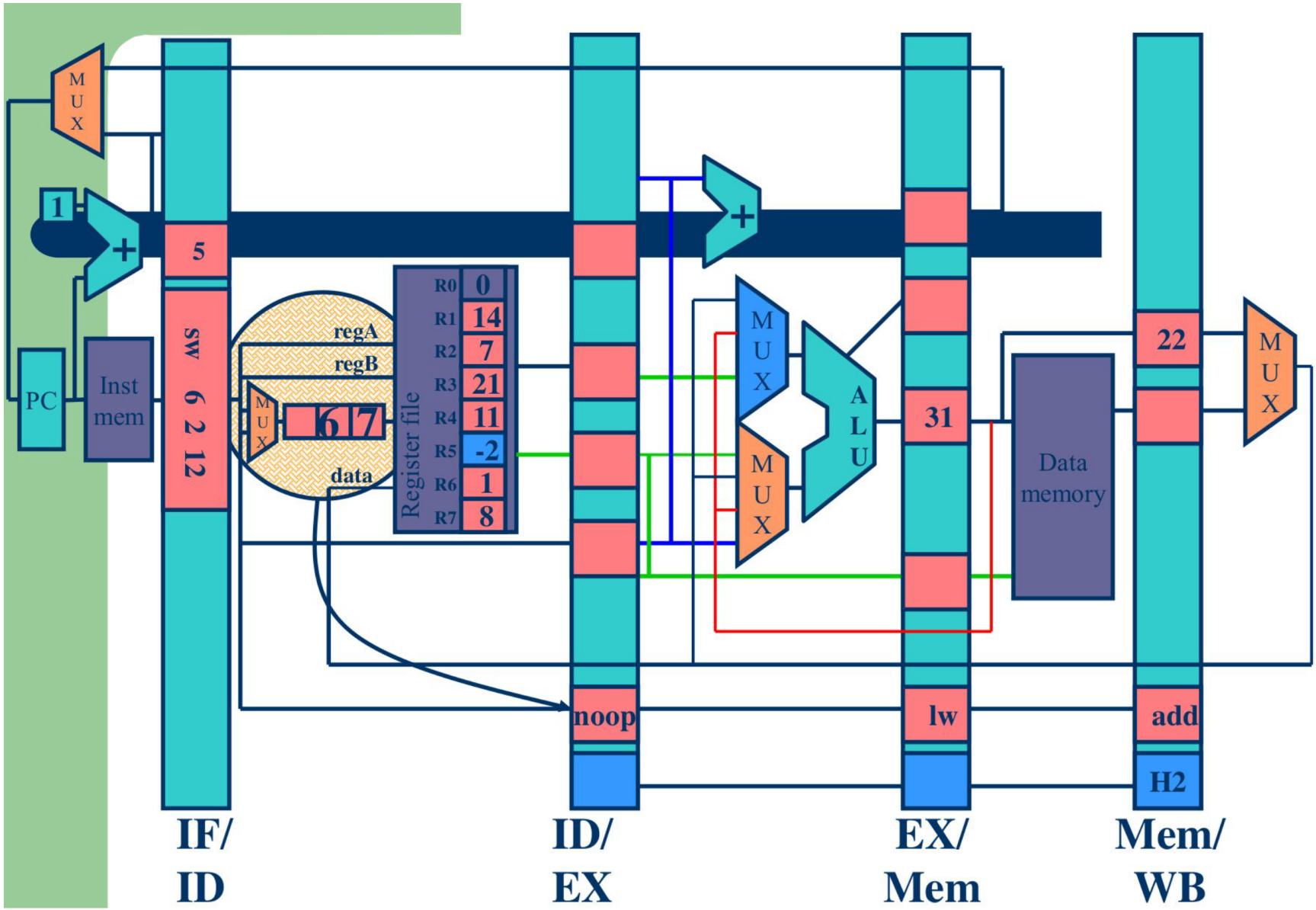
Lab4 Hints


MUX
1
+
5
PC
Inst mem
sw 6 2 12
regA
regB
MUX
6 7 5
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 77
R6 1
R7 8
4
21
10
lw
+
MUX
MUX
ALU
22
add
H2
Data memory
-2
nand
H1
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

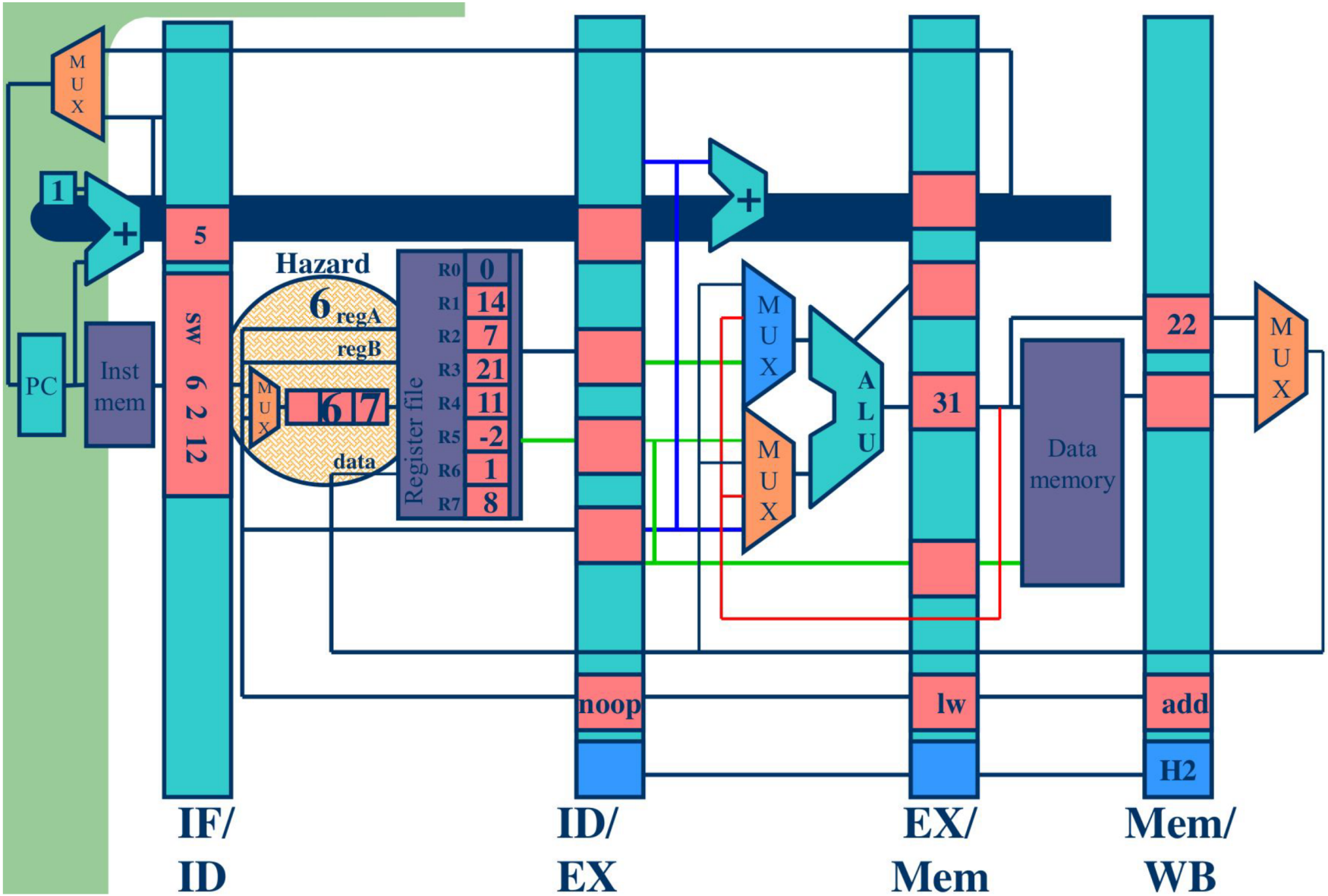
Lab4 Hints


MUX
en
1
+
5
SW 6 2 12
en
PC
Inst mem
Hazard
6
regA
regB
MUX
6 7 5
L
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 77
R6 1
R7 8
4
21
10
lw
MUX
MUX
ALU
22
add
H2
Data memory
-2
nand
H1
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

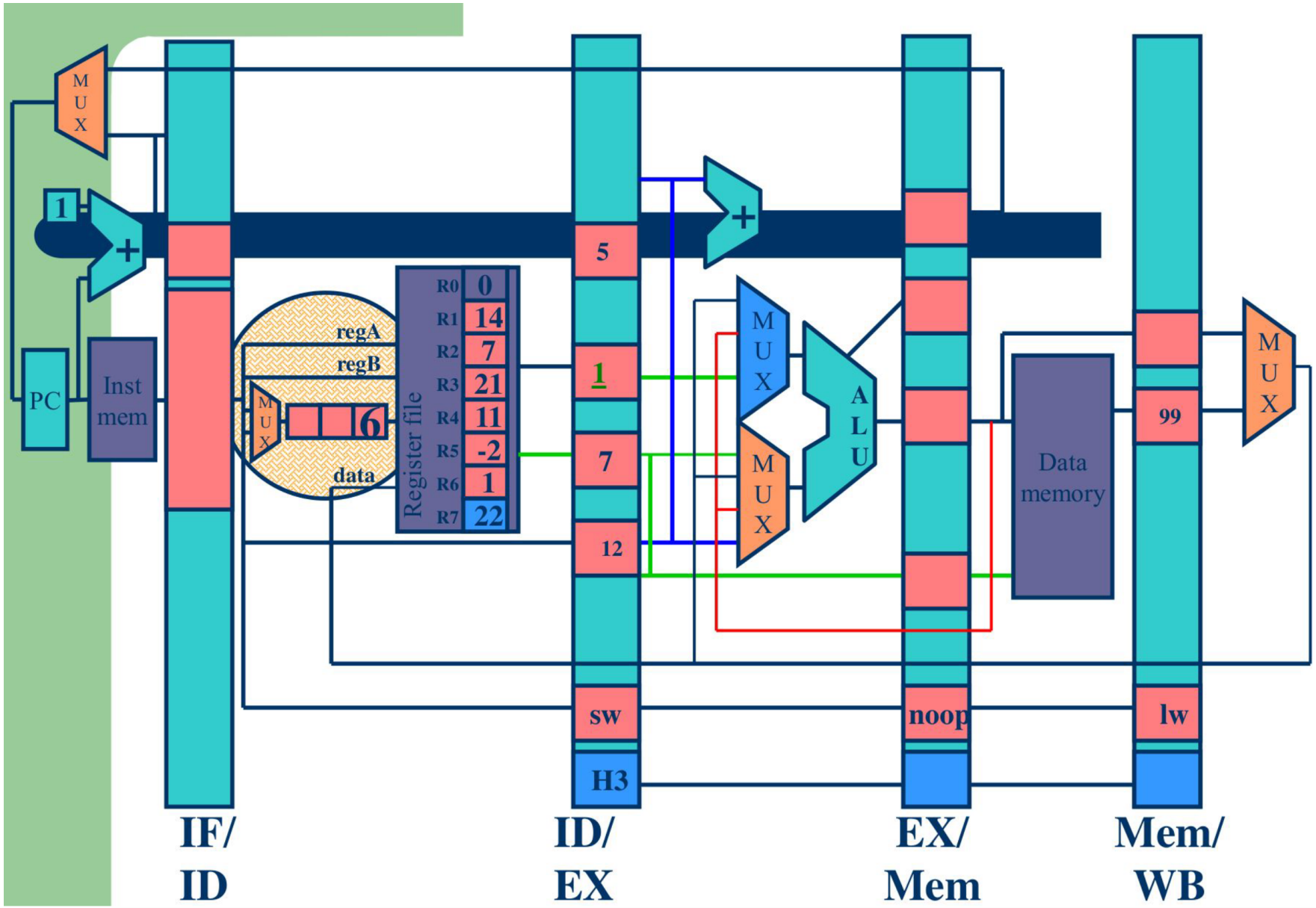
Lab4 Hints


MUX
1
+
5
PC
Inst mem
sw 6 2 12
regA
regB
MUX
6 7
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 1
R7 8
+
MUX
MUX
ALU
Data memory
22
MUX
31
noop
lw
add
H2
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

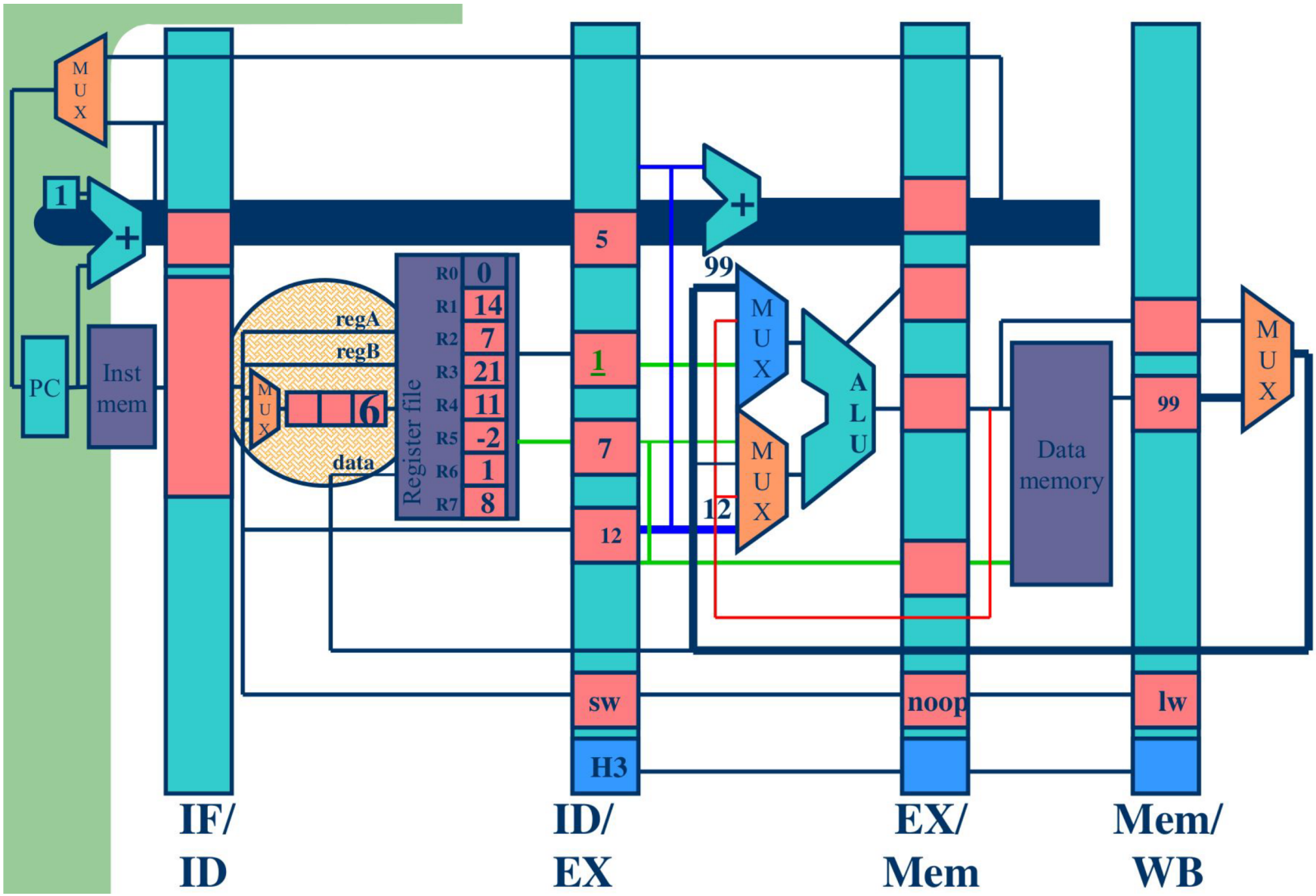
Lab4 Hints


MUX
1
+
5
sw 6 2 12
PC
Inst mem
Hazard
6 regA
regB
MUX
6 7
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 1
R7 8
+
MUX
MUX
ALU
31
Data memory
22
MUX
noop
lw
add
H2
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

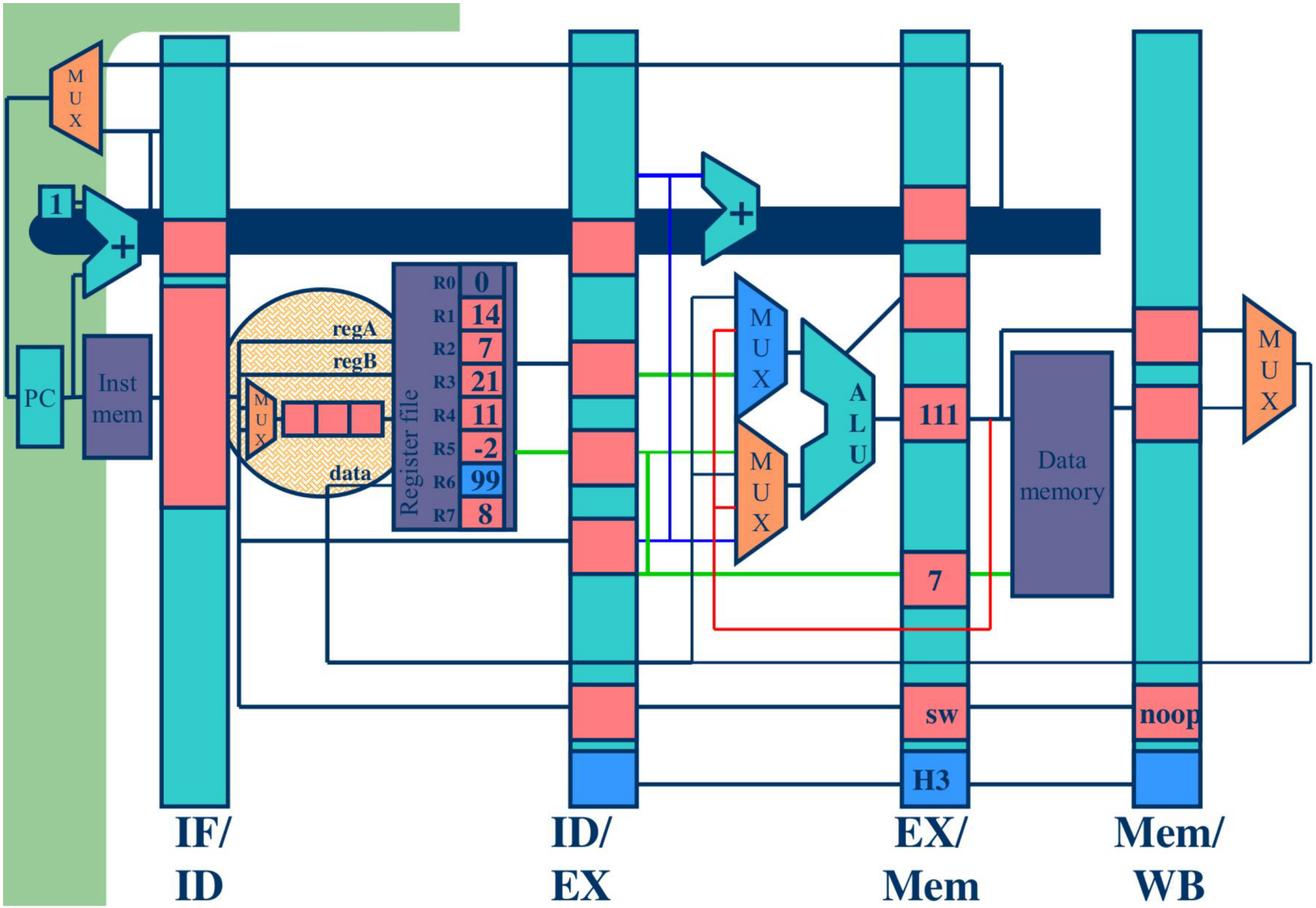
Lab4 Hints


MUX
1
+
PC
Inst mem
regA
regB
MUX
6
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 1
R7 22
5
1
7
12
sw
H3
+
MUX
MUX
ALU
Data memory
noop
99
lw
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

Lab4 Hints


MUX
1
+
PC
Inst mem
regA
regB
MUX
6
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 1
R7 8
5
1
7
12
sw
H3
99
MUX
MUX
12
ALU
+
Data memory
noop
99
lw
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

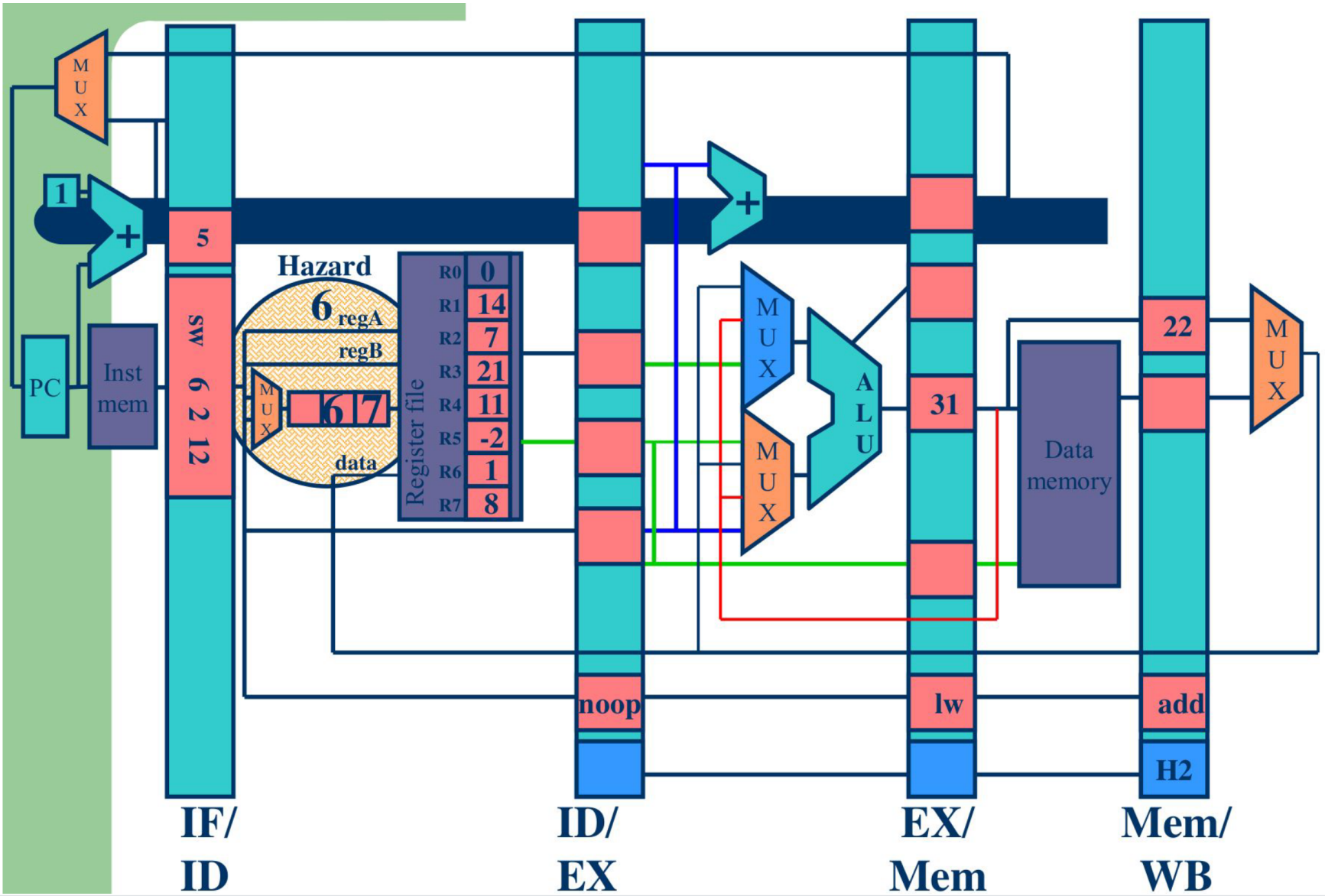
Lab4 Hints


MUX
1
+
PC
Inst mem
regA
regB
MUX
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 99
R7 8
MUX
MUX
ALU
+
111
7
sw
H3
Data memory
MUX
noop
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

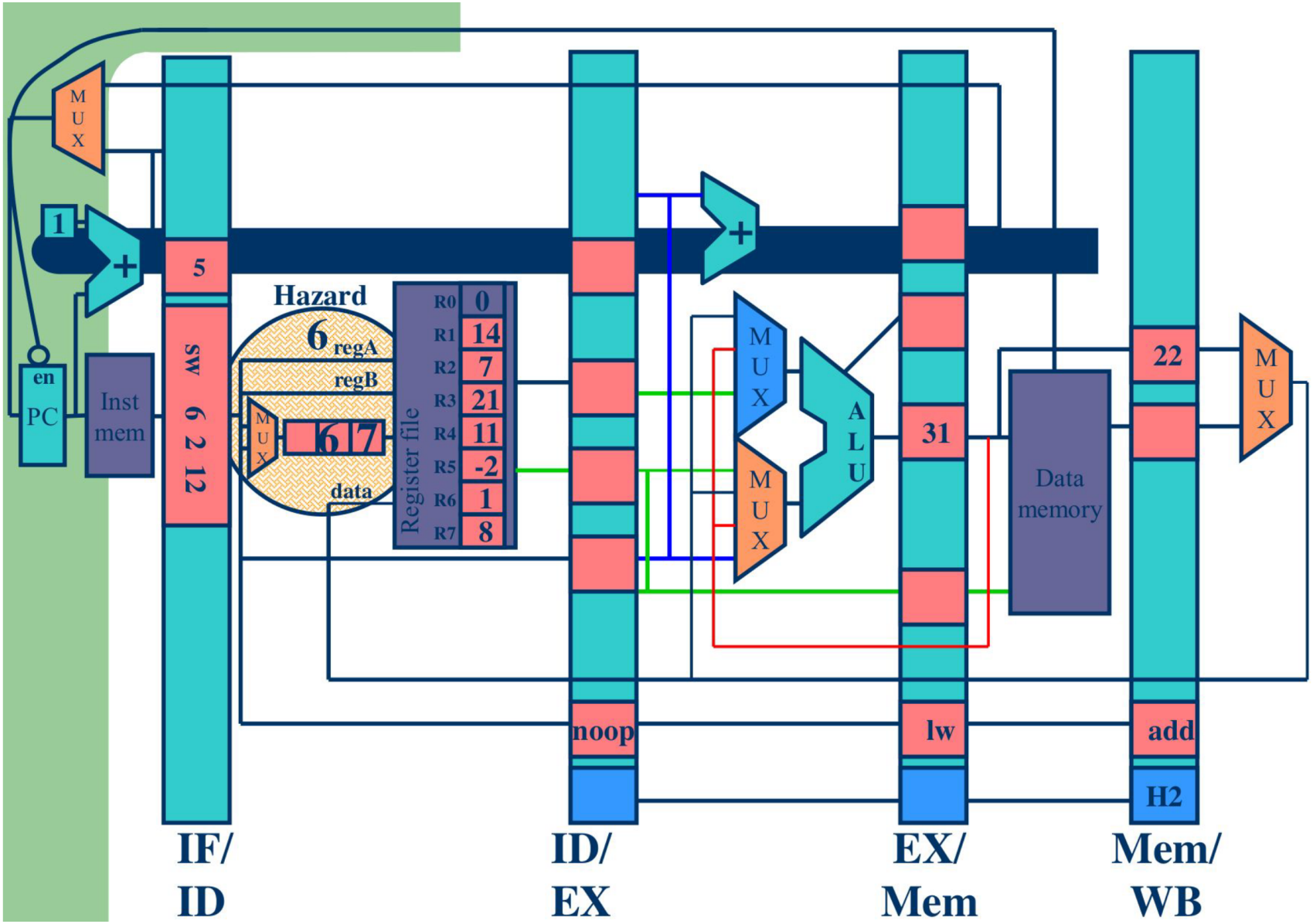
Lab4 Hints

- If you wish to insert a nop you must invalidate the instruction. Otherwise your CPI numbers will be wrong
- If there is a structural hazard in the memory, you should let the load/store go and have the fetch stage wait on getting memory

Lab4 Hints


MUX
1
+
5
SW 6 2 12
PC
Inst mem
Hazard
6 regA
regB
MUX
6 7
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 1
R7 8
+
MUX
MUX
ALU
31
Data memory
22
MUX
noop
lw
add
H2
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

Lab4 Hints


MUX
1
+
5
en
PC
Inst mem
sw 6 2 12
Hazard
6 regA
regB
MUX
6 7
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 1
R7 8
+
MUX
MUX
ALU
31
Data memory
22
MUX
noop
lw
add
H2
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

Implementation Hints

- Try to tackle one thing at a time
- Be careful of register 0!
- Be aware of where operand data is coming from
  - Not all instructions receive source data from ALU output.
- “Forward data into EX stage”
  - Essentially means widen muxes for ALU input, or add muxes for EX/MEM pipeline register inputs.
- Adding signals should be very easy if you use structs wisely

References

Jielun Tan, James Connolly (2021)

EECS470 Lab4-part2

Jon Beaumont (2021)

EECS470 Lab4

Thanks