Introduction to RISC-V

Jielun Tan, James Connolly, Ian Wrzesinski

Last updated 02/2024

Overview

- What is RISC-V
- Why RISC-V
- ISA overview
- Software environment
- Project 3 stuff

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
    - This is just marketing talk obviously, don’t get carried away…
- It originated in UC Berkeley, but now it has own its foundation with a large number of contributors
- Wide adoption within industry
    - Nvidia, Alibaba, Western Digital, etc. all have RV cores in their product
    - A fruit company, a letters company and others have developments
    - Many startups: SiFive, Esperanto, Tenstorrent, etc.

Why RISC-V

- Why not OpenRISC?
  - OpenRISC had condition codes and branch delay slots, which complicate higher performance implementations, fixed encoding immediates, no support for 2008 revision of floating standard, blah blah blah etc.
  - The reality is no one uses it so yeah
- MIPS although now open sourced…
  - MIPS now makes RISC-V products
  - MIPS is also very convoluted
- License issues for Arm…
- Only academia still deals with Alpha
  - Press F to pay respect to DEC, Compaq and HP

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 ISAs can be used for educational purposes if we just take a subset, however we'd have to build custom software around that
    - With RV, that support is built-in

Why RISC-V

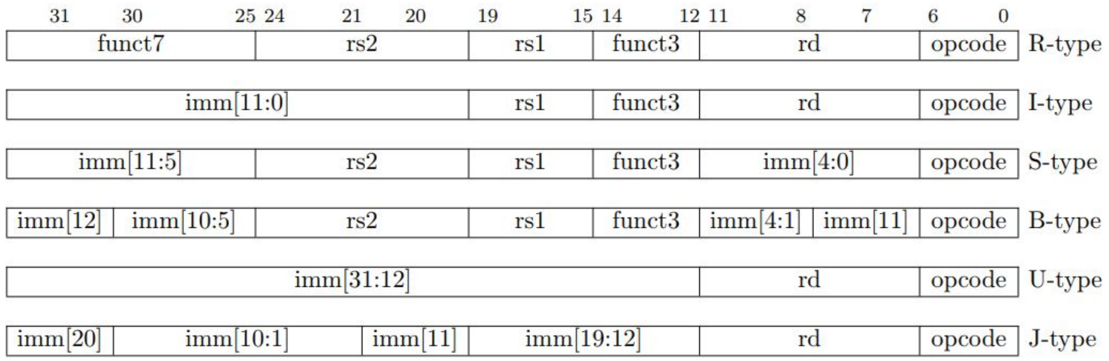
- Lets us explore more layers of the computing stack, mainly compilers and systems
- Can arbitrarily generate test cases, since we can just write in C now!
  - Easier for you to test
  - Easier for the staff to shuffle around test cases
  - Easier to generate large test cases that can actually benefit 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
- Does support misaligned memory access by default
    - You don’t have to worry about this, all tests should be compiled with strict alignment

ISA Overview - Instruction Formats

- 6 different encoding format for instructions
- A looooooooot of pseudoinstructions
    - You can read about all of them in the specification here (highly recommended, the base integer and multiplication extension aren’t long at all)

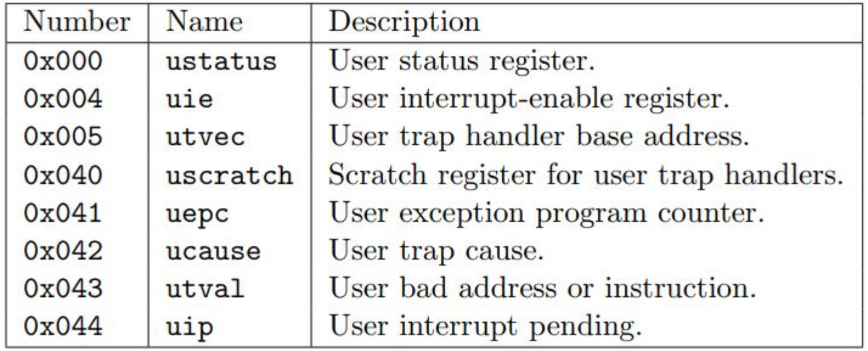


31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

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

ISA Overview - CSRs

- Also a list of Control Status Registers (CSR)
  - Many are important if interrupt support is needed
  - You can choose to implement them in the final project
  - Here are some examples, you can read more about them in the privileged spec

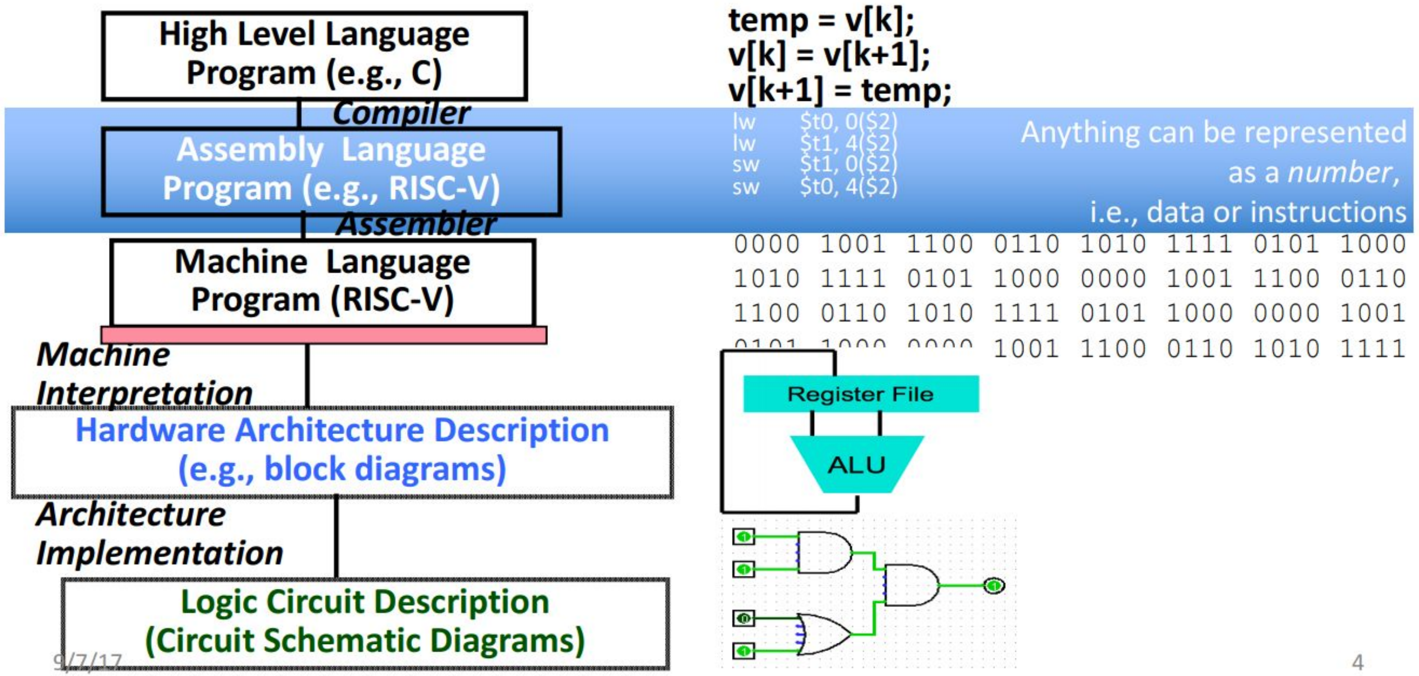


Number	Name	Description
0x000	`ustatus`	User status register.
0x004	`uie`	User interrupt-enable register.
0x005	`utvec`	User trap handler base address.
0x040	`uscratch`	Scratch register for user trap handlers.
0x041	`uepc`	User exception program counter.
0x042	`ucause`	User trap cause.
0x043	`utval`	User bad address or instruction.
0x044	`uip`	User interrupt pending.

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 (omegalul)
- Z series, basically all future extensions since they ran out of letters

Software Environment



High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

Assembler

Machine Language Program (RISC-V)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

9/7/17

4

Software Environment

Assembly vs. High-level Language

Software Environment

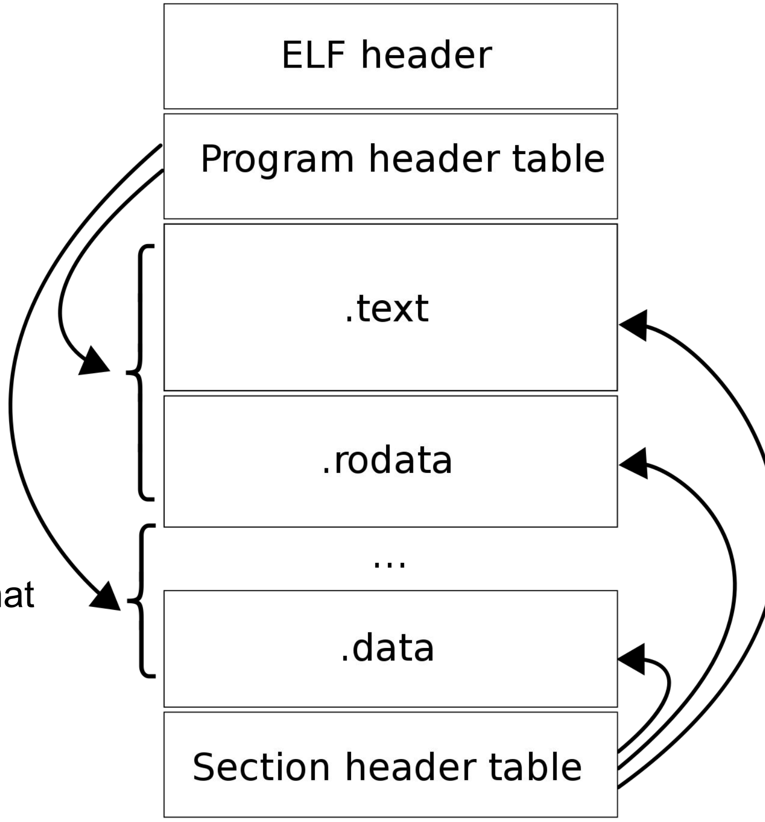
- Why do I remotely even care about software in a hardware class
  - Believe me, it’s important
  - Architecture is the bridge between the two (insert preaching)
  - Software support dictates hardware adoption
    - There’d be nothing to run on hardware without software just saying…
- For RISC-V, we 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, 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` - disassembler
  - `objcopy` - copies one object file to another, can change link formats
  - `g++` - don’t use this, no linker support
  - a lot more that you can explore yourself...

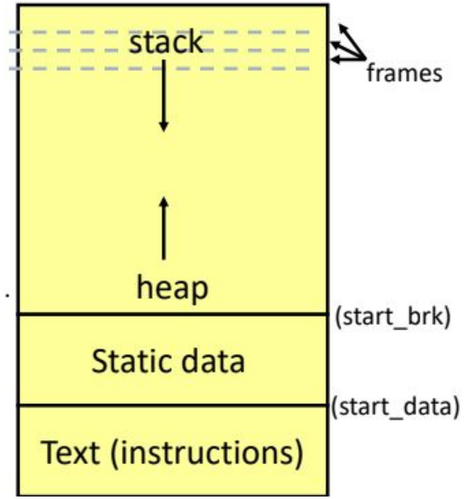
Software Environment - ELF

- 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

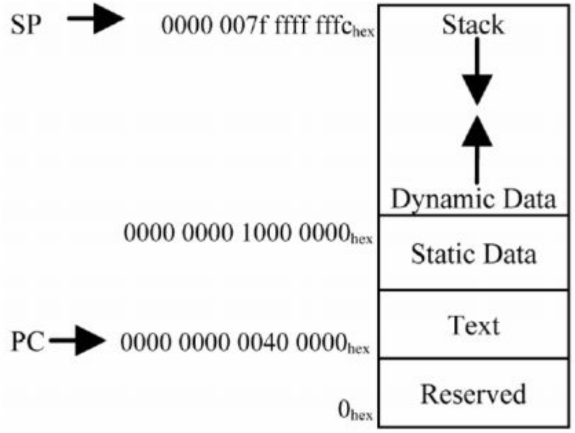
- Flashback to 370 or whatever computer organization class you had
- What does the memory space for a program look like?
- Stack for statically allocated variables, pointer decrements
- Heap for dynamic memory, pointer increments



Software Environment - Program Space

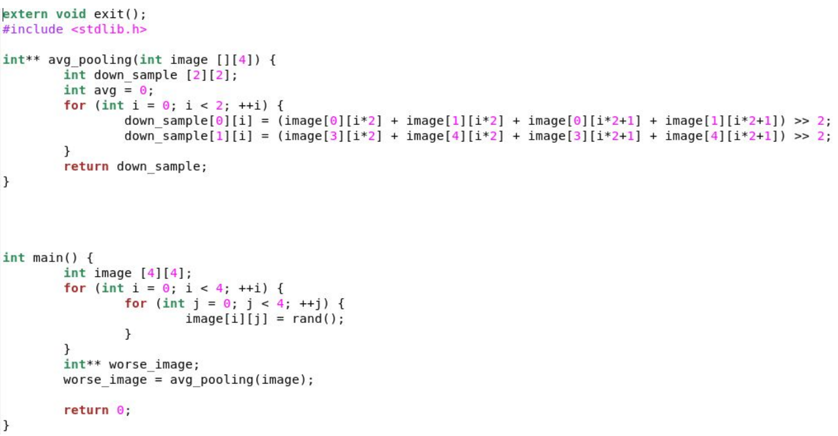
- Example of program space allocation for arm processors->
  - The linker allocate the memory space
- In general memory addresses around 0x0 are precious
  - Some peripherals on the serial buses can only talk to

    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


SP
0000 007f ffff fffc$_{hex}$
Stack
Dynamic Data
0000 0000 1000 0000$_{hex}$
Static Data
Text
PC
0000 0000 0040 0000$_{hex}$
Reserved
0$_{hex}$

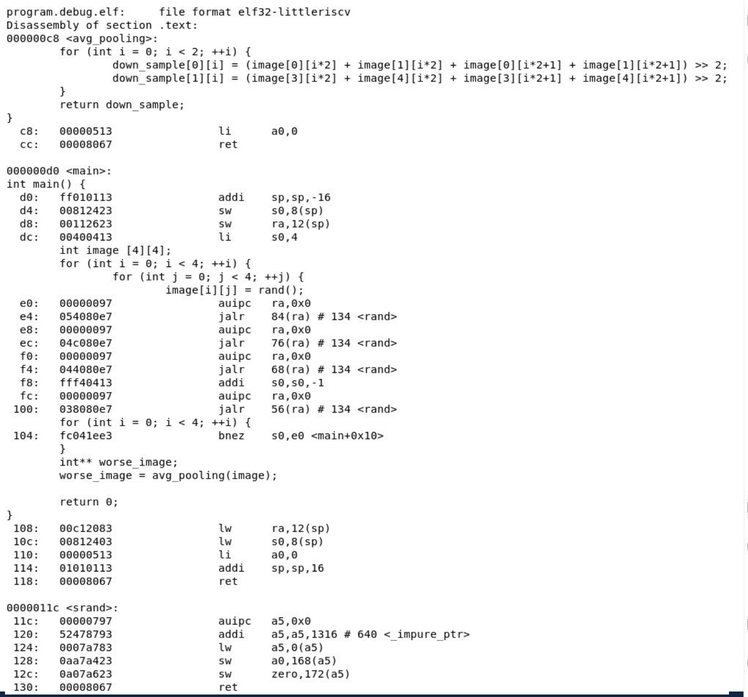
Software Environment - Program Space

C Source Code



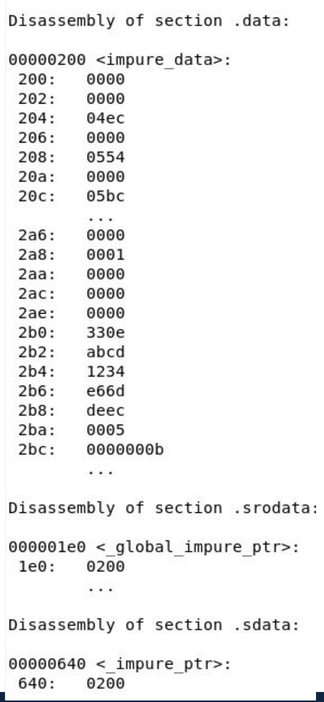
```c
extern void exit();
#include <stdlib.h>

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;
}
```

RISC-V Assembly (dump files)



```
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
```

Machine Code

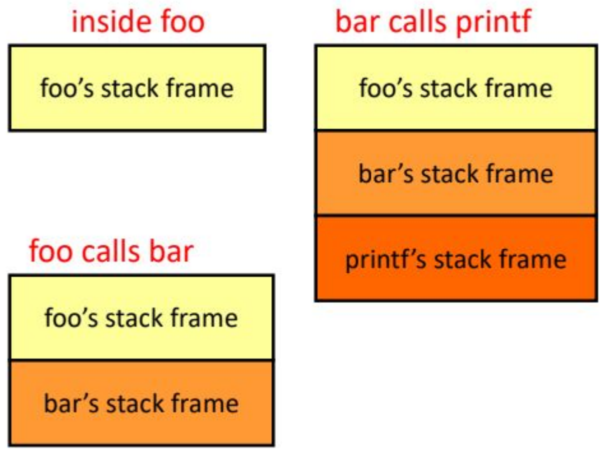


```
Disassembly of section .data:

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

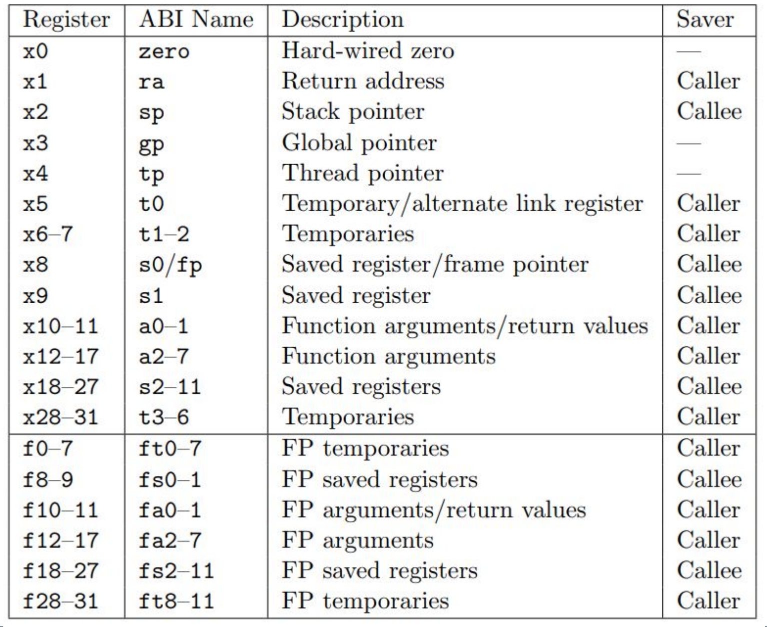
- Every time there's a function call, have a frame pointer that saves the previous stack pointer
- Caller/callee save the variables

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

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



Software Environment - ABI

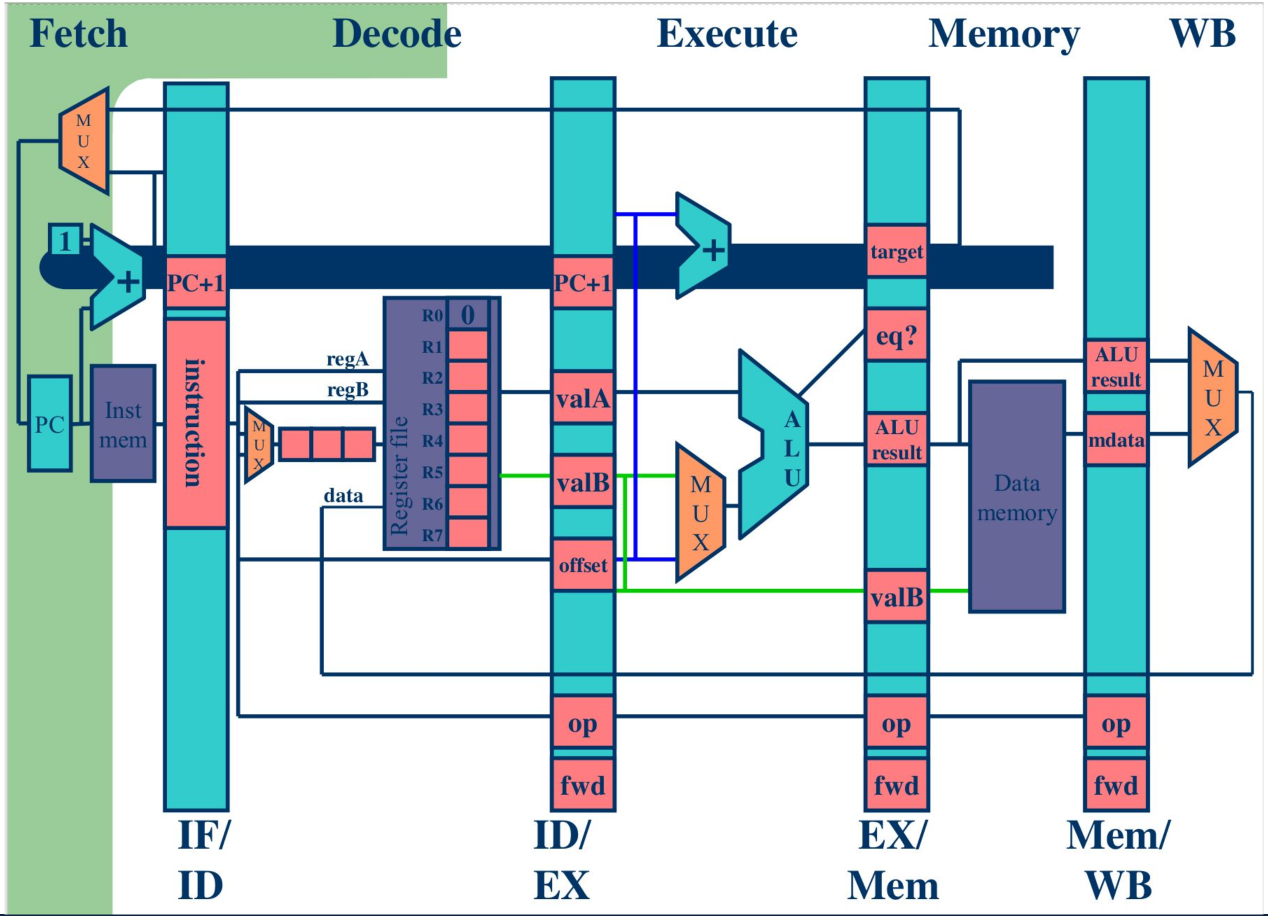
- Registers aren't just registers, each of them has a meaning
- Such concept is called Application Binary Interface (ABI)



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

Project 3 - VeriSimple Pipeline

- Same 5 stage in-order pipeline as in class
  - No hazard detection or forwarding logic
  - 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
- Programs still run because only one instruction is allowed in the pipeline at a time



Fetch
Decode
Execute
Memory
WB
MUX
1
+
PC
Inst mem
PC+1
instruction
regA
regB
data
Register file
R0
R1
R2
R3
R4
R5
R6
R7
0
MUX
PC+1
valA
valB
offset
op
fwd
+
MUX
ALU
target
eq?
ALU result
valB
op
fwd
Data memory
ALU result
mdata
op
fwd
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

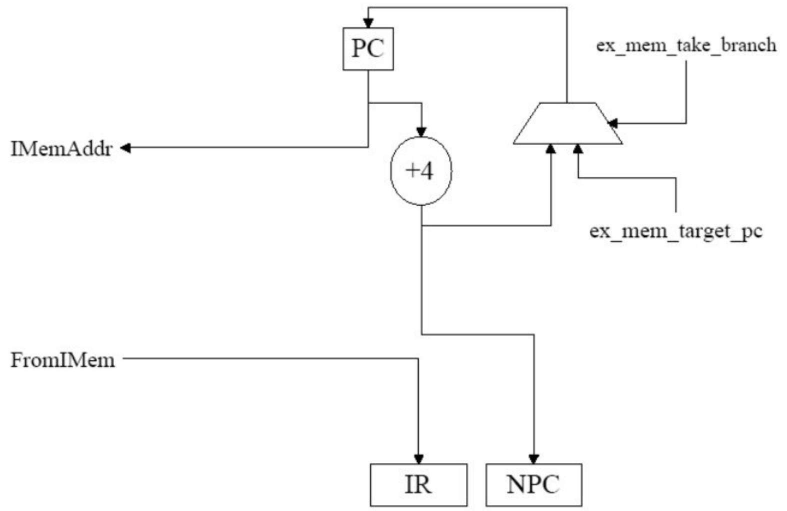
VeriSimple Pipeline

- 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

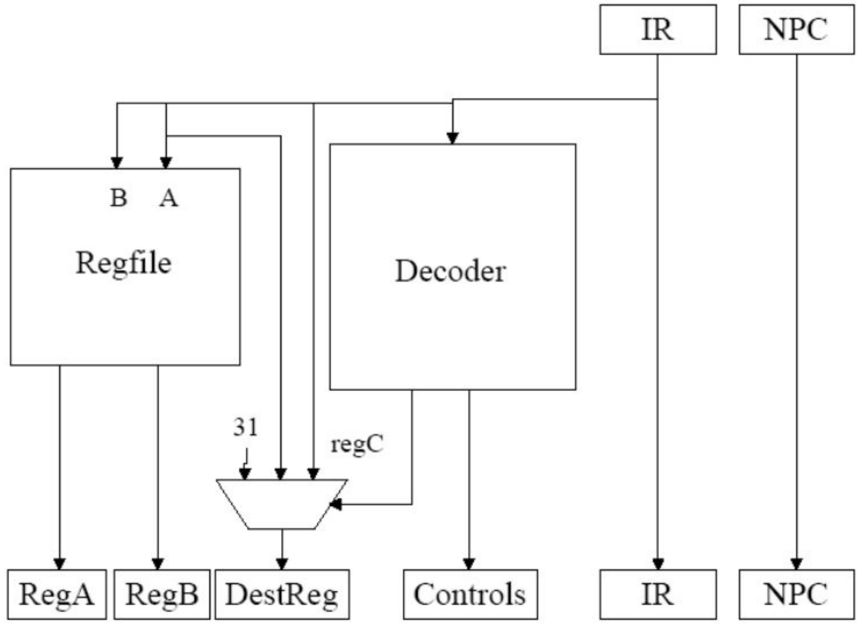
Pipeline Stages

- These diagrams are outdated, but still helpful

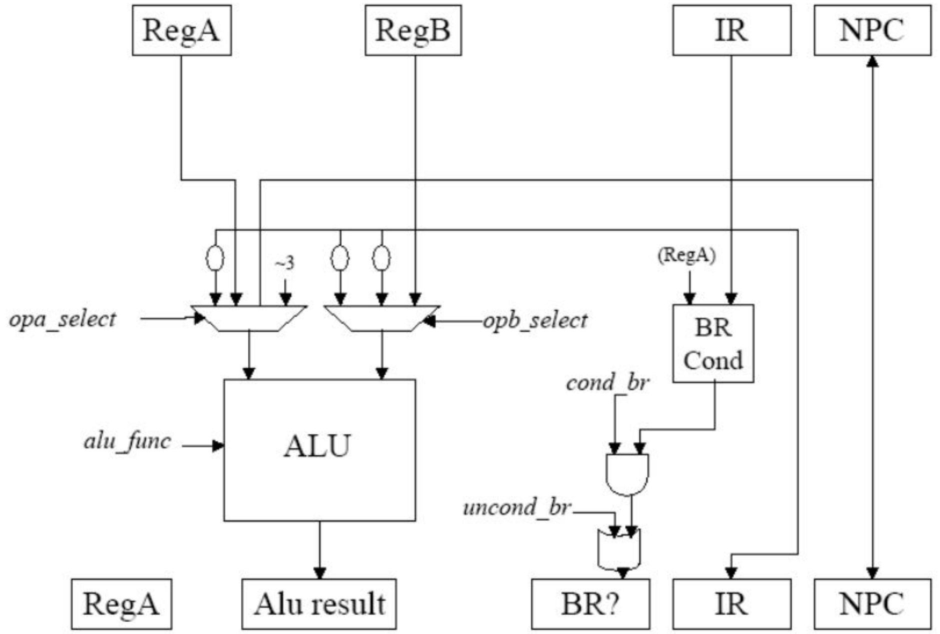
Pipeline Stages - Fetch



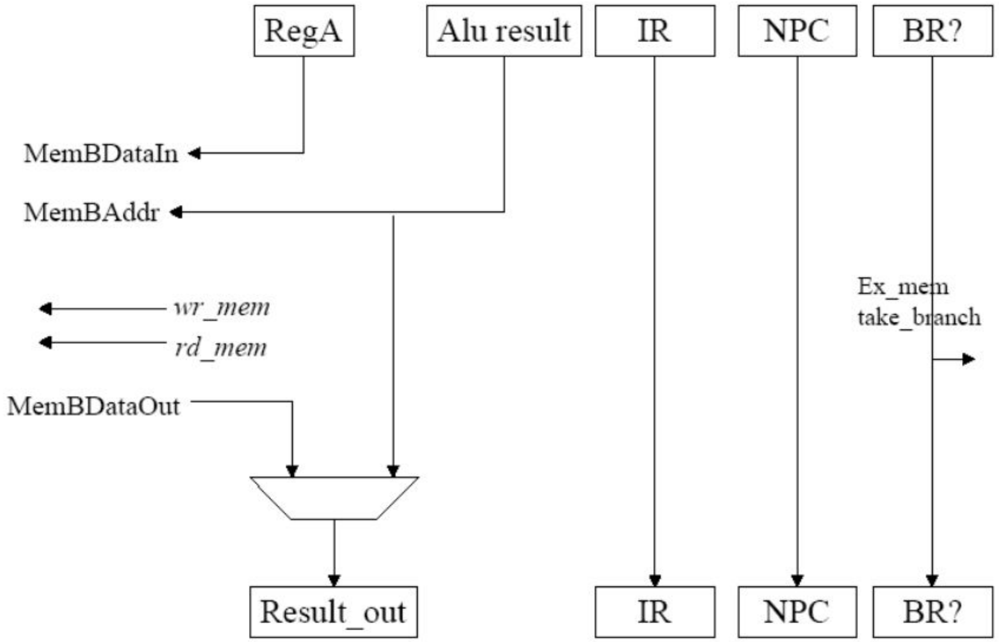
Pipeline Stages - Decode



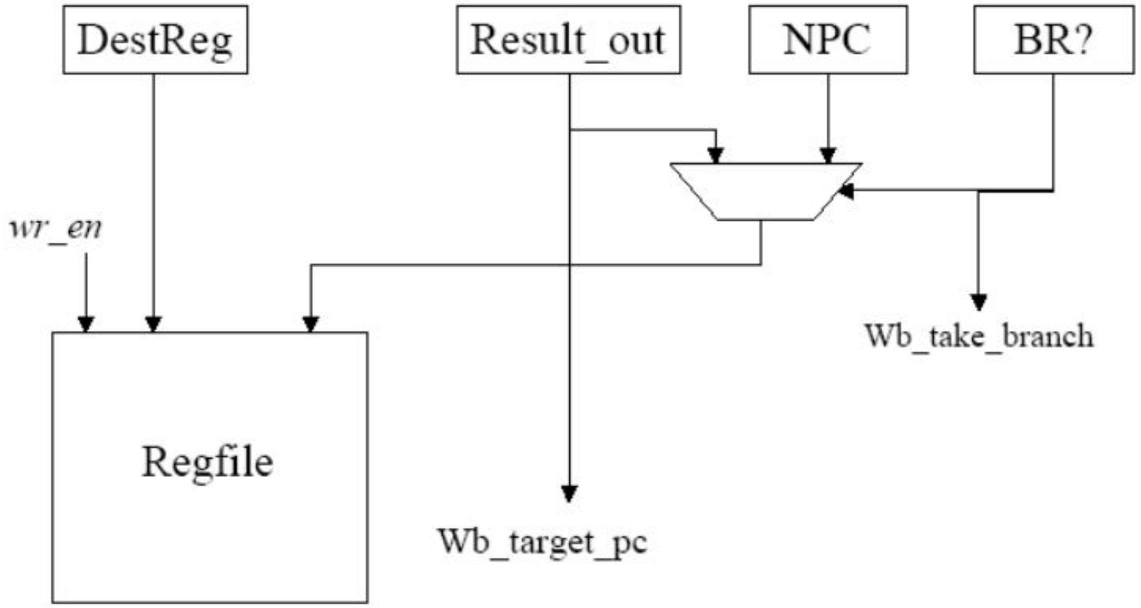
Pipeline Stages - Execute



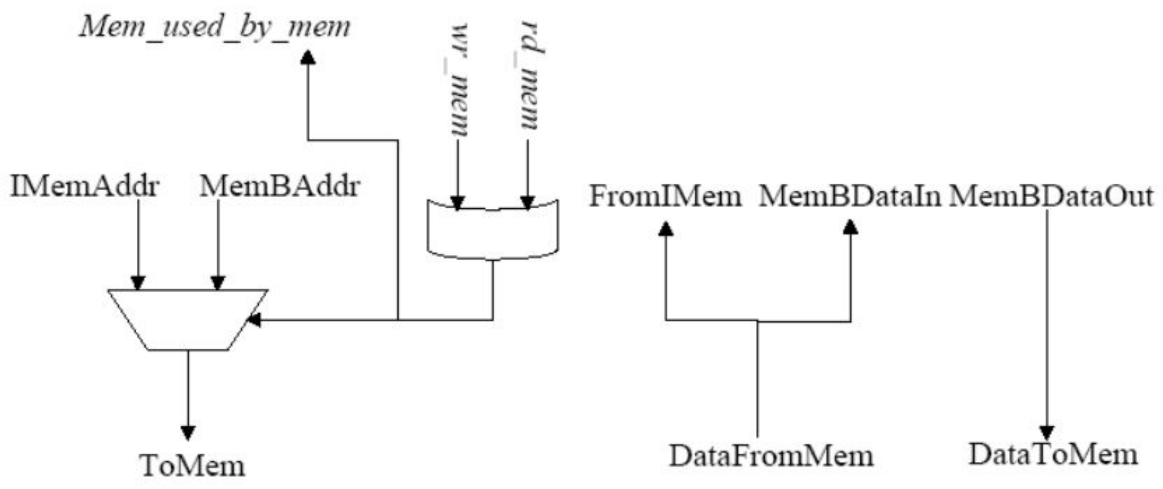
Pipeline Stages - Memory



Pipeline Stages - Writeback



Pipeline Stages - Memory Arbitration



Directory Structure

- `Makefile` – You’ve seen multiple times now
  - Compiles the verilog executables `simv` and `syn_simv` (and `vis_simv`)
  - Also contains targets for compiling/linking C and assembly programs
  - More on the next slide
- `verilog/` – Verilog code directory you should edit and change
- `test/` – directory with the testbench, memory, and pipeline printing code
- `synth/` – directory where synthesis output will be created. Also where the synthesis script is
- `programs/` – test programs, both C code and assembly. Where `.mem` files are compiled
  - You will write your own `test_[12345].s` assembly files here!
- `output/` – where and `.out`, `.cpi`, `.wb`, and `.ppln` files are created

Makefile Targets

- `make programs/<my program>.mem` – compile `<my program>` into a machine code `.mem` file
- `make <my program>.out` – run `<my program>` on `simv` by loading the `.mem` file into memory

- Ex: for the program, `programs/no_hazard.s`, run '`make no_hazard.out`'
- For the program, `programs/omegalul.c`, run '`make omegalul.out`'
- This creates `<my program>.out`, `<my program>.cpi`, `<my program>.wb`, and `<my program>.ppln` in the `output/` directory

- `make <my_program.dump>` – create `.dump_x` and `.dump_abi` dump files in `programs/`
- `.dump_x` has numeric register names: x0, x1, x2, etc.
- `.dump_abi` has RISC-V ABI register names: sp, ra, a0, t0, etc.

- `make <my program>.verdi` – run the program on `simv` with verdi
- `make <my program>.syn.verdi` – run the program on `syn_simv` with verdi
- `make <my program>.vis` – run the VTUBER visual debugger, an extremely useful tool for project 3

- `make simv` – create the verilog executable `simv` from the testbench/sources
- `make syn_simv` – create the synthesis executable

Checking Your Output

- `output/<my_program>.out` - $display() output of the testbench - contains final memory status
- `output/<my_program>.cpi` - Final CPI and total time running
- `output/<my_program>.wb` - Register file writeback at each PC
- `output/<my_program>.ppln` - Pipeline output file of which PC/instruction is in each stage as well as activity to/from memory

We've given you three correct outputs for project 3 in the `correct_out/` directory. However, they test separate things, and neither contains every combination of hazards.

Use the program `diff` to check correctness:

```
make no_hazard.out
```

```
diff output/no_hazard.wb correct_out/no_hazard.wb
diff output/no_hazard.ppln correct_out/no_hazard.ppln
grep ‘@@@’ output/no_hazard.out | diff correct_out/no_hazard.out
```

(Try with: `alias diff=“git diff --no-index”`)

Writing your own assembly unit tests

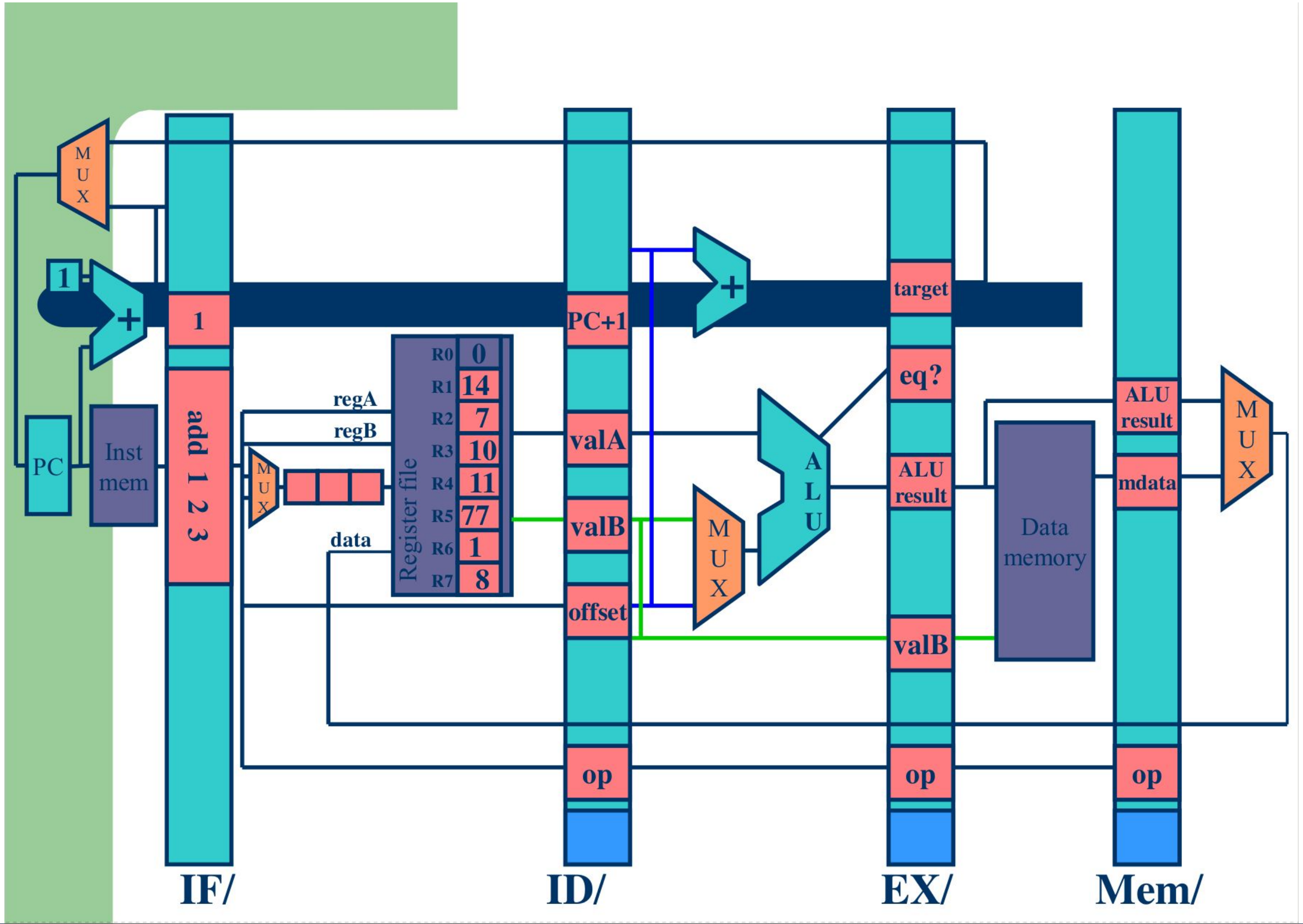
- You will write up to 5 assembly unit tests matching `programs/test_[12345].s`
- These must expose either a correctness or CPI bug in 4 buggy processors
- Copy the assembly from the existing assembly files in `programs/`
- Look at each of the hazard bullet points in the project spec
- Some internet assembly resources (not affiliated):
    - https://projectf.io/posts/riscv-arithmetic/
    - https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md
- Notes:
    - Don’t write anything that has misaligned memory access in assembly
    - Compiler won’t generate any for C

Project 3 Goals

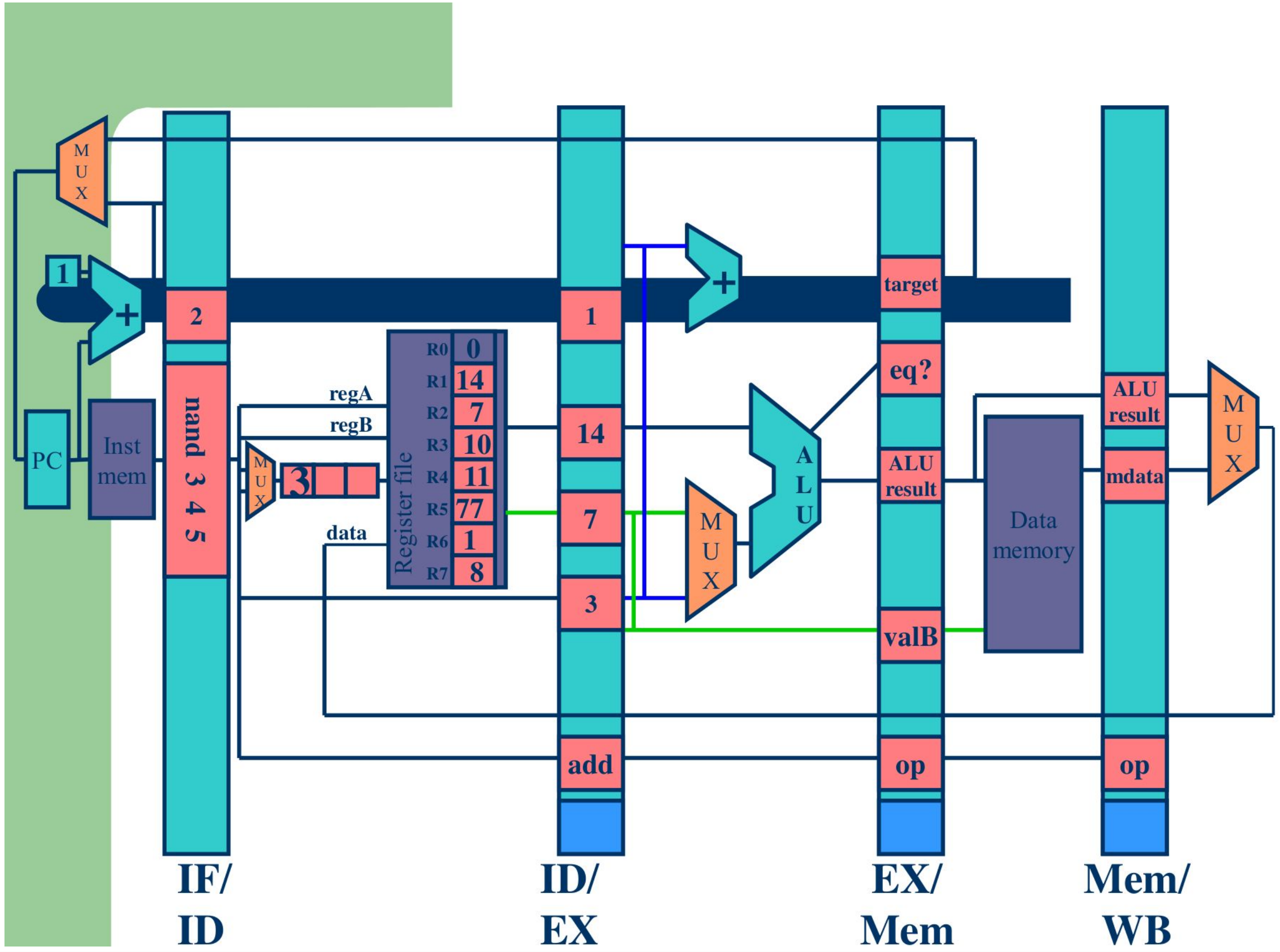
- Branches should resolve in the same stage they are currently resolved in
- All forwarding must be to the EX stage, even if the data isn’t needed until a later stage

Sample Code

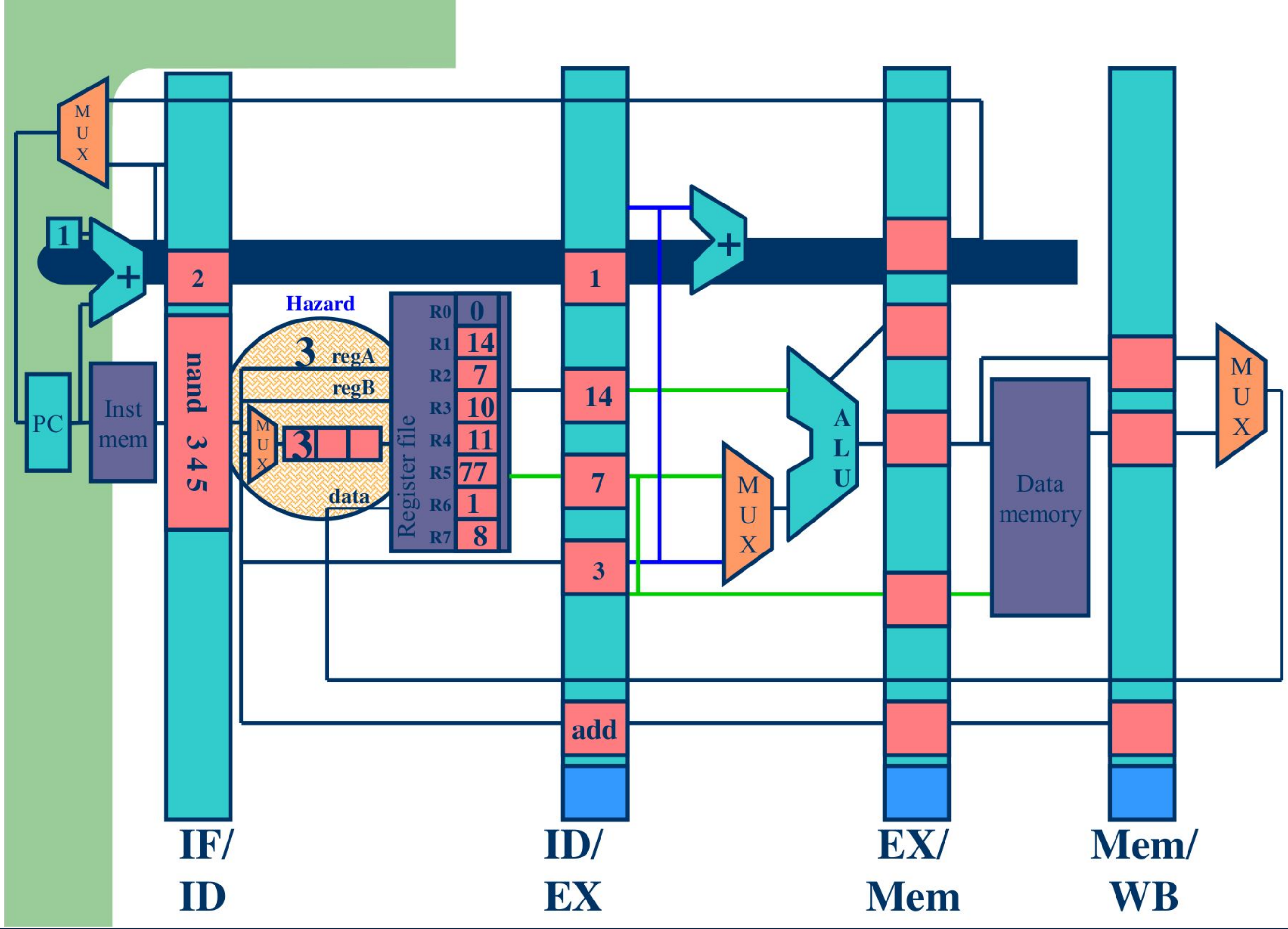
```
add    1  2  3    ; reg 3 = reg 1 + reg 2
nand   3  4  5    ; reg 5 = reg 3 ~& reg 4
add    6  3  7    ; reg 7 = reg 6 + reg 3
lw     3  6  10   ; reg 6 =  Mem[reg 3 + 10]
sw     6  2  12   ; Mem[reg6+12] = reg 2
```

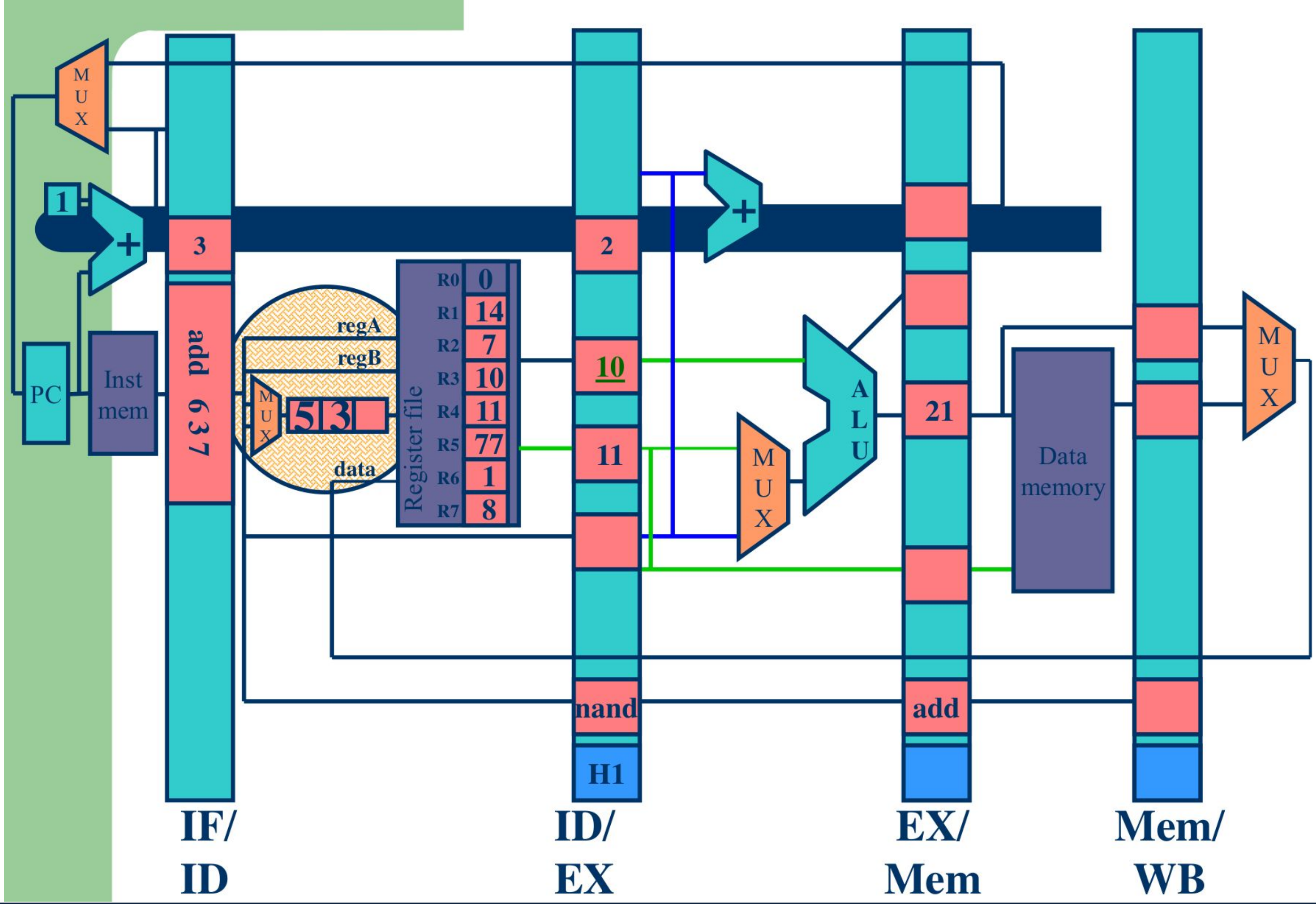


M U X
1
+
1
PC+1
+
target
PC
Inst mem
add 1 2 3
regA
regB
M U X
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
valA
valB
offset
M U X
A L U
eq?
ALU result
valB
Data memory
ALU result
mdata
M U X
op
op
op
IF/
ID/
EX/
Mem/

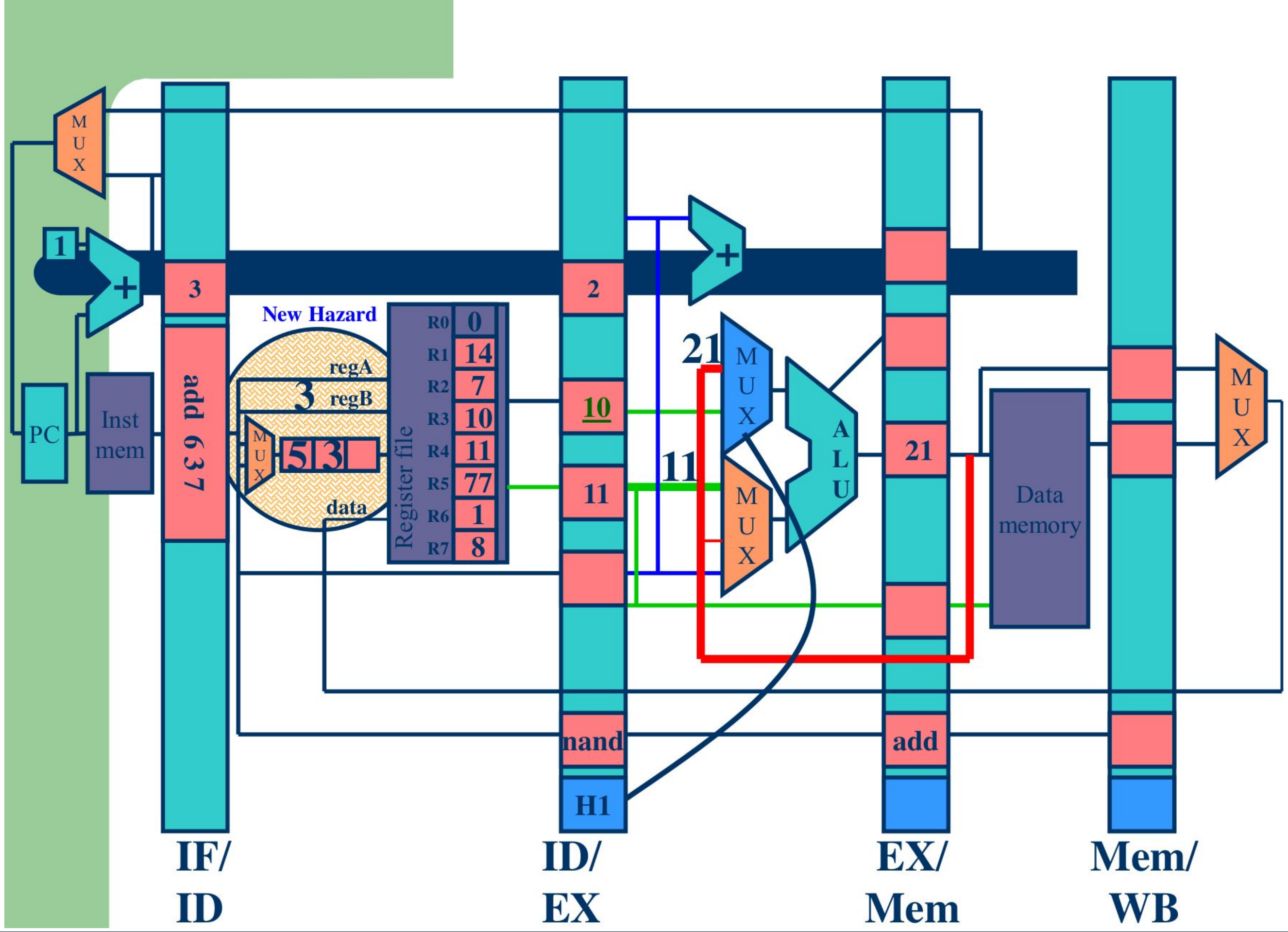


MUX
1
+
2
PC
Inst mem
nand 3 4 5
regA
regB
MUX
3
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
1
14
7
3
add
+
MUX
ALU
target
eq?
ALU result
valB
op
Data memory
ALU result
mdata
op
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

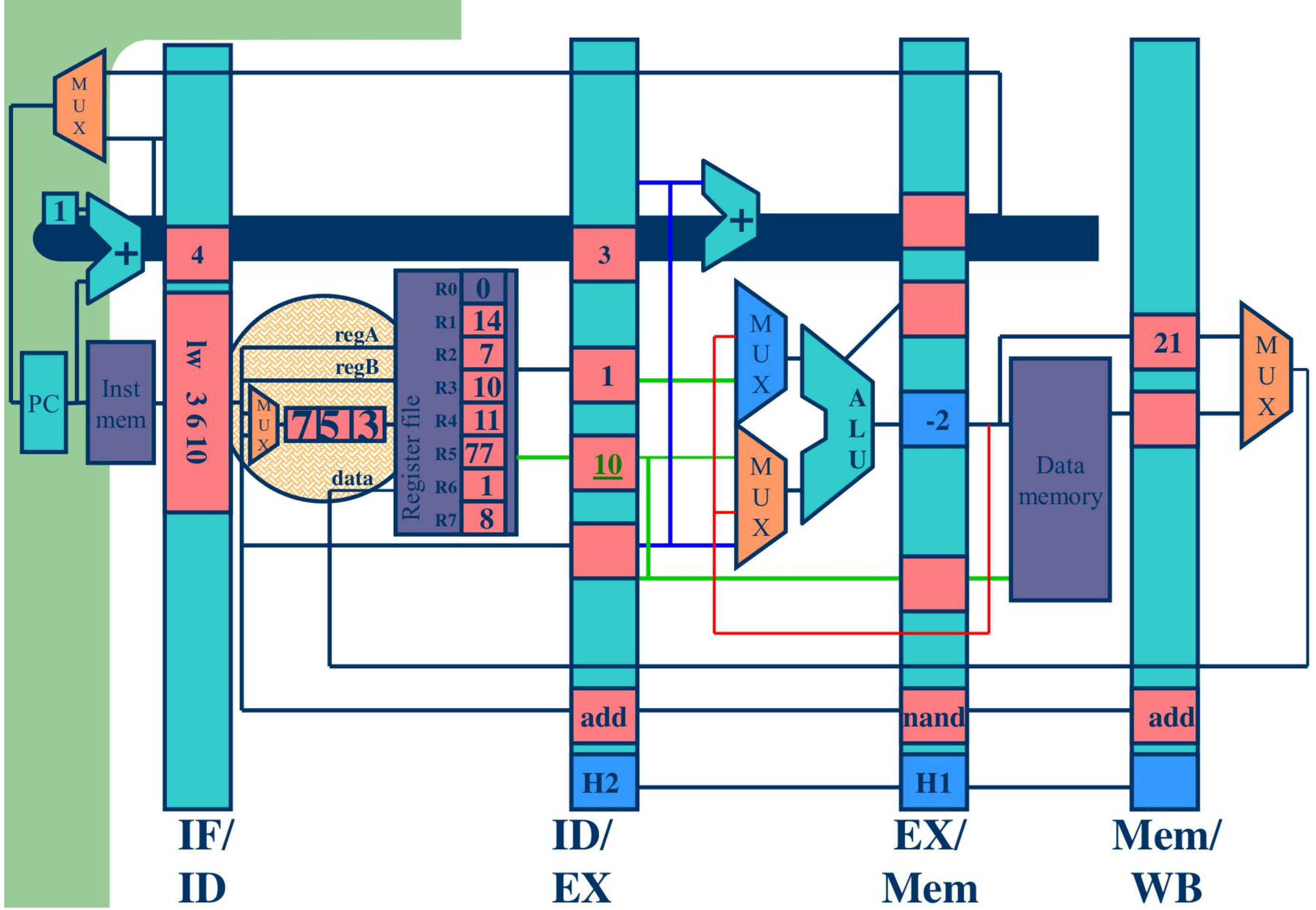

M U X
1
+
2
Hazard
3
regA
regB
M U X
3
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
PC
Inst mem
nand 3 4 5
1
14
7
3
add
+
M U X
A L U
Data memory
M U X
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB



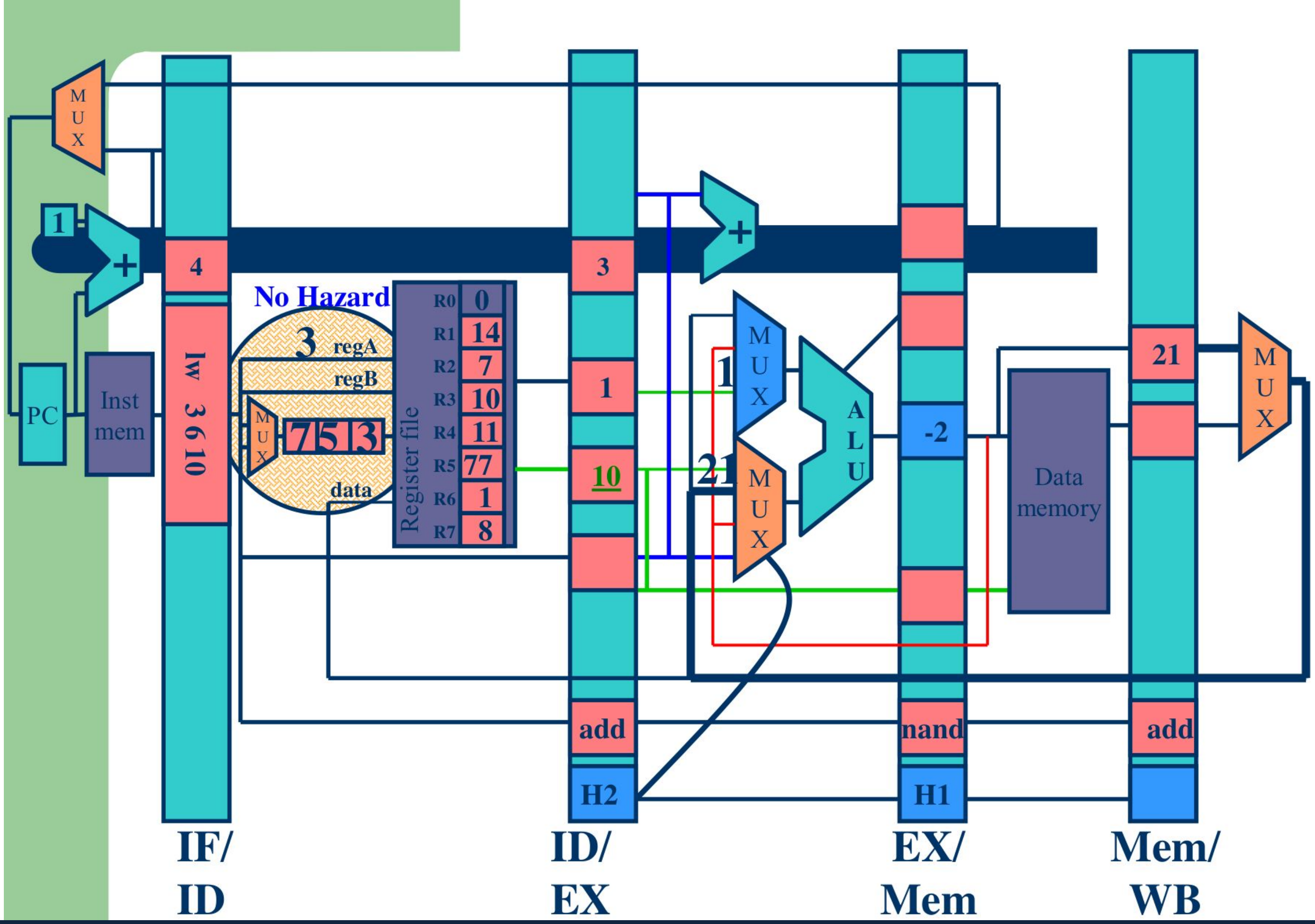
MUX
1
+
3
2
+
add 6 3 7
regA
regB
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
Register file
MUX
5 3
data
PC
Inst mem
10
11
MUX
ALU
21
Data memory
MUX
nand
add
H1
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB



M U X
1
+
3
add 6 3 7
PC
Inst mem
New Hazard
regA
3
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
+
21
11
M U X
M U X
A L U
21
add
Data memory
M U X
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB



MUX
1
+
4
3
PC
Inst mem
lw 3 6 10
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
1
10
+
MUX
MUX
ALU
-2
Data memory
21
MUX
add
nand
add
H2
H1
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB



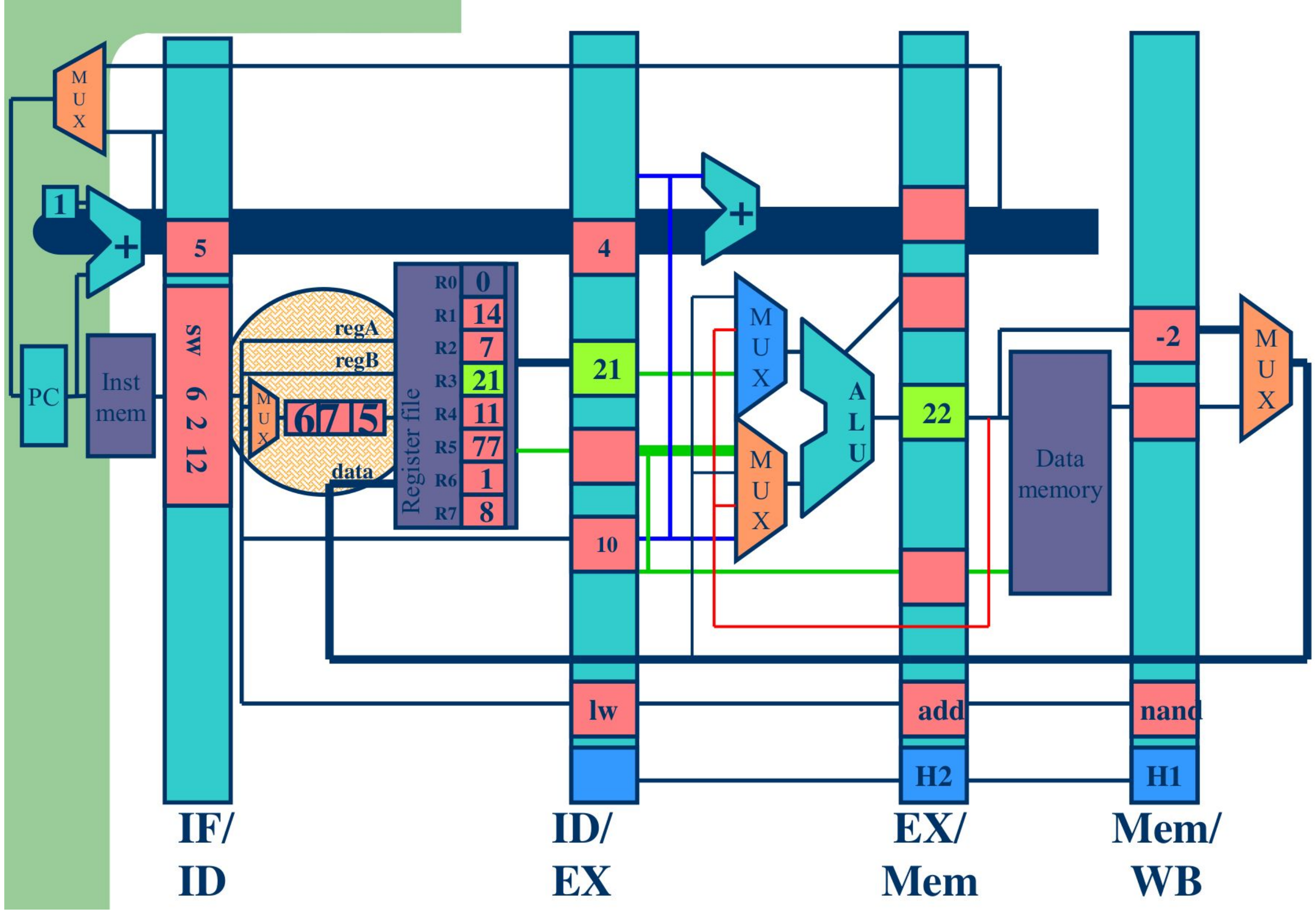
MUX
1
+
4
3
+
No Hazard
3 regA
regB
data
MUX
7 5 3
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
PC
Inst mem
lw 3 6 10
1
10
1
21
MUX
MUX
ALU
-2
Data memory
21
MUX
add
nand
add
H2
H1
IF/ID
ID/EX
EX/Mem
Mem/WB

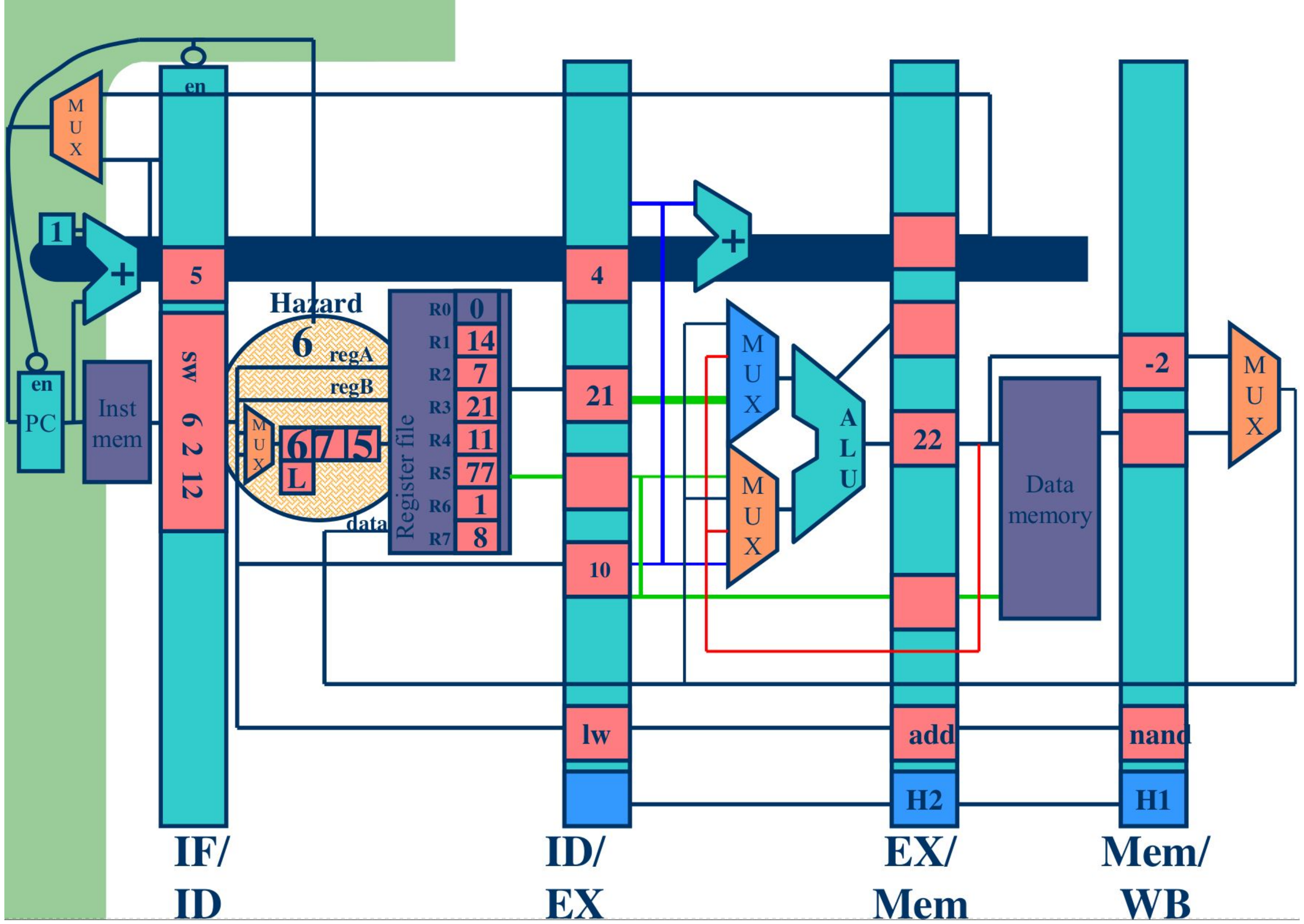
Project 3 Goals

- 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.)
  - 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

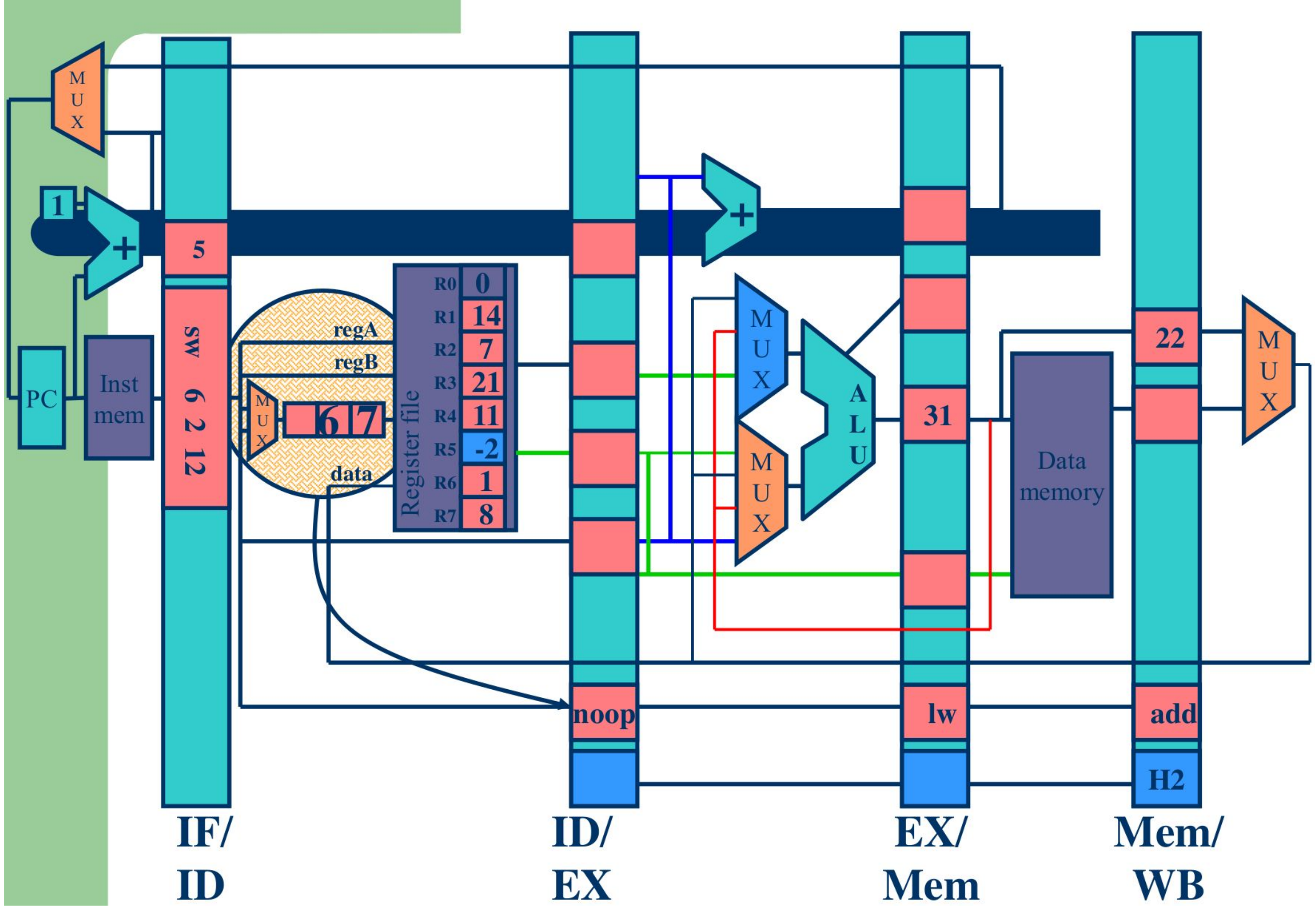
Sample Code

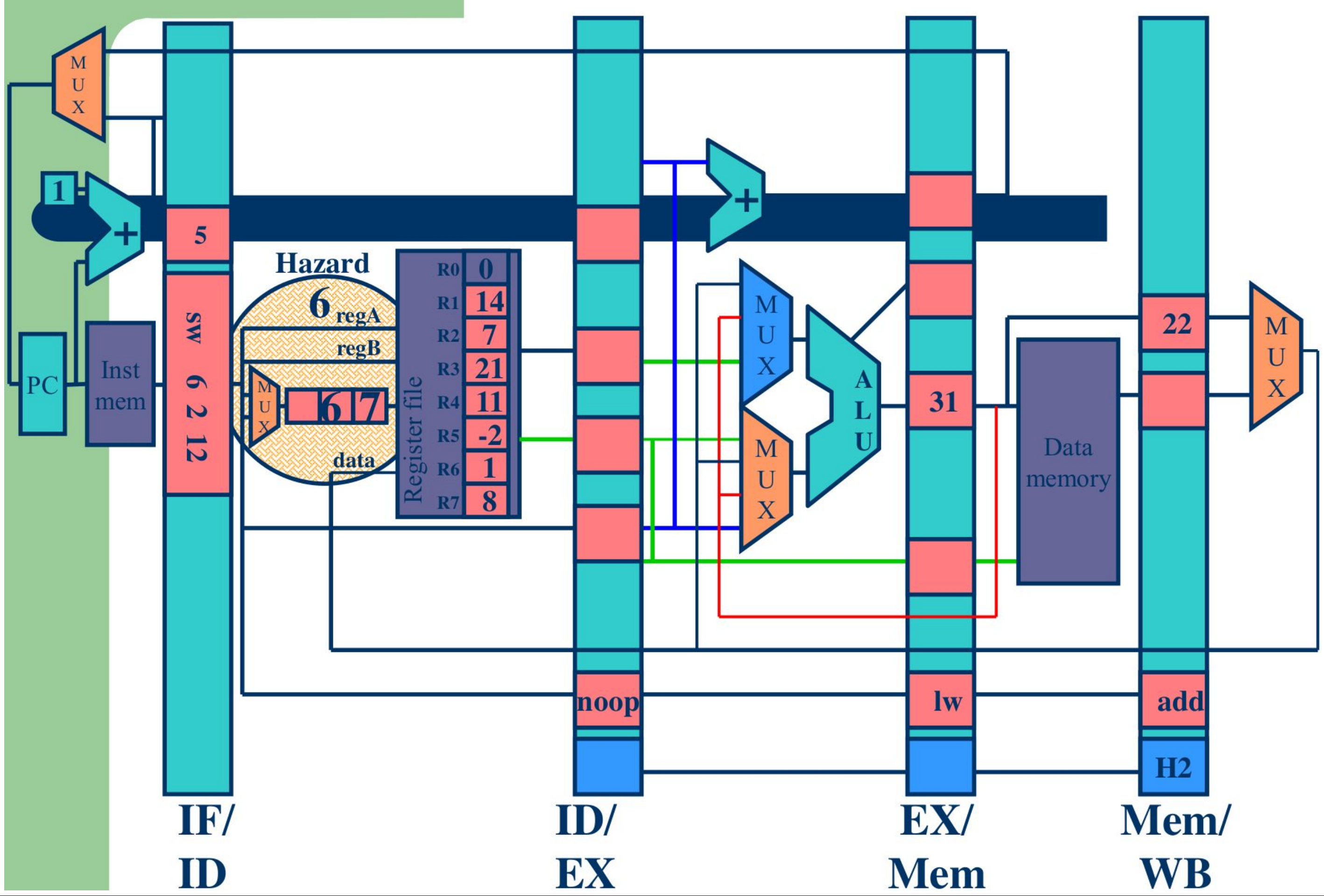
```
add    1  2  3    ;  reg 3 = reg 1 + reg 2
nand   3  4  5    ;  reg 5 = reg 3 ~& reg 4
add    6  3  7    ;  reg 7 = reg 6 + reg 3
lw     3  6  10   ;  reg 6 =  Mem[reg 3 + 10]
sw     6  2  12   ;  Mem[reg6+12] = reg 2
```

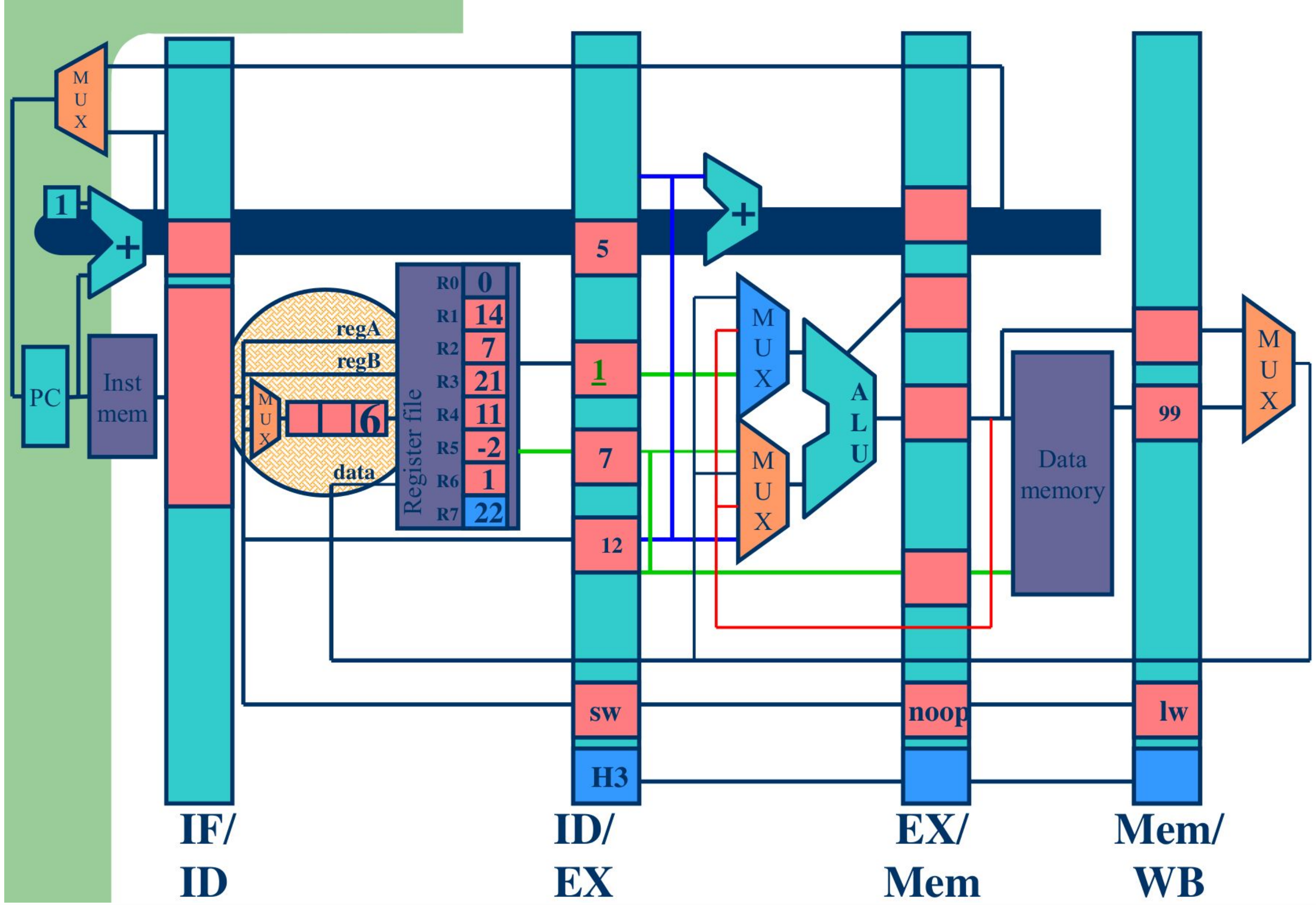

MUX
1
+
5
4
+
PC
Inst mem
sw 6 2 12
regA
regB
MUX
6 7 5
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 77
R6 1
R7 8
21
10
MUX
MUX
ALU
22
Data memory
-2
MUX
lw
add
nand
H2
H1
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB



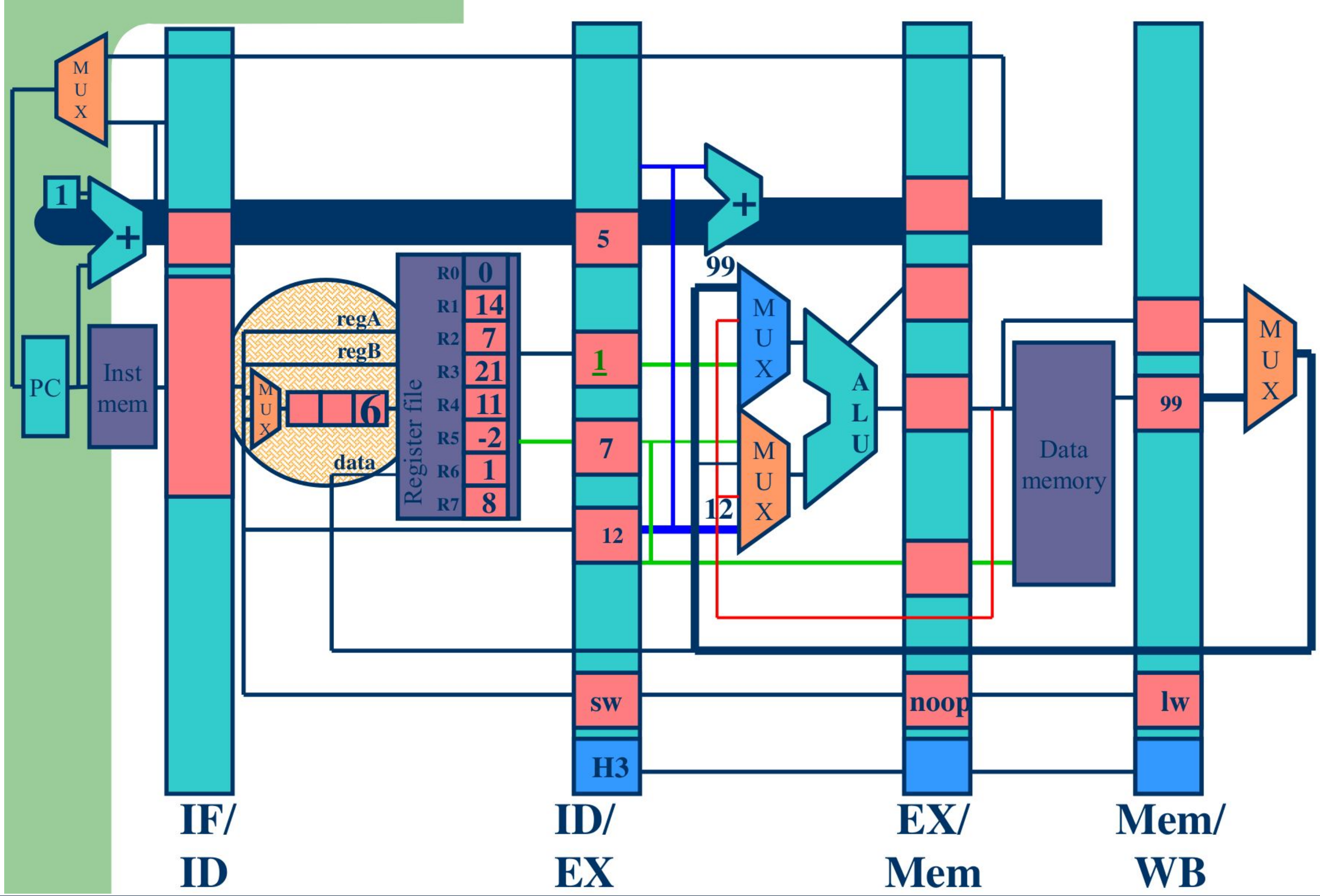
M
U
X
en
1
+
5
4
+
Hazard
6
regA
regB
R0 0
R1 14
R2 7
R3 21
R4 11
R5 77
R6 1
R7 8
Register file
M
U
X
6 7 5
L
data
en
PC
Inst mem
sw 6 2 12
21
10
M
U
X
M
U
X
A
L
U
22
Data memory
-2
M
U
X
lw
add
nand
H2
H1
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB


MUX
1
+
PC
Inst mem
5
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
noop
+
MUX
MUX
ALU
31
lw
Data memory
22
MUX
add
H2
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

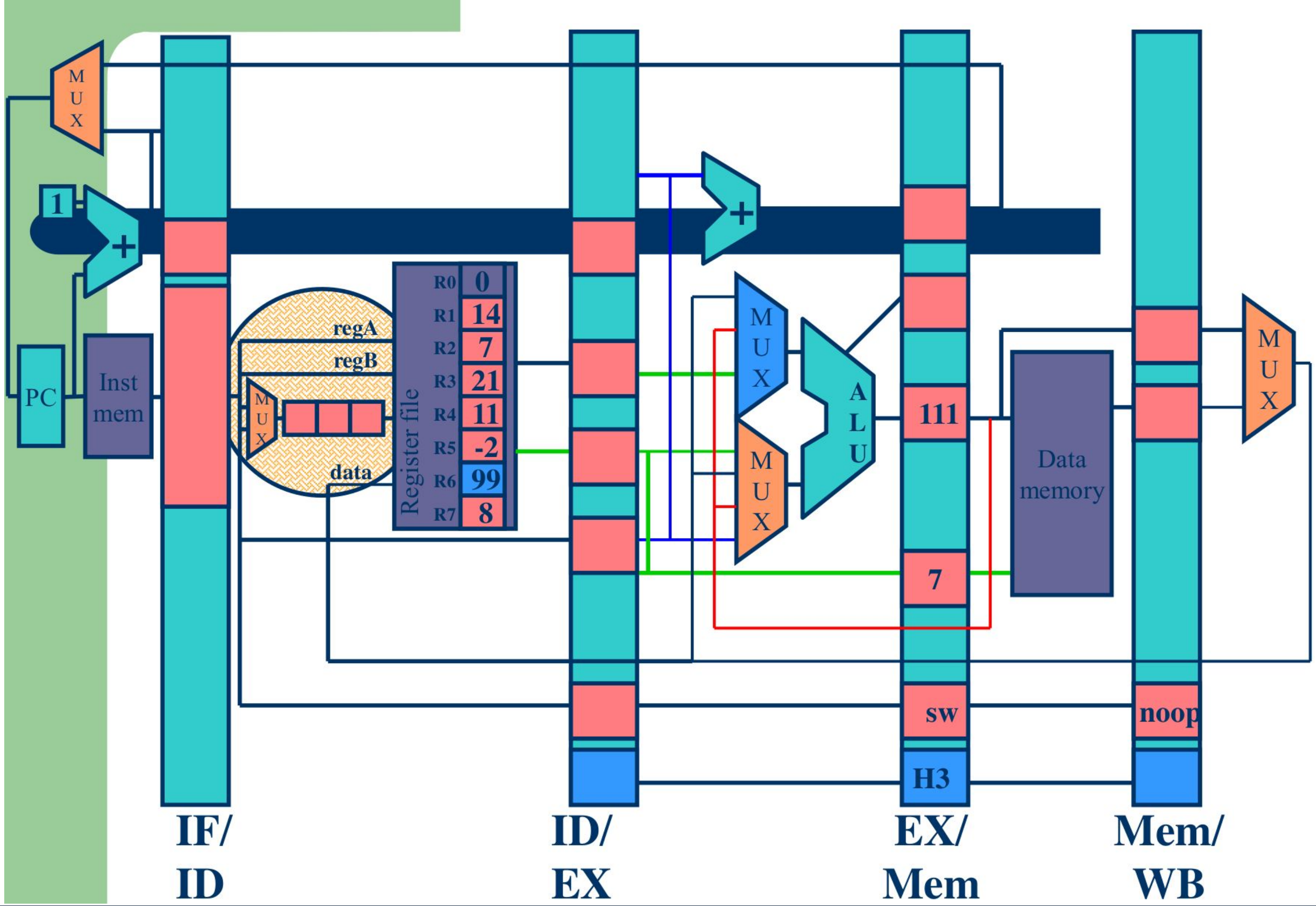

MUX
1
+
5
PC
Inst mem
sw 6 2 12
Hazard
6 regA
regB
MUX
6 7
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 1
R7 8
noop
+
MUX
MUX
ALU
31
lw
Data memory
22
MUX
add
H2
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB



M U X
1
+
PC
Inst mem
regA
regB
M U X
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
+
M U X
M U X
A L U
noop
Data memory
99
lw
M U X
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB



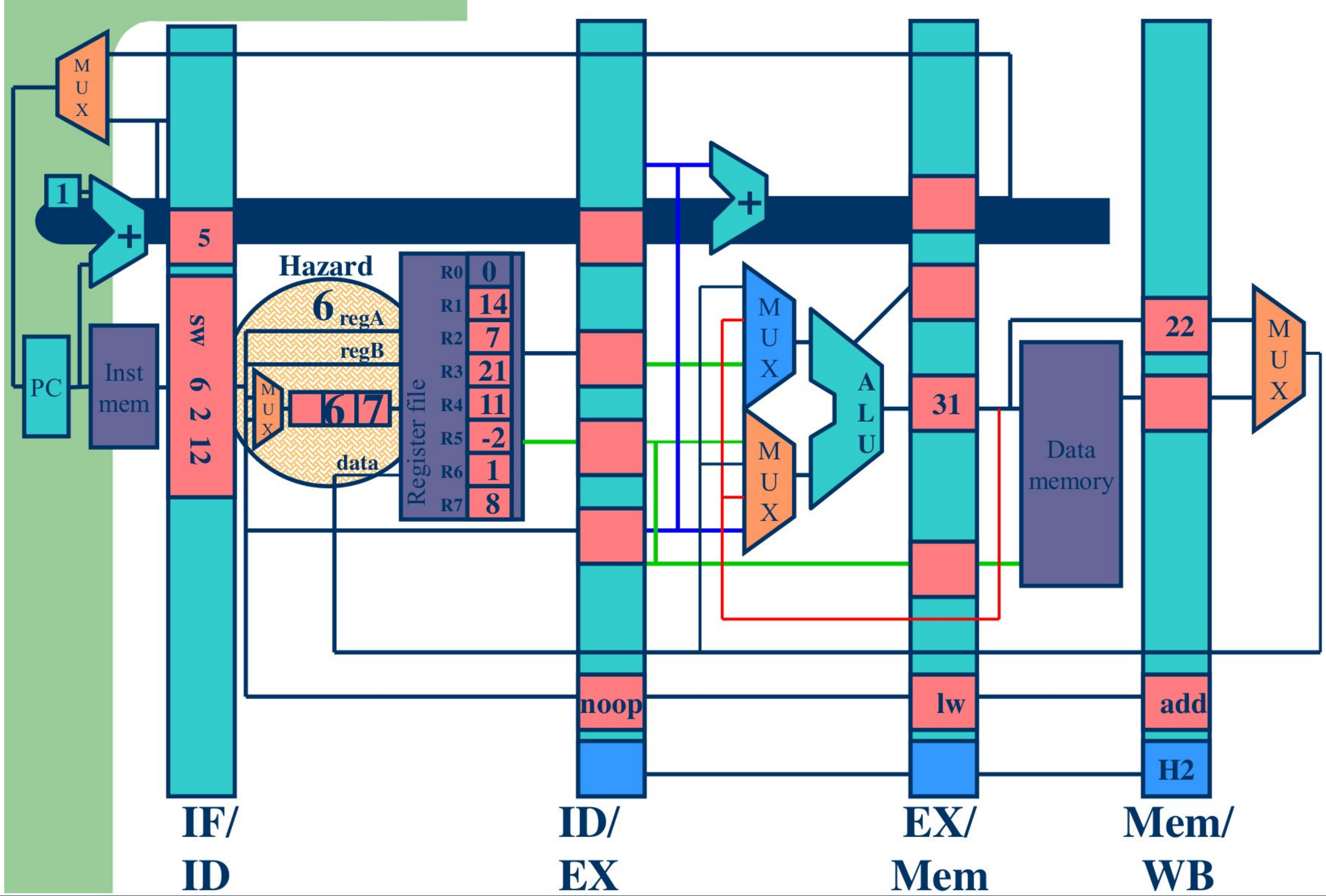
MUX
1
+
PC
Inst mem
regA
regB
MUX
6
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 1
R7 8
5
1
7
12
sw
H3
+
99
MUX
MUX
12
ALU
noop
Data memory
99
lw
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

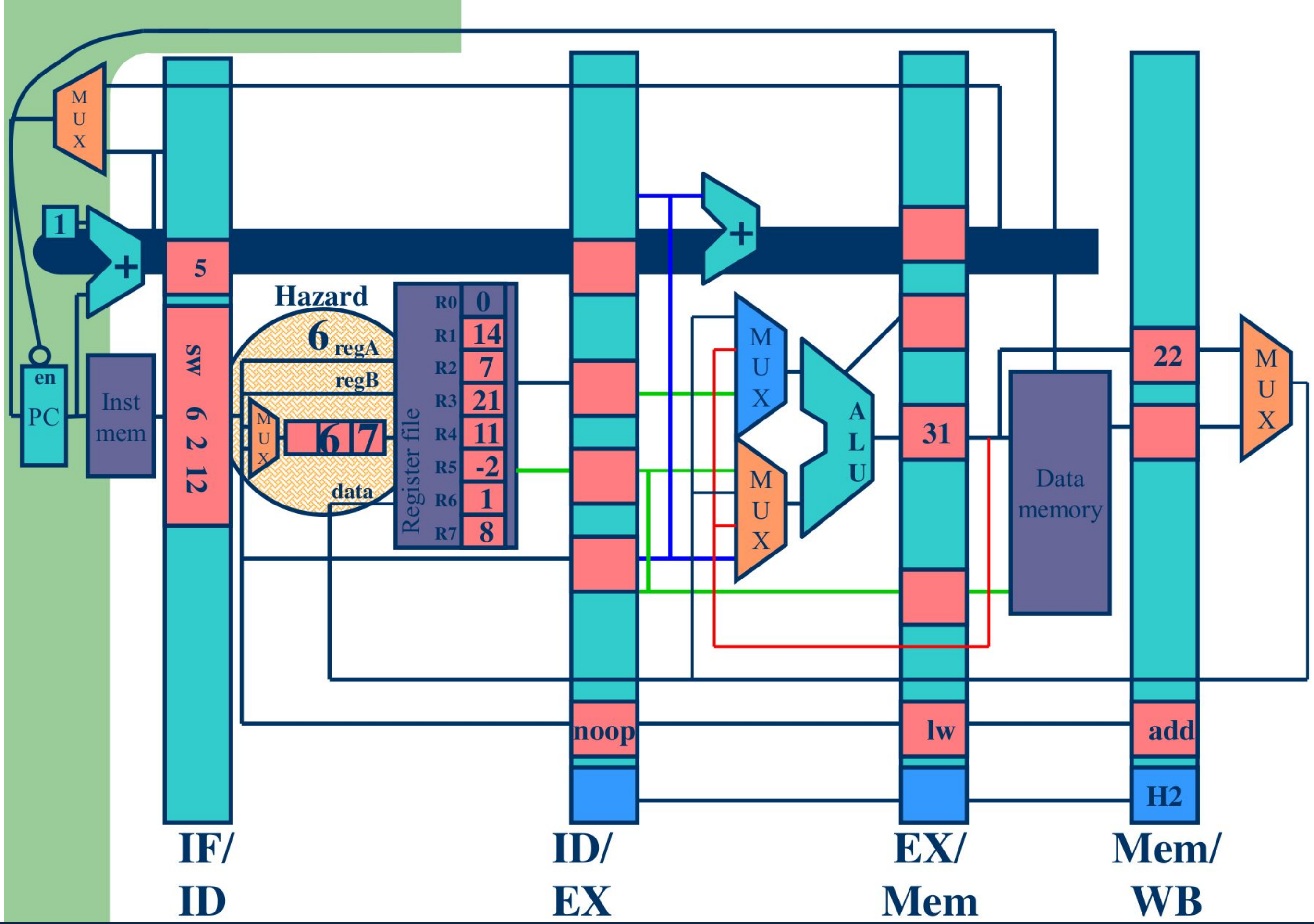


M
U
X
1
+
PC
Inst
mem
regA
regB
M
U
X
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 99
R7 8
+
M
U
X
M
U
X
A
L
U
111
7
sw
H3
Data
memory
noop
M
U
X
IF/
ID
ID/
EX
EX/
Mem
Mem/
WB

Project 3 Goals

- 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.)
    - 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
- 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


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



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
noop
+
MUX
MUX
ALU
31
lw
Data memory
22
add
H2
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

Common Problems

- Error on load access
  - Tried to access an invalid memory address
- Be careful of using “op{a,b}_select” signals
  - They are mux select signals for ALU inputs, not indications of whether instruction uses rs1 and/or rs2

Implementation Tip$

- 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

Debugging Project 3

- Examine `<my_program>.ppln` output
- Find first incorrect register write/memory load
- Trace back execution of that instruction
- Don't start with Verdi, instead try the Visual debugger!
    - `make <my_program>.vis`
    - Verdi is useful only once you know precisely where/when a bug is occurring
    - But is still a good tool for synthesis and tracing X values
- Avoid "staring at your screen" debugging