Pipeline: Exceptions

Dr. Tao Xie

Fall, 2017

These slides are adapted from notes by Dr. David Patterson (UCB) and Dr. Xiao Qin (Auburn)

Exceptions - “Stuff Happens”

- Exceptions definition: “*unexpected change in control flow*”
- Another form of control hazard.

For example:

```
add $1, $2, $1;     causing an arithmetic overflow
sw   $3, 400($1);
add $5,  $1, $2;
```

Invalid $1 contaminates other registers or memory locations!

Two Types of Exceptions: Interrupts and Traps

- Interrupts
  - Caused by external events:
    - Network, Keyboard, Disk I/O, Timer
    - Page fault - virtual memory
    - System call - user request for OS action
  - Asynchronous to program execution
  - May be handled between instructions
  - Simply suspend and resume user program
- Traps
  - Caused by internal events
    - Exceptional conditions (overflow)
    - Undefined Instruction
    - Hardware malfunction
  - Usually Synchronous to program execution
  - Condition must be remedied by the handler
  - Instruction may be retried or simulated and program continued or program may be aborted

Interrupts

- Interrupts are external events that require the processor's attention.
  - Peripherals and other I/O devices may need attention.
  - Timer interrupts to mark the passage of time.
- These situations are not errors.
  - They happen normally.
  - All interrupts are recoverable:
    - The interrupted program will need to be resumed after the interrupt is handled.
- It is the operating system's responsibility to do the right thing, such as:
  - Save the current state.
  - Find and load the correct data from the hard disk
  - Transfer data to/from the I/O device.

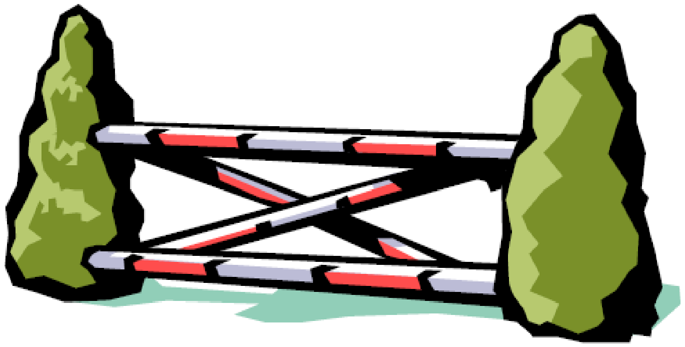
Exception handling

- Exceptions are typically errors that are detected within the processor.
  - The CPU tries to execute an illegal instruction opcode.
  - An arithmetic instruction overflows, or attempts to divide by 0.
  - The a load or store cannot complete because it is accessing a virtual address currently on disk

- There are two possible ways of resolving these errors.
  - If the error is un-recoverable, the operating system kills the program.
  - Less serious problems can often be fixed by the O/S or the program itself.

How interrupts/exceptions are handled

- For simplicity exceptions and interrupts are handled the same way.
- When an exception/interrupt occurs, we stop execution and transfer control to the operating system, which executes an “exception handler” to decide how it should be processed.
- The exception handler needs to know two things.
  - The cause of the exception (e.g., overflow or illegal opcode).
  - What instruction was executing when the exception occurred. This helps the operating system report the error or resume the program.
- This is another example of interaction between software and hardware, as the cause and current instruction must be supplied to the operating system by the processor.



Synchronous vs Asynchronous

- *Definition*: If the event occurs at the same place every time the program is executed with the same data and memory allocation, the event is synchronous. Otherwise asynchronous.
- Except for hardware malfunctions, asynchronous events are caused by devices external to the CPU and memory.

•Asynchronous events usually are easier to handled because asynchronous events can be handled after the completion of the current instruction.

Exceptions in Simple five-stage pipeline

- Instruction Fetch, & Memory stages
  - Page fault on instruction/data fetch
  - Misaligned memory access
  - Memory-protection violation
- Instruction Decode stage
  - Undefined/illegal opcode
- Execution stage
  - Arithmetic exception
- Write-Back stage
  - *No exceptions!*

What happens during an exception?

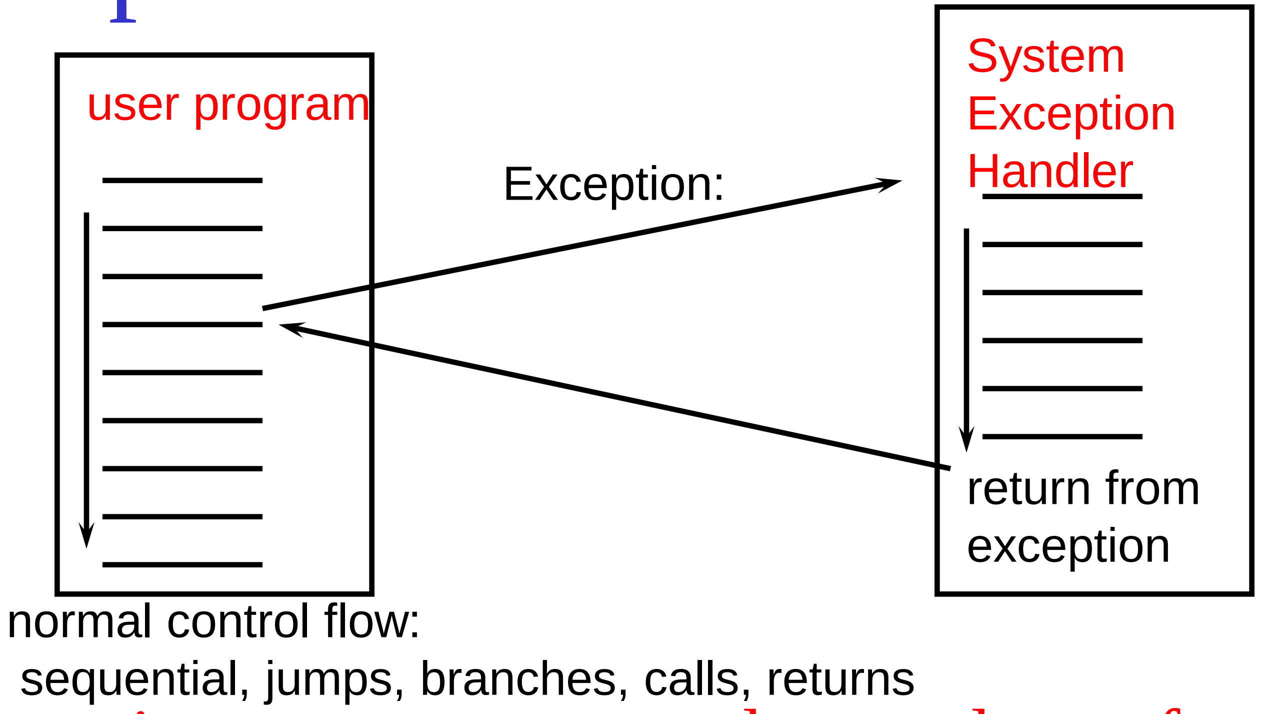
The Hardware Part

- The pipeline has to
  1) stop executing the offending instruction in midstream,
  2) let all prior instructions complete,
  3) flush all following instructions,
  4) set a register to show the cause of the exception,
  5) save the address of the offending instruction, and
  6) then jump to a prearranged address (the address of the exception handler code)

The Software Part

- The software (OS) looks at the cause of the exception and “deals” with it
- Normally OS kills the program

Exceptions



Exception = unprogrammed control transfer

– system takes action to handle the exception

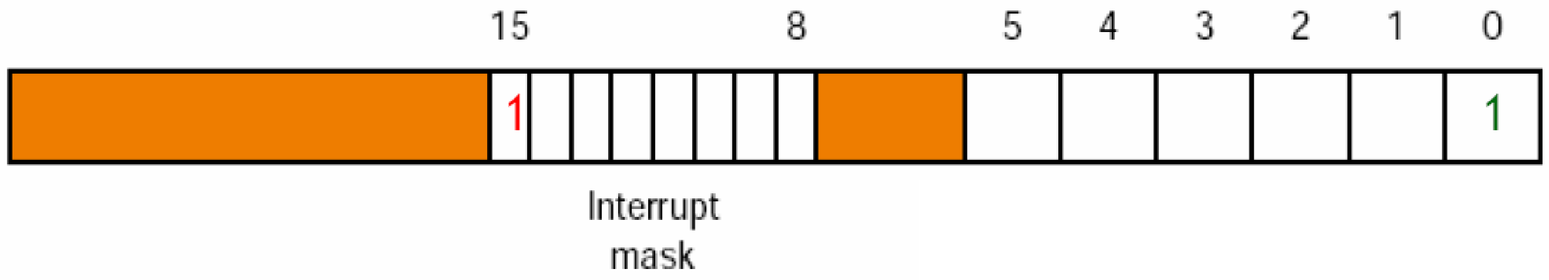
- must record the address of the offending instruction
- record any other information necessary to return afterwards

– returns control to user

– must save & restore user state

MIPS Interrupt Programming

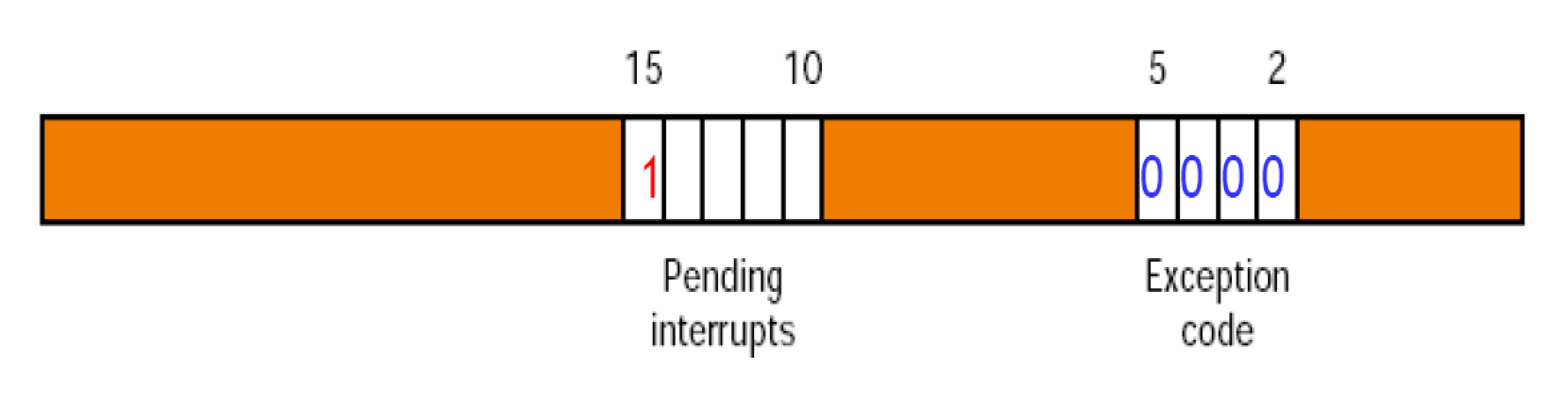
- In order to receive interrupts, the software has to enable them.
  - On a MIPS processor, this is done by writing to the Status register.
    - Interrupts are enabled by setting bit zero.



- MIPS has multiple interrupt levels
  - Interrupts for different levels can be selectively enabled.
  - To receive an interrupt, it's bit in the interrupt mask (bits 8-15 of the Status register) must be set.
    - In the Figure, interrupt level 15 is enabled.

MIPS Interrupt Programming

- When an interrupt occurs, the Cause register indicates which one.
  - For an exception, the exception code field holds the exception type.
  - For an interrupt, the exception code field is 0000 and bits will be set for pending interrupts.
    - The register below shows a pending interrupt at level 15



- The exception handler is generally part of the operating system.

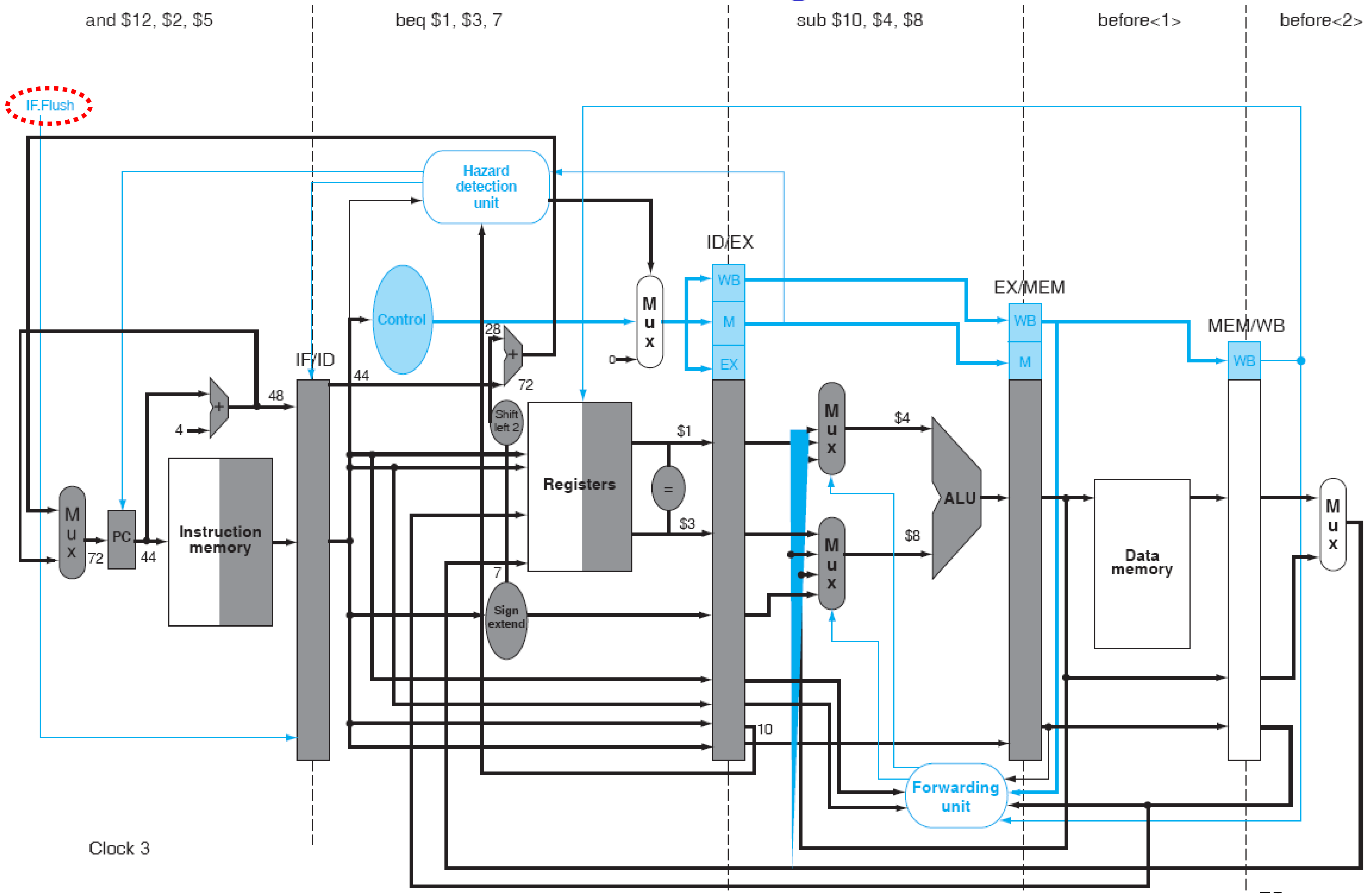
Additions to MIPS ISA to support Exceptions

- EPC (Exceptional Program Counter)
  - A 32-bit register
  - Hold the address of the offending instruction
- Cause
  - A 32-bit register in MIPS (some bits are unused currently.)
  - Record the cause of the exception
- Status - interrupt mask and enable bits and determines what exceptions can occur.
- Control signals to write EPC , Cause, and Status
- Be able to write exception address into PC, increase mux set PC to exception address (MIPS uses 8000 00180_{hex}).
- May have to undo PC = PC + 4, since want EPC to point to offending instruction (not its successor); PC = PC – 4
- What else? **flush all following instructions**

Flush instructions in Branch Hazard

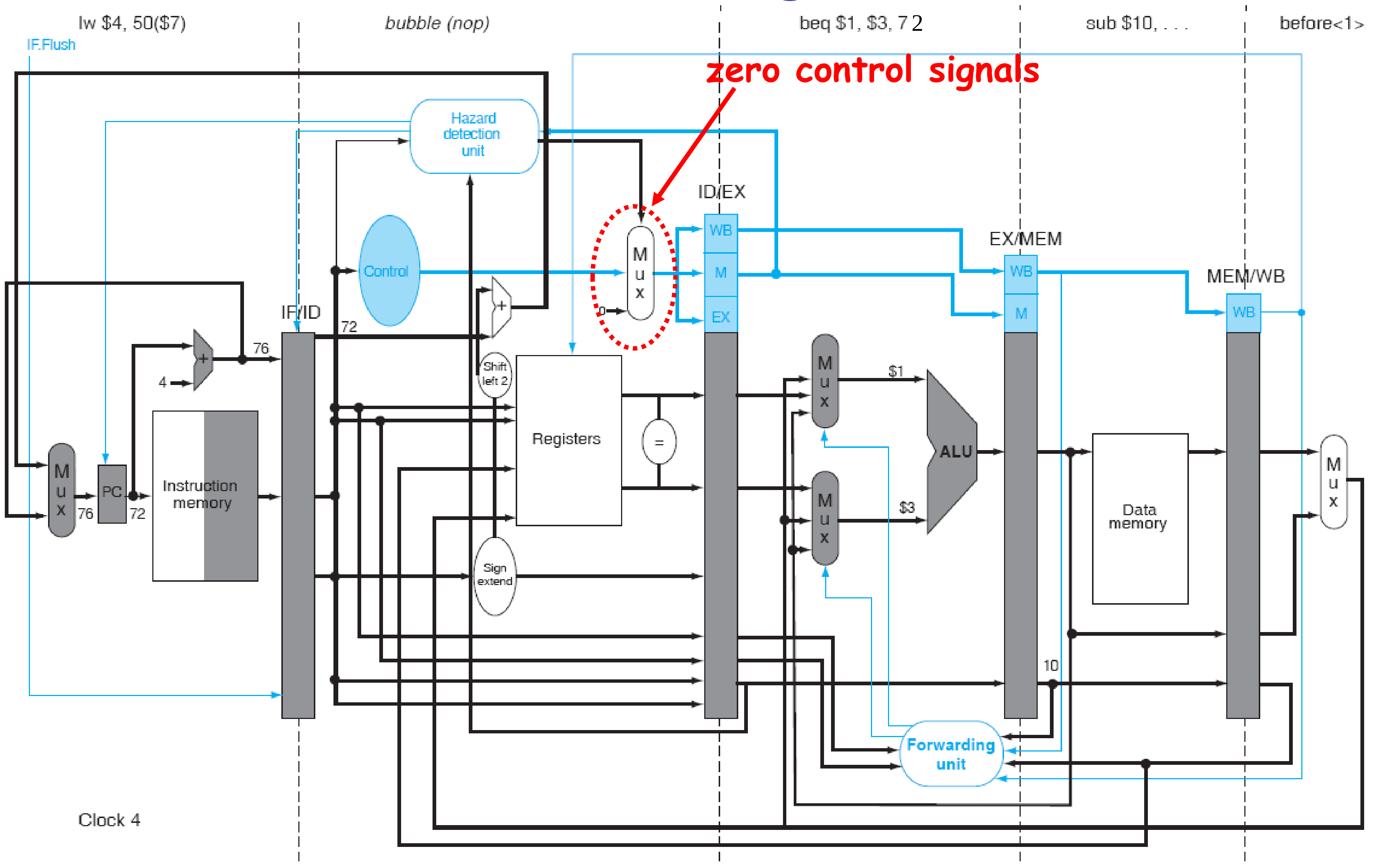
```
36 sub    $10,  $4,   $8
40 beq    $1,   $3,   7   # taget = 40 + 4 + 7*4 = 72
44 and    $12,  $2,   $5
48 or     $13,  $2,   $6
52 ….
….
72 lw     $4,   50($7)
```

Flush instructions at IF stage in Branch Hazard



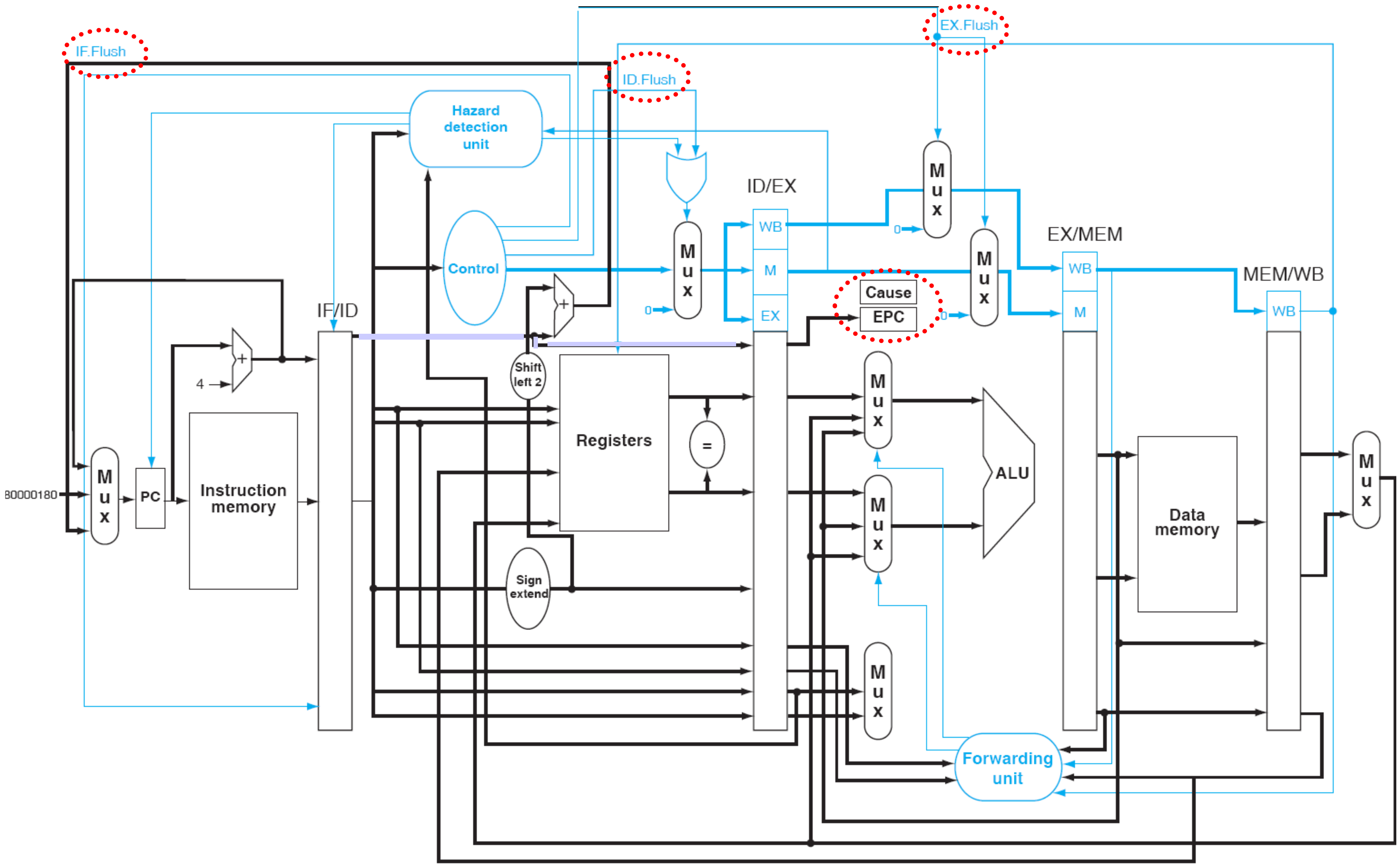
Turn the instructions at IF stage into *nop*.

Flush instructions at IF stage in Branch Hazard



Turn the instructions at IF stage into *nop*.

Additions to MIPS ISA to support Exceptions



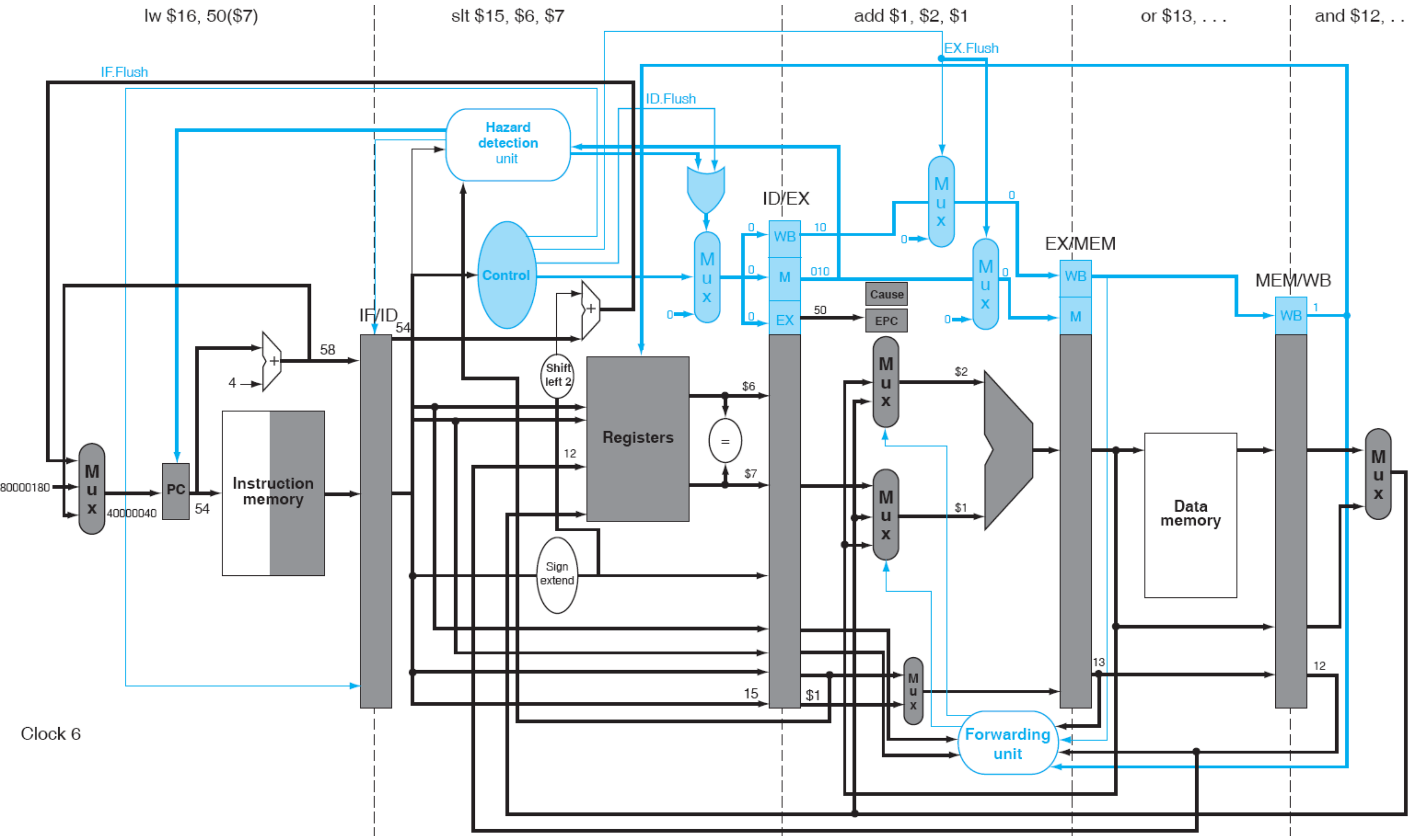
Exceptions Example

```
40hex  sub  $11, $2,  $4
44hex  and  $12, $2,  $5
48hex  or   $13, $2,  $6
4Chex  add  $1,  $2,  $1; // arithmetic overflow
50hex  slt  $15, $6,  $7
54hex  lw   $16, 50($7)
```

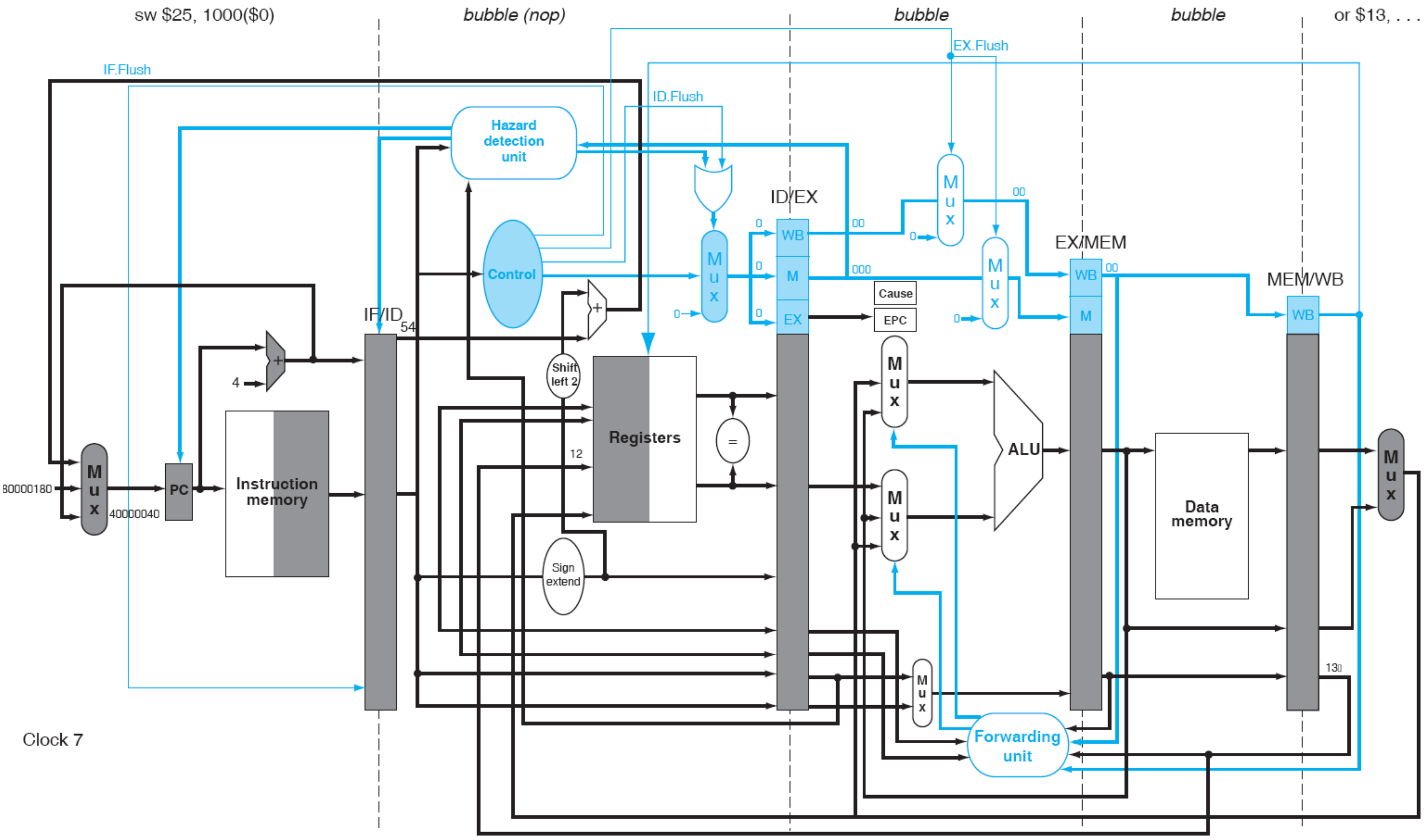
Exception handling program:

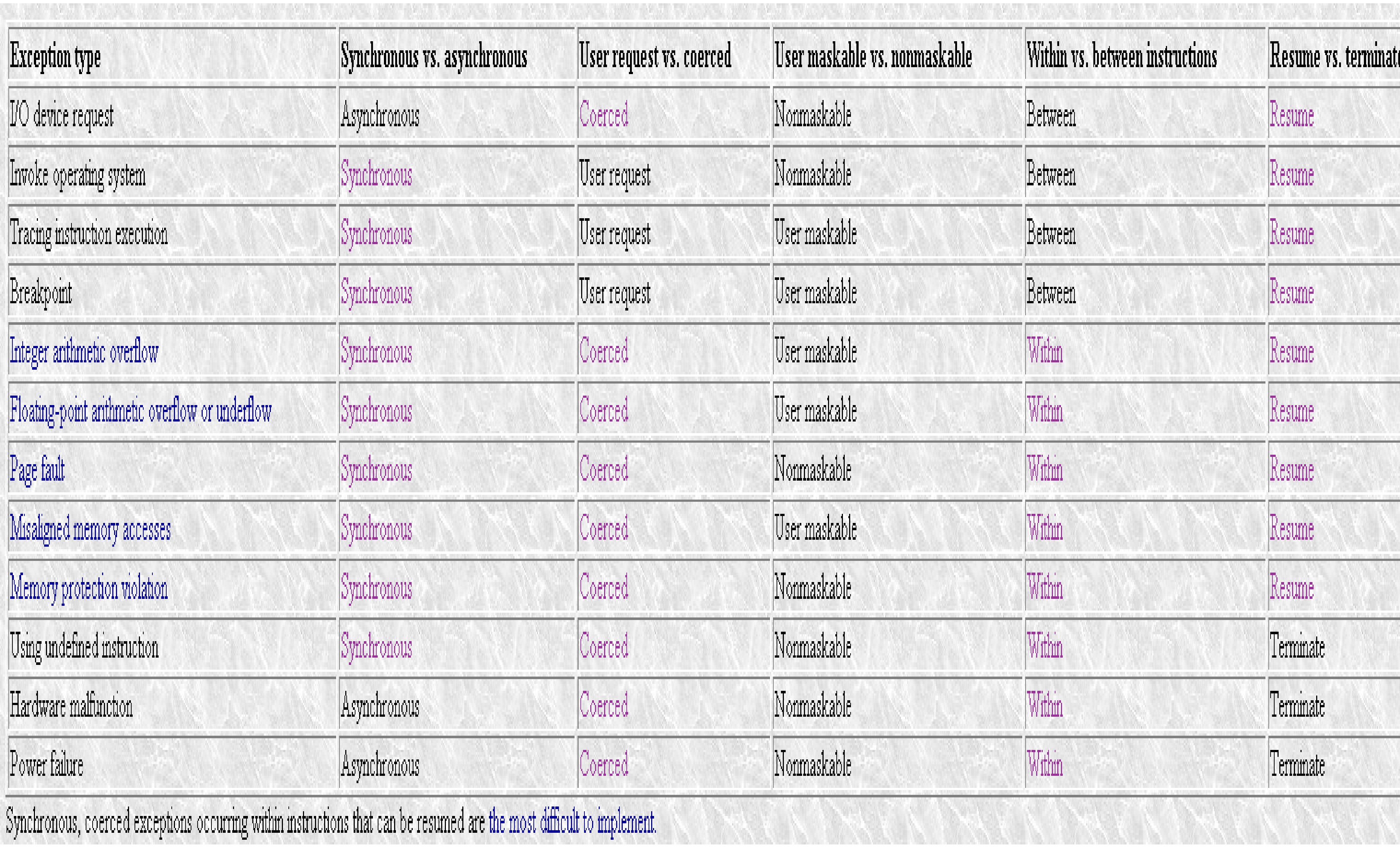
```
40000040hex   sw   $25,   1000($0)
40000044hex   sw   $12,   1000($0)
```

Exceptions Example



Exceptions Example





Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminat
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violation	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instruction	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunction	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Synchronous, coerced exceptions occurring within instructions that can be resumed are the most difficult to implement.

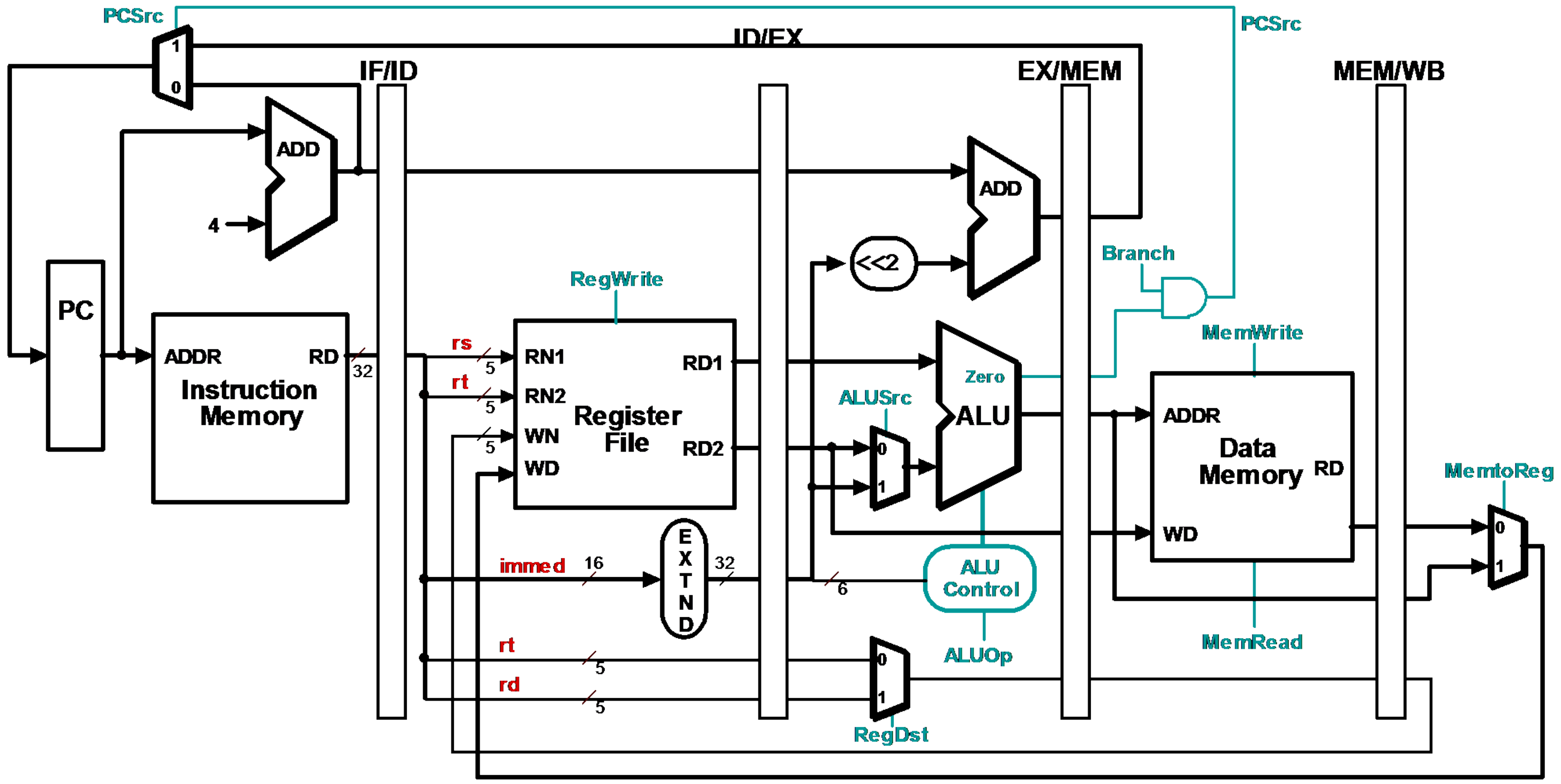
Summary

- Exceptions
  - Interrupts
  - Traps
- Exceptions in five-stage pipeline
- Exception detection (not covered)
- Exception handling
  - Stop the offending instruction
  - Flush instructions following the offending instructions
  - Save the address of the offending instruction, and
  - Jump to a prearranged exception handler code

Pipelining in MIPS

- MIPS architecture was designed to be pipelined
  - Simple instruction format (makes IF, ID easy)
    - Single-word instructions
    - Small number of instruction formats
    - Common fields in same place (e.g., rs, rt) in different formats
  - Memory operations only in lw, sw instructions (simplifies EX)
  - Memory operands aligned in memory (simplifies MEM)
  - Single value for writeback (limits forwarding)
- Pipelining is harder in CISC architectures

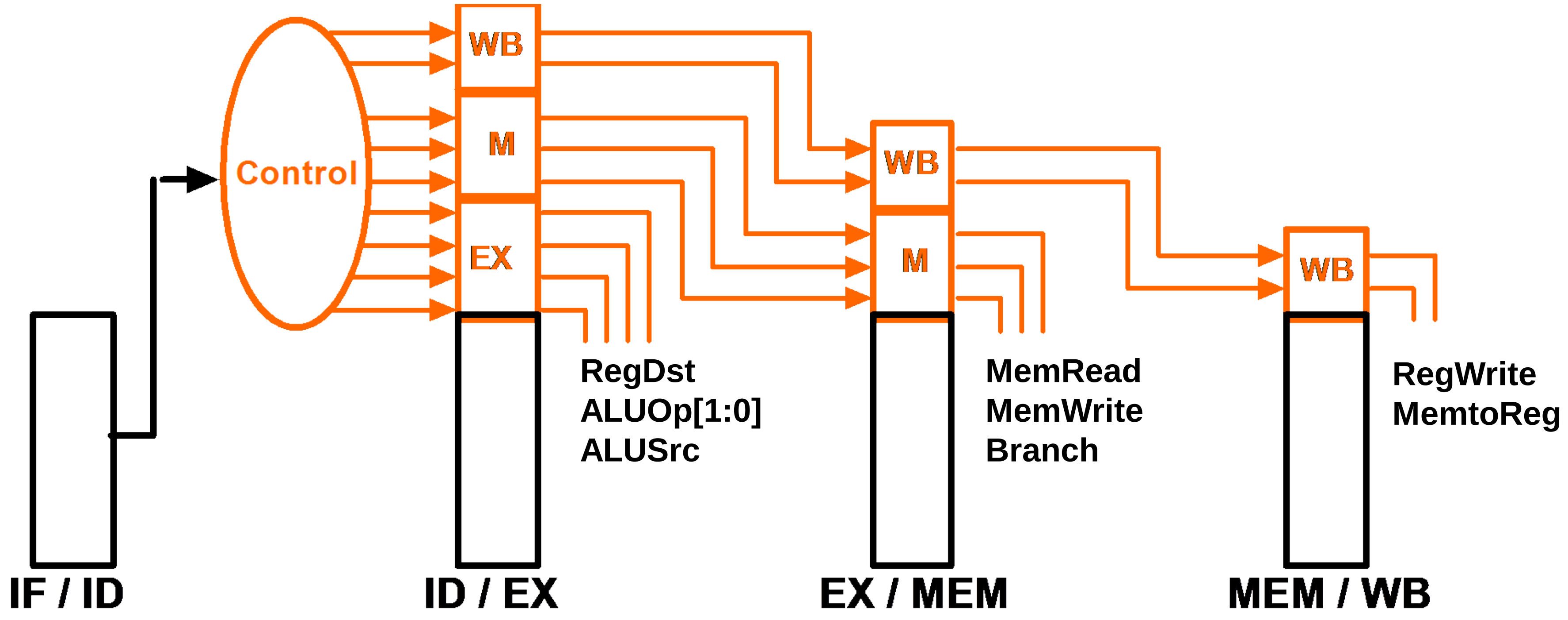
Pipelined Datapath with Control Signals



Next Step: Adding Control

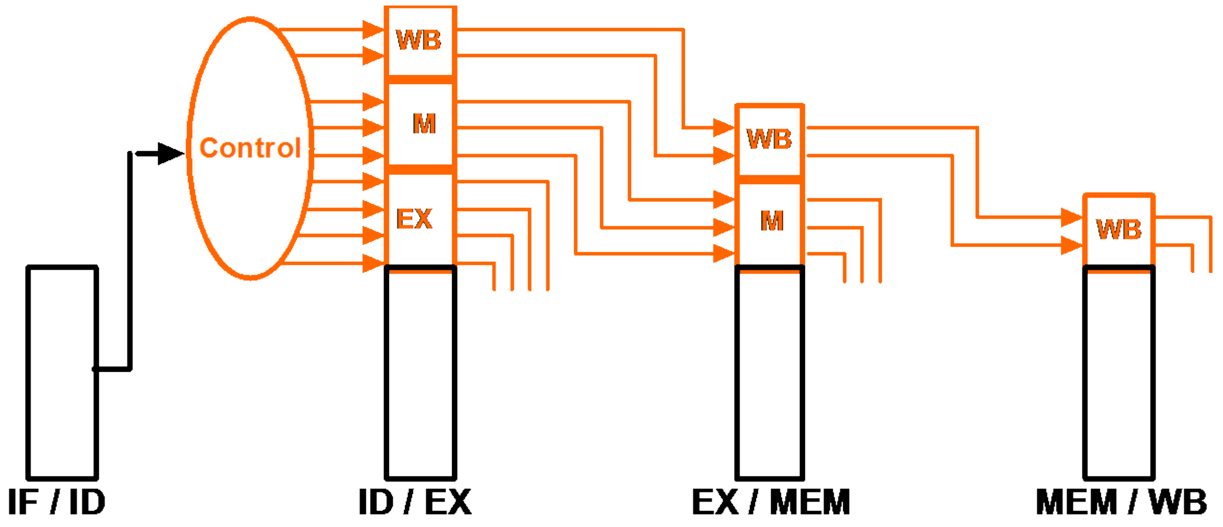
- Basic approach: build on single-cycle control
  - Place control unit in ID stage
  - Pass control signals to following stages
- Later: extra features to deal with:
  - Data forwarding
  - Stalls
  - Exceptions

Control for Pipelined Datapath

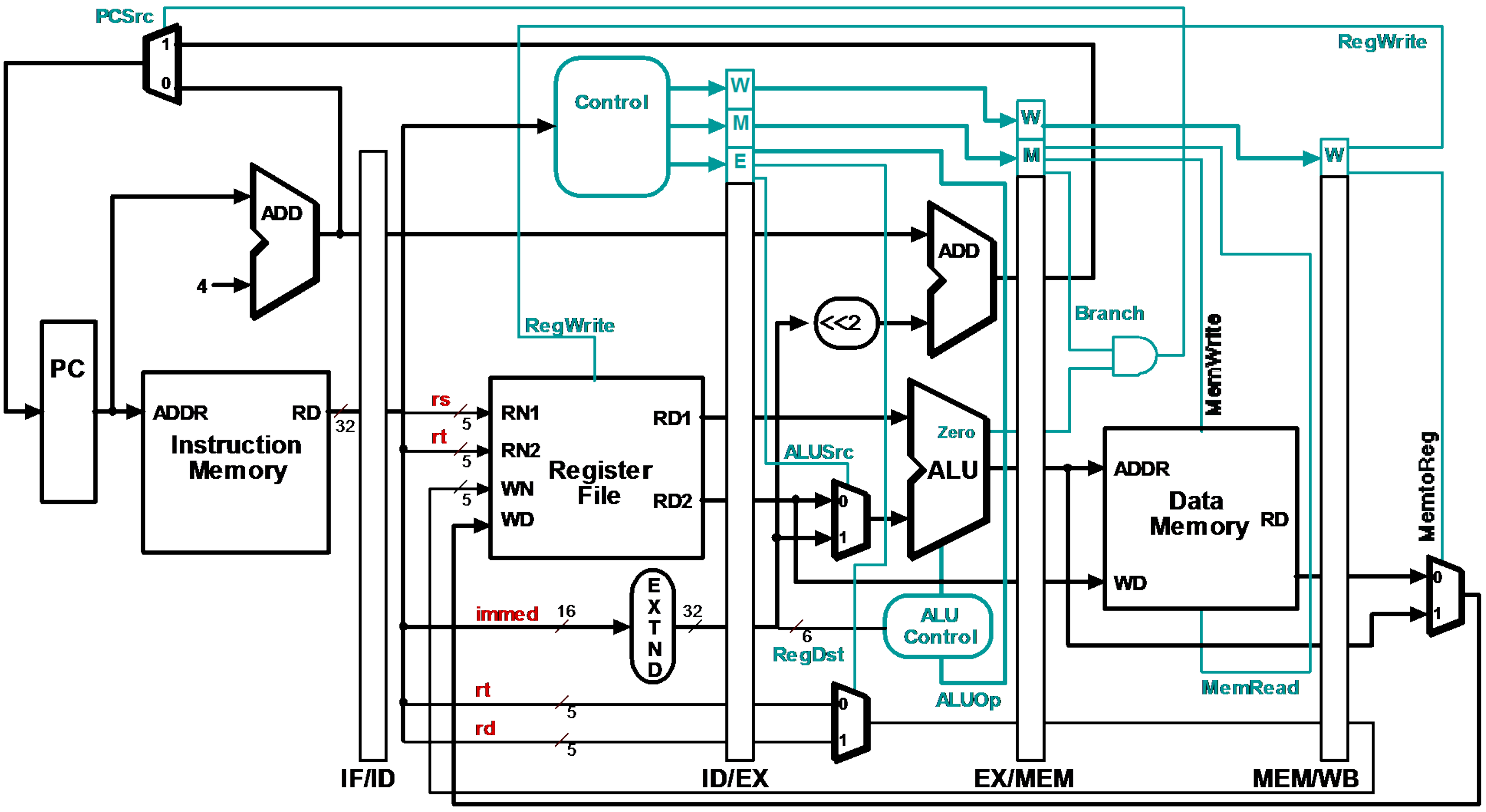


Control for Pipelined Datapath

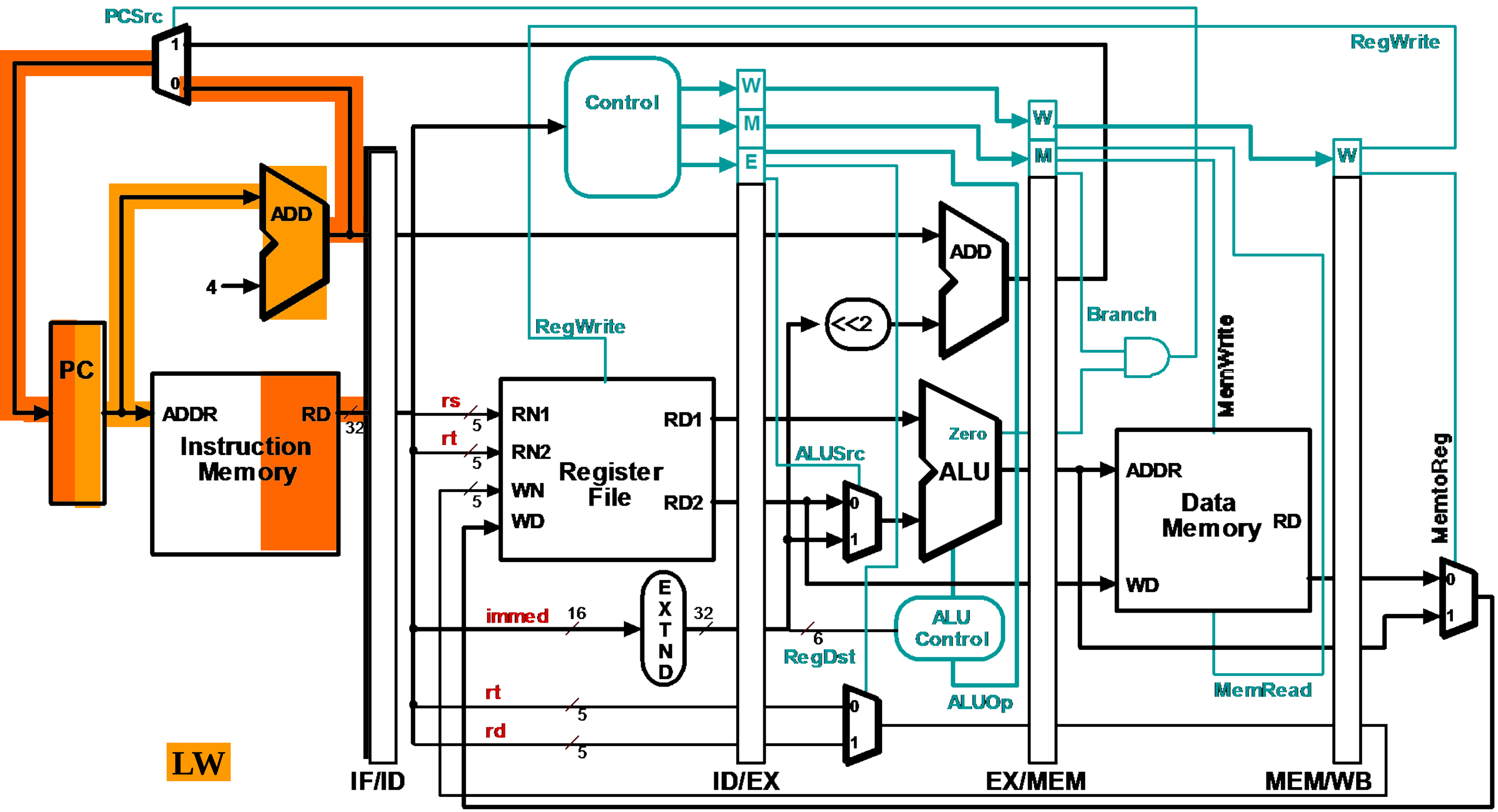
Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X



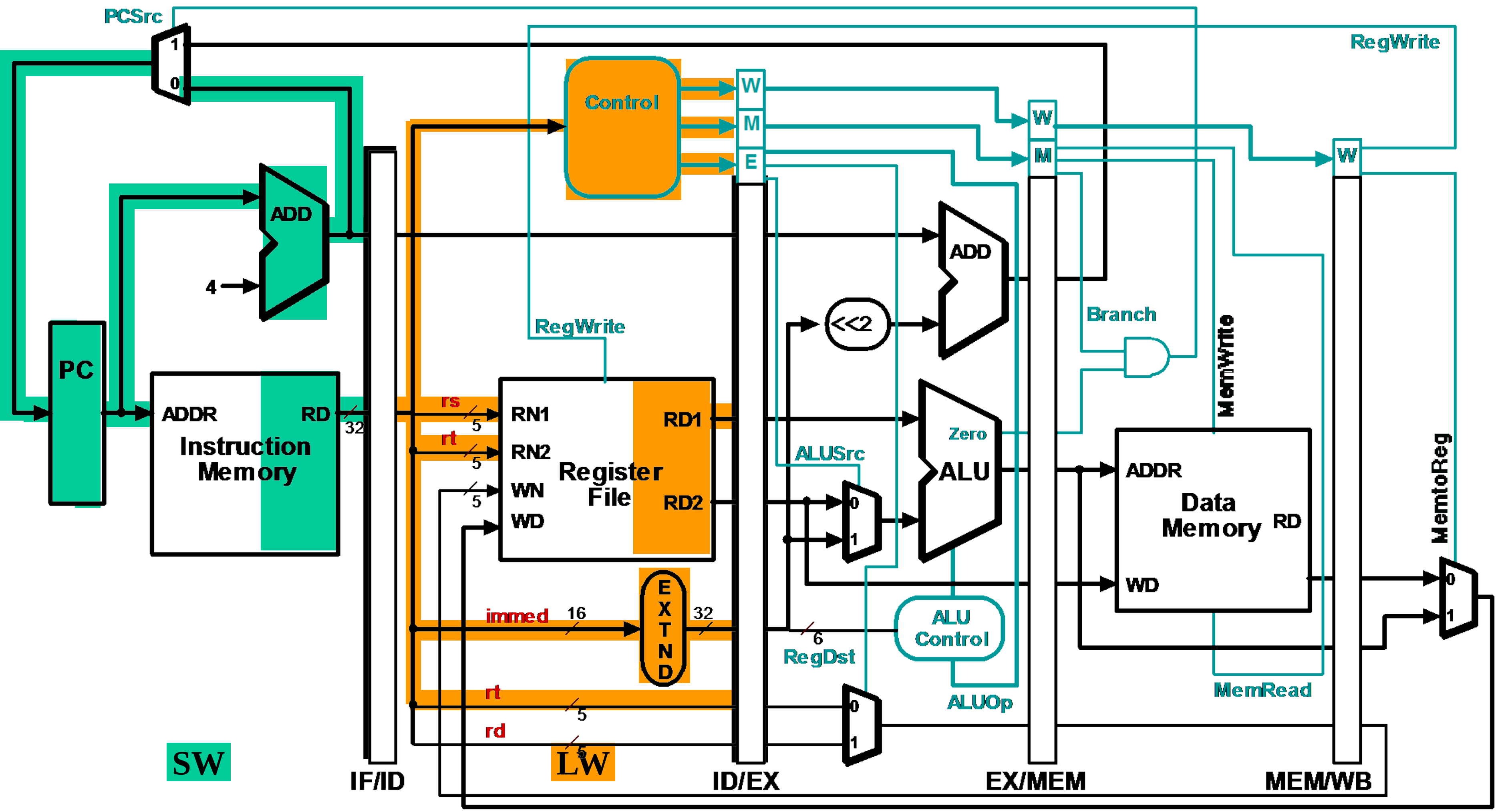
Datapath and Control Unit



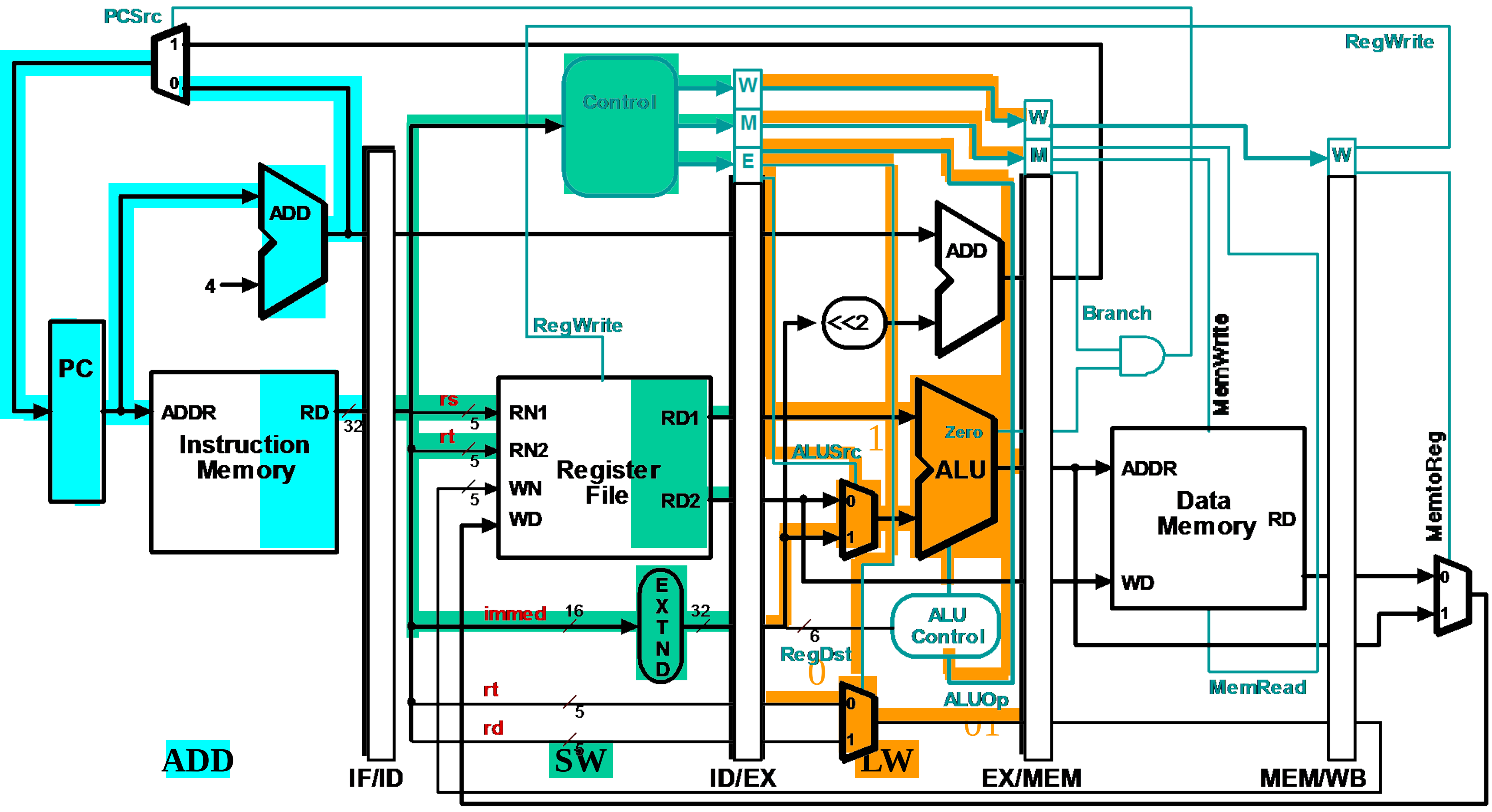
Tracking Control Signals - Cycle 1



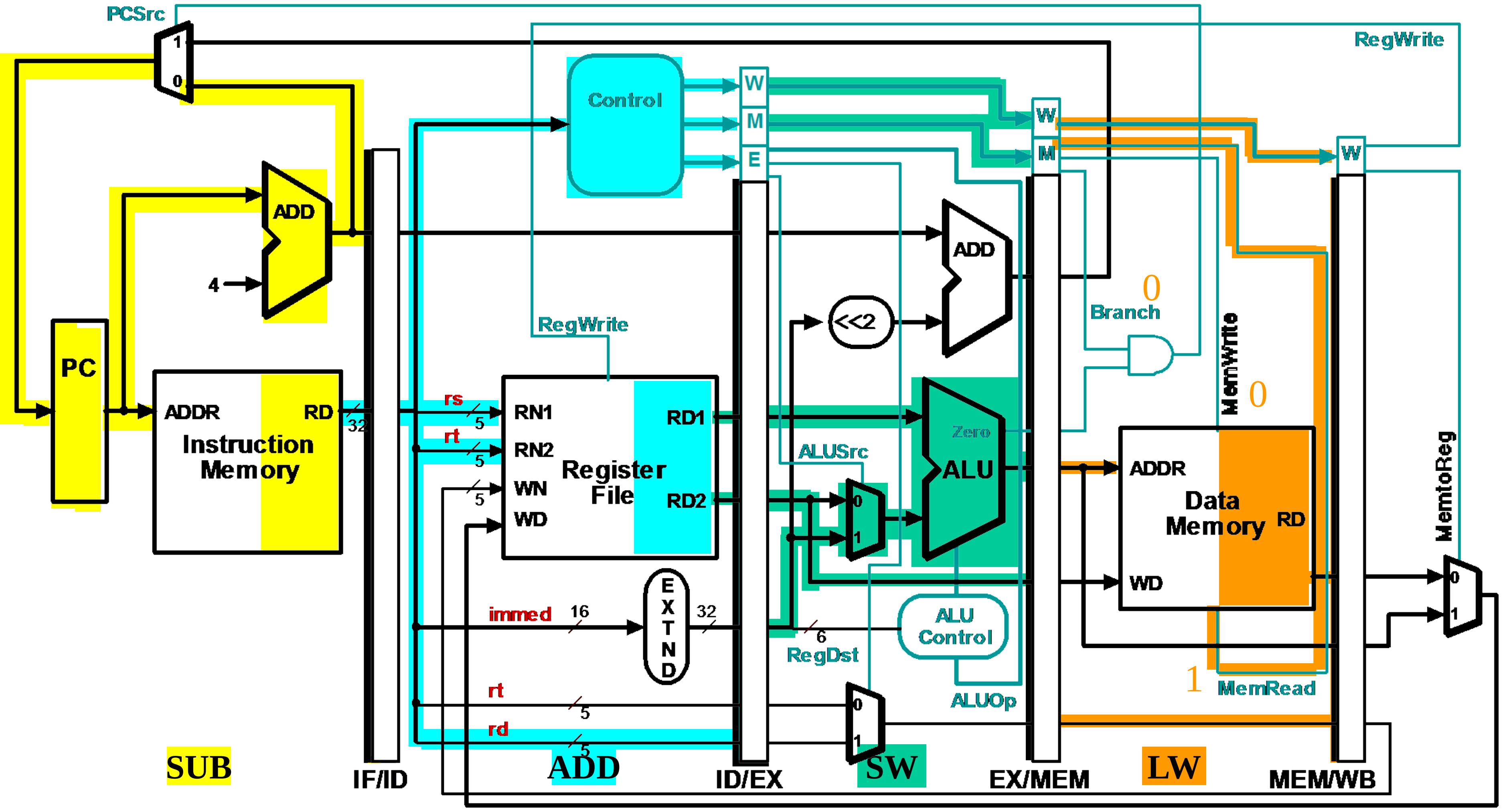
Tracking Control Signals - Cycle 2



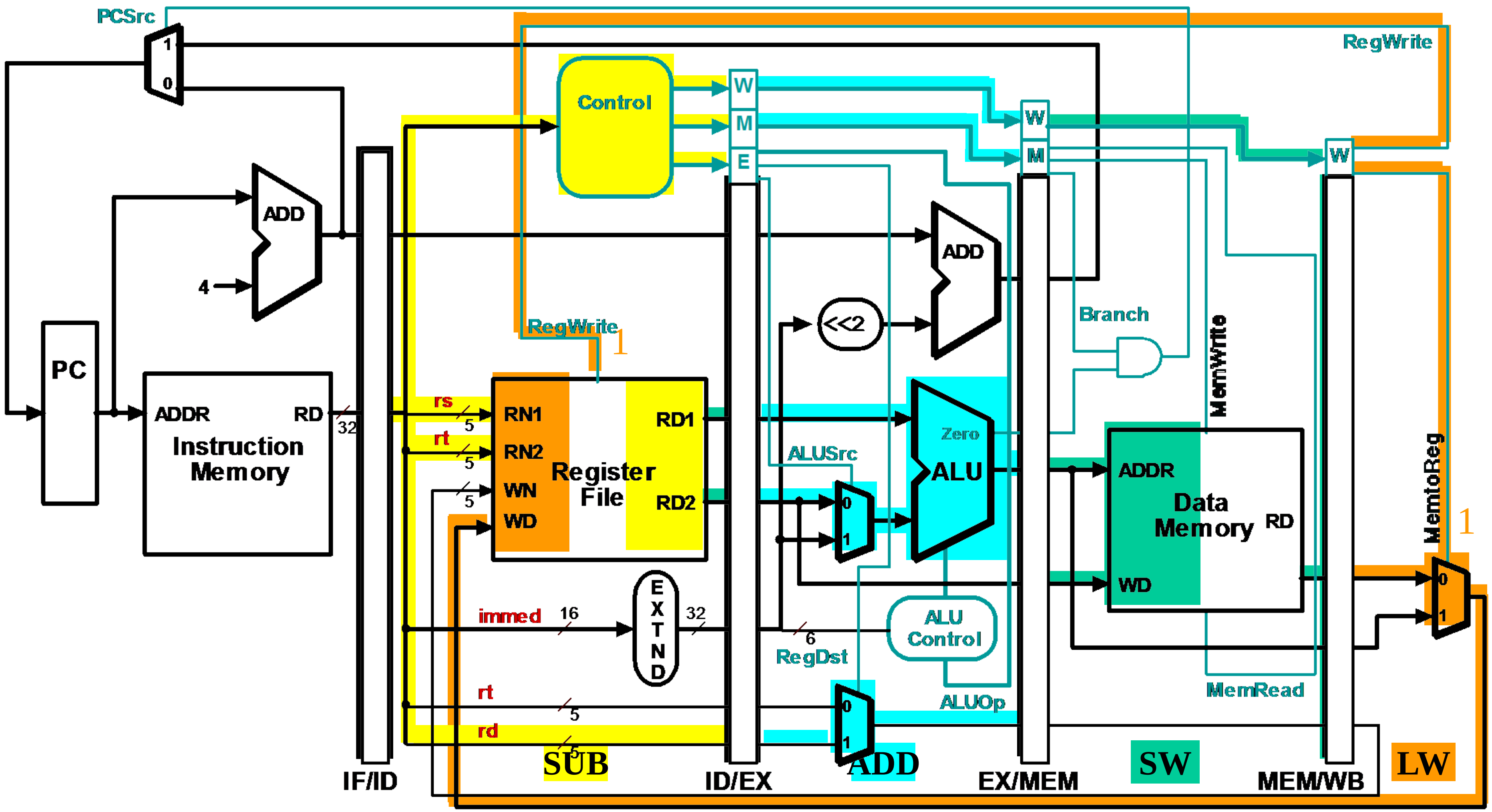
Tracking Control Signals - Cycle 3



Tracking Control Signals - Cycle 4



Tracking Control Signals - Cycle 5



Class Exercise

Consider the following code segment

```
1.      LW R1, 0(R4)
2.      LW R2, 0(R5)
3.      ADD R3, R1, R2
4.      BNZ R3, L
5.      LW  R4, 100(R1)
6.      LW R5, 100(R2)
7.      SUB R3, R4, R5
8.   L: SW R3, 50(R1)
```

Assuming that

- **there is no forwarding,**
- **zero testing is being resolved during ID, and**
- **registers can be written in the first of the WB cycle and also be read in the send half of the same WB cycle,**

Question: identify the resources of various hazards in the above code sequence.

Class Exercise

Consider the following code segment

```
1.     LW R1, 0(R4)
2.     LW R2, 0(R5)
3.     ADD R3, R1, R2
4.     BNZ R3, L
5.     LW  R4, 100(R1)
6.     LW R5, 100(R2)
7.     SUB R3, R4, R5
8.  L: SW R3, 50(R1)
```

Assuming that

- **there is no forwarding,**
- **zero testing is being resolved during ID, and**
- **registers can be written in the first of the WB cycle and also be read in the send half of the same WB cycle,**

Question: identify the resources of various hazards in the above code sequence.

Class Exercise

Consider the following code segment

```
1.      LW R1, 0(R4)
2.      LW R2, 0(R5)
3.      ADD R3, R1, R2
4.      BNZ R3, L
5.      LW  R4, 100(R1)
6.      LW R5, 200(R2)
7.      SUB R3, R4, R5
8.   L: SW R3, 50(R1)
```

Use compiler techniques to reshuffle/rewrite the code (without changing the meaning of the program) as to minimize data hazards as far as possible. Assume that no other general purpose registers other than those used in the code, are available.

Class Exercise

Consider the following code segment

```
1.      LW R1, 0(R4)
2.      LW R2, 0(R5)
3.      ADD R3, R1, R2
4.      BNZ R3, L
5.      LW  R4, 100(R6)
6.      LW R5, 200(R6)
7.      SUB R3, R4, R5
8.    L: SW R3, 50(R1)
```

```
1.      LW R1, 0(R4)
2.      LW R2, 0(R5)
3.        LW  R4, 100(R6)
4.        LW R5, 200(R6)
5.        ADD R3, R1, R2
6.      BNZ R3, L
7.      SUB R3, R4, R5
8.    L: SW R3, 50(R1)
```

Use compiler techniques to reshuffle/rewrite the code (without changing the meaning of the program) as to minimize data hazards as far as possible. Assume that no other general purpose registers other than those used in the code, are available.

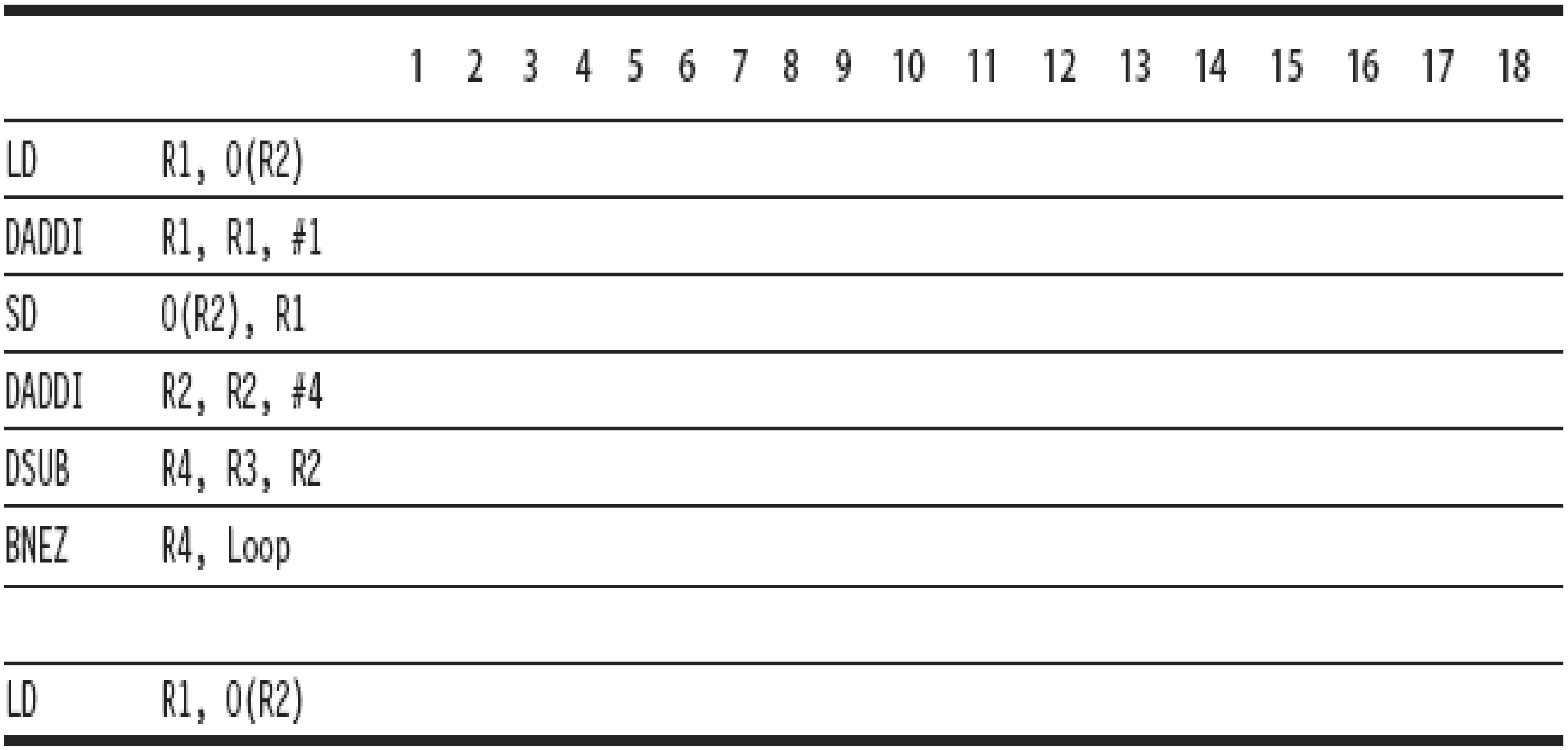
Sample Question

Use the following code fragment:

```
loop: LD R1,0(R2)          ; load R1 from address 0+R2
DADDI R1,R1,1              ;R1=R1+1
SD 0(R2),R1                ;store R1 at address 0+R2
DADDI R2,R2,4              ;R2=R2+4
DSUB R4,R3,R2              ;R4=R3-R2
BNEZ R4,loop               ;branch to loop is R4!=0
```

Assume that the initial value of R3 is R2+396. Let us use the classic RISC five-stage integer pipeline (see Figure A.1) and assume all memory accesses take 1 clock cycle.

Show the timing of this instruction sequence for the RISC pipeline without any forwarding or bypassing hardware but assuming a register read and a write in the same clock cycle “forwards” through the register file. Please fill up the following pipeline timing chart like Figure A.5. Assume that the branch is handled by flushing the pipeline. If all memory references take 1 cycle, how many cycles does this loop take to execute? (hints: branch outcomes and targets are not known until the end of the execute stage. All instructions introduced to the pipeline prior to this point are flushed.)



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
LD	R1, 0(R2)																		
DADDI	R1, R1, #1																		
SD	0(R2), R1																		
DADDI	R2, R2, #4																		
DSUB	R4, R3, R2																		
BNEZ	R4, Loop																		
LD	R1, 0(R2)																		

A.1 a. Forwarding is performed only via the register file. Branch outcomes and targets are not known until the end of the execute stage. All instructions introduced to the pipeline prior to this point are flushed.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`LD`	`R1, 0(R2)`	F	D	X	M	W													
`DADDI`	`R1, R1, #1`		F	s	s	D	X	M	W										
`SD`	`0(R2), R1`					F	s	s	D	X	M	W							
`DADDI`	`R2, R2, #4`								F	D	X	M	W						
`DSUB`	`R4, R3, R2`									F	s	s	D	X	M	W			
`BNEZ`	`R4, Loop`												F	s	s	D	X	M	W
`LD`	`R1, 0(R2)`																	F	D

Since the initial value of R3 is R2 + 396 and equal instance of the loop adds 4 to R2, the total number of iterations is 99. Notice that there are 8 cycles lost to RAW hazards including the branch instruction. Two cycles are lost after the branch because of the instruction flushing. It takes 16 cycles between loop instances; the total number of cycles is 98*16 + 18 = 1584. The last loop takes two addition cycles since this latency cannot be overlapped with additional loop instances.