CMPE110 Lecture 11
Pipelining III

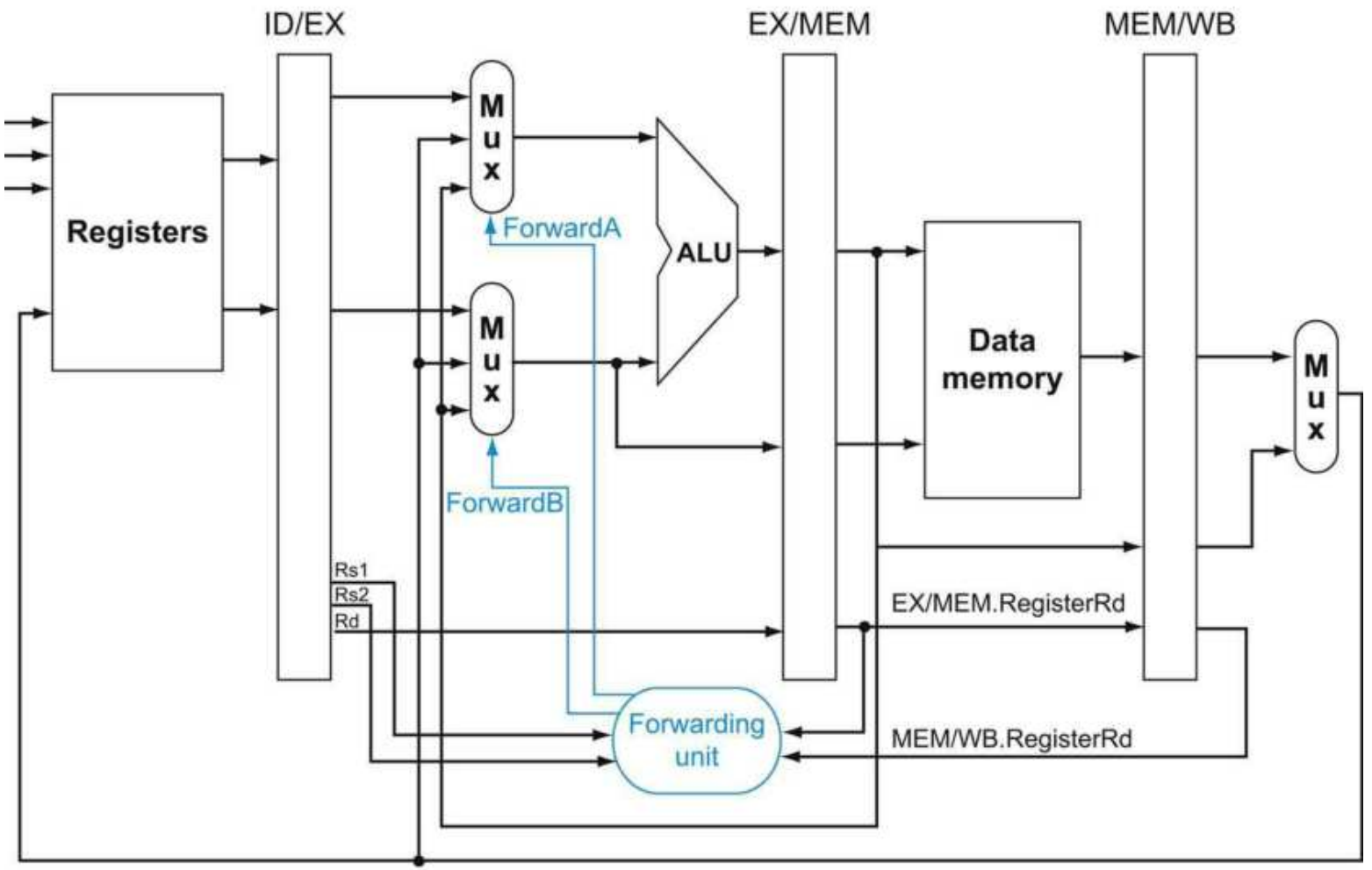
Heiner Litz

https://canvas.ucsc.edu/courses/12652

Announcements

Review

Forwarding Paths: Partial



Double Data Hazard

- Consider the sequence:

```
add $1,$1,$2
sub $1,$1,$3
or  $1,$1,$4
```

Mem Fwd

Ex Fwd

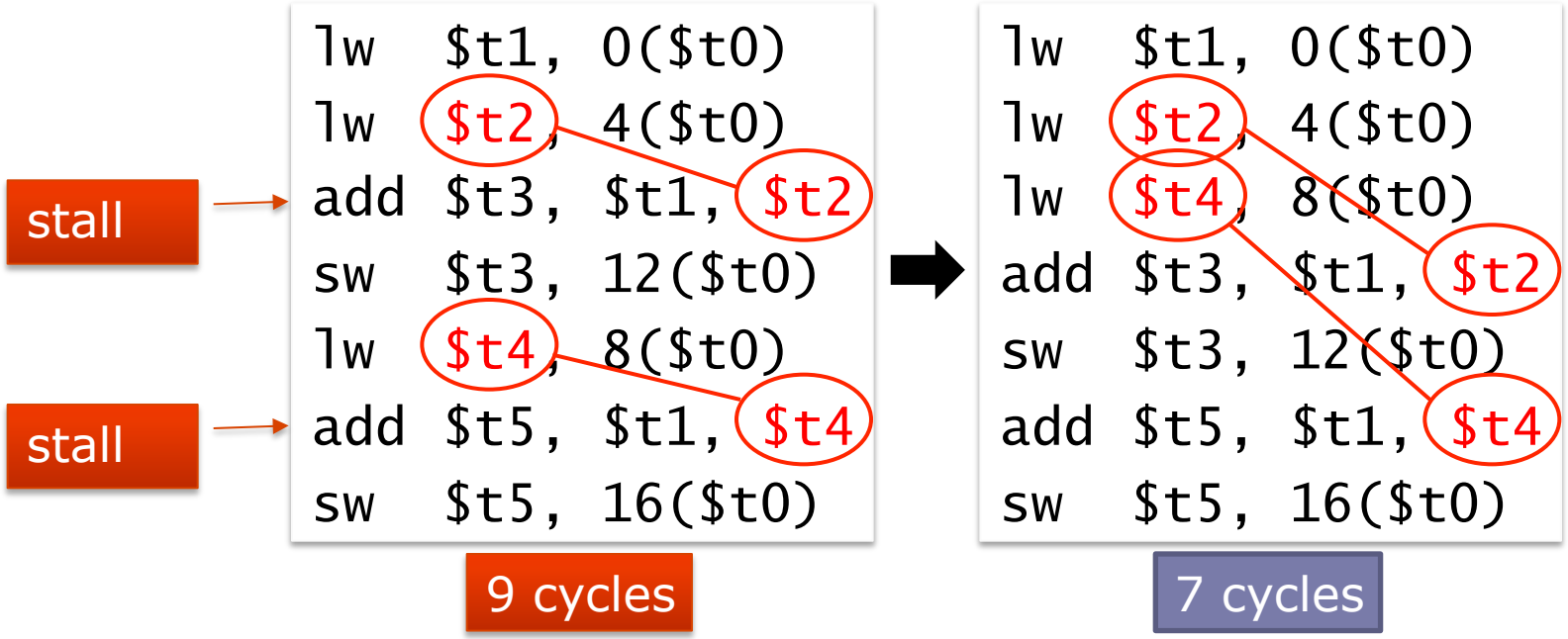
- Both hazards occur
  - Want to use the most recent result from the sub
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

Forwarding Control (Revised)

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for A = B + E; C = B + F;

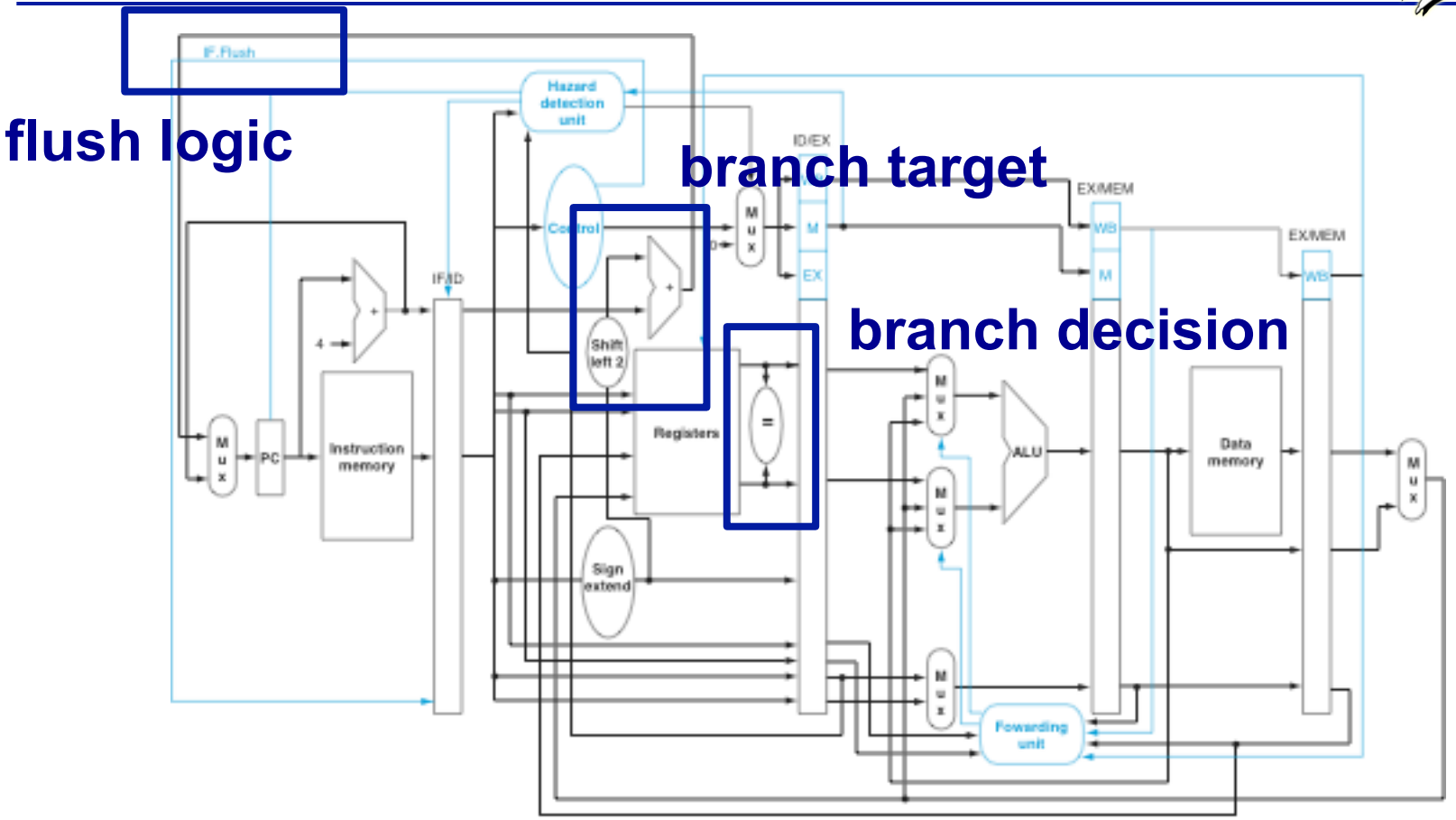


Control Hazards

Reducing Branch Delay

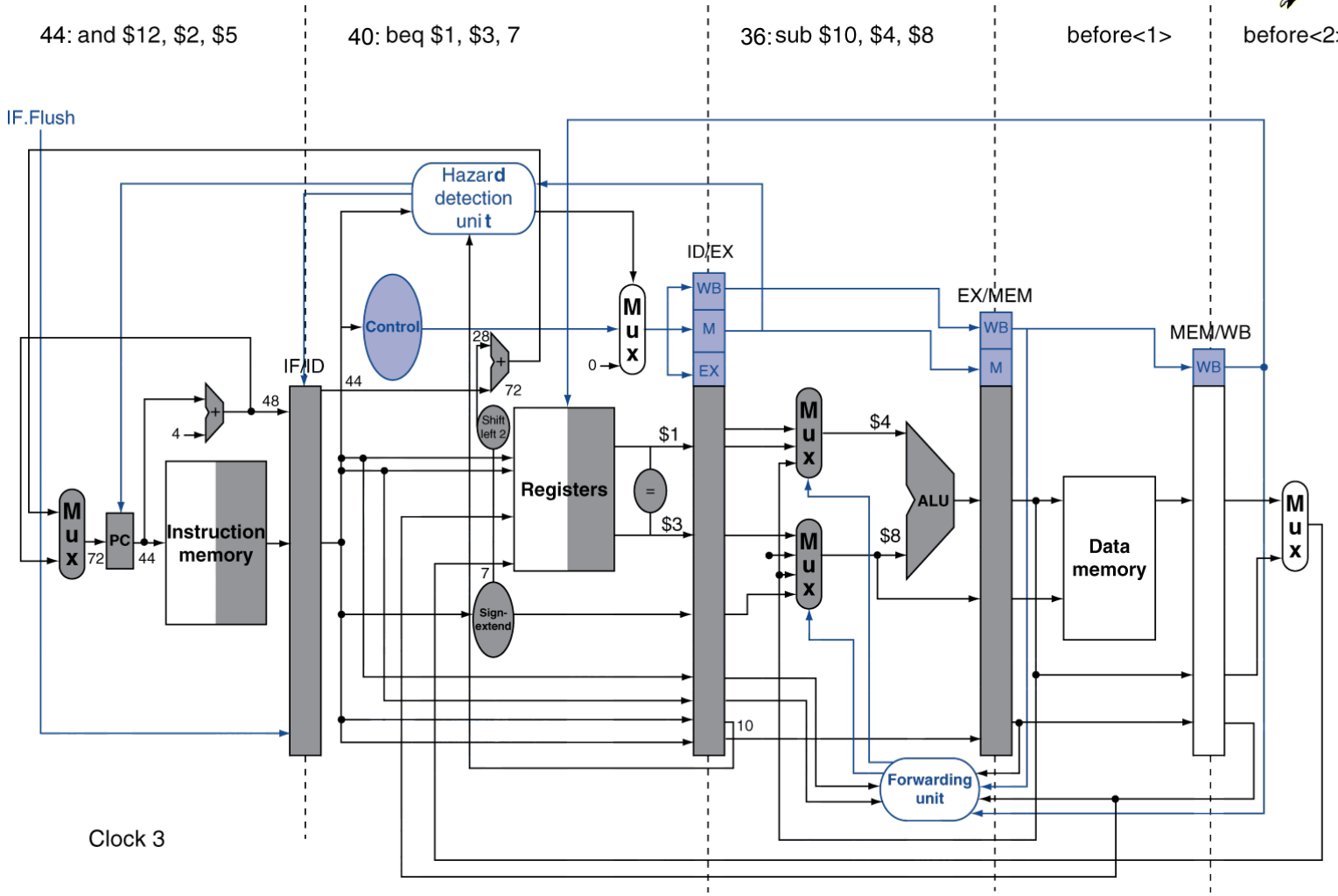
- Minimize “bubble” slots
  - Move branch computation earlier in the pipeline
  - branch outcome: add comparator to ID stage
  - branch target: add adder to ID stage
- Predict branch not taken
  - if correct, no bubbles inserted
  - if wrong, flush pipe, inserting one bubble

Reducing Branch Delay

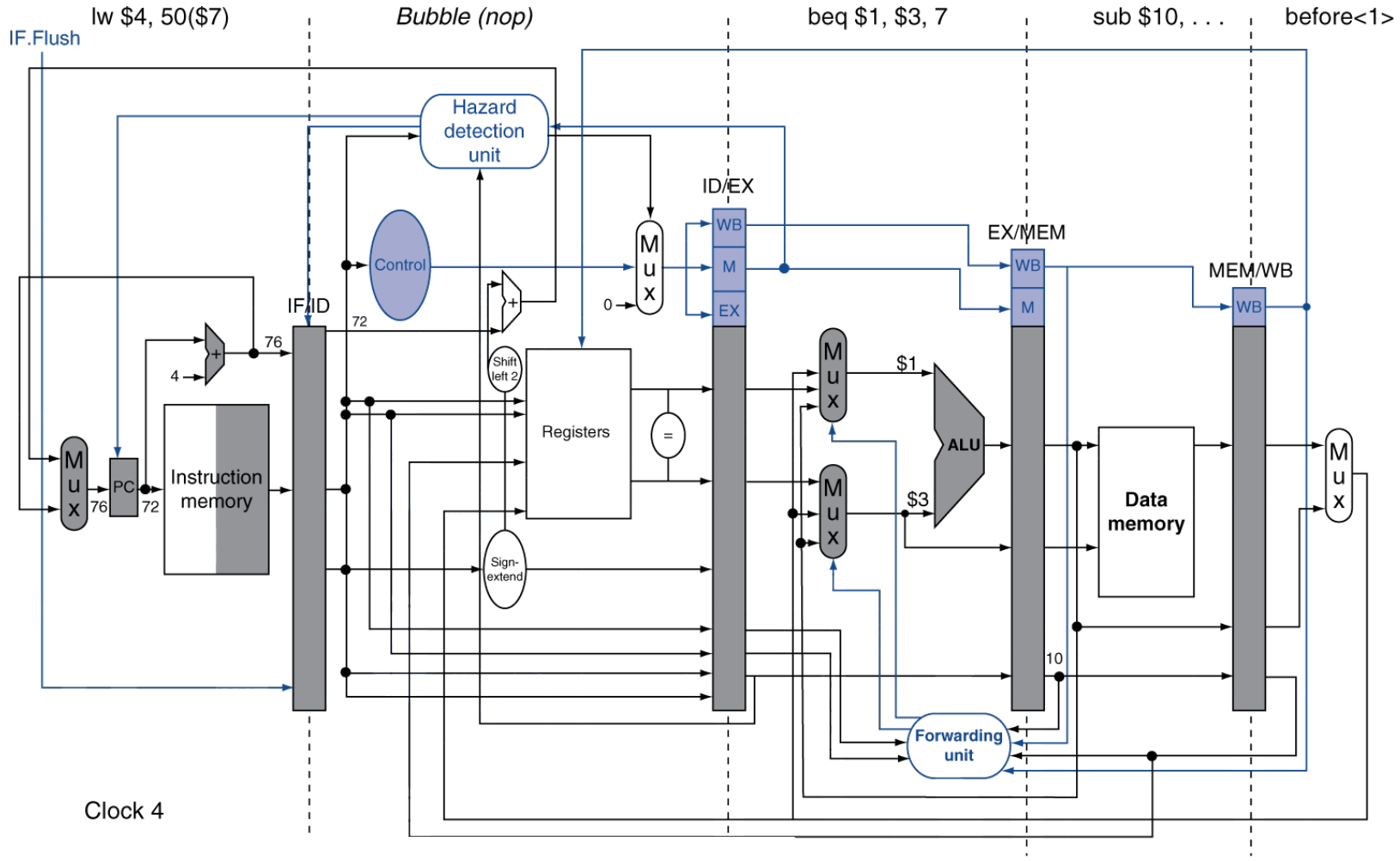


move branch computation earlier in the pipeline
enable flush capability if mispredicted

Example: Branch Taken



Example: Branch Taken



nop inserted using IF.Flush

How to Flush?

- 5-stage Pipeline
  - only one instruction to flush
  - Can do by “flushing” the output register of IF
- Deeper Pipelines
  - Multiple cycles to determine branch outcome
  - Need to know how many instructions in the pipe
  - Flush all state changing actions
    - RegWrite, MemWrite, PCWrite (jmp, beq), overflow..

Performance Impact of Branch Stalls

- Need to stall for one cycle on every branch
- Consider the following case
  - The ideal CPI of the machine is 1
  - The branch causes a stall
- Effective CPI if 15% of the instructions are branches?
  - The new effective CPI is $1 + 1 \times 0.15 = 1.15$
  - The old effective CPI was $1 + 3 \times 0.15 = 1.45$

Delayed Branches

- An ISA-based solution used in early MIPS machines
  - Had a branch delay of one

- Example:

beq r1, r2, L	Branch instruction
sub r4, r1, r3	This operation ALWAYS is executed
and r6, r2, r7	This operation executes if branch fails

- Worked well initially
  - Compiler can fill one slot 50% of the time (50% nops)
- For modern processors, it is a pain
  - Many delay slots needed due to deeper pipelines
  - Processors deal with this using branch prediction
  - Delayed branches complicate exceptions as well

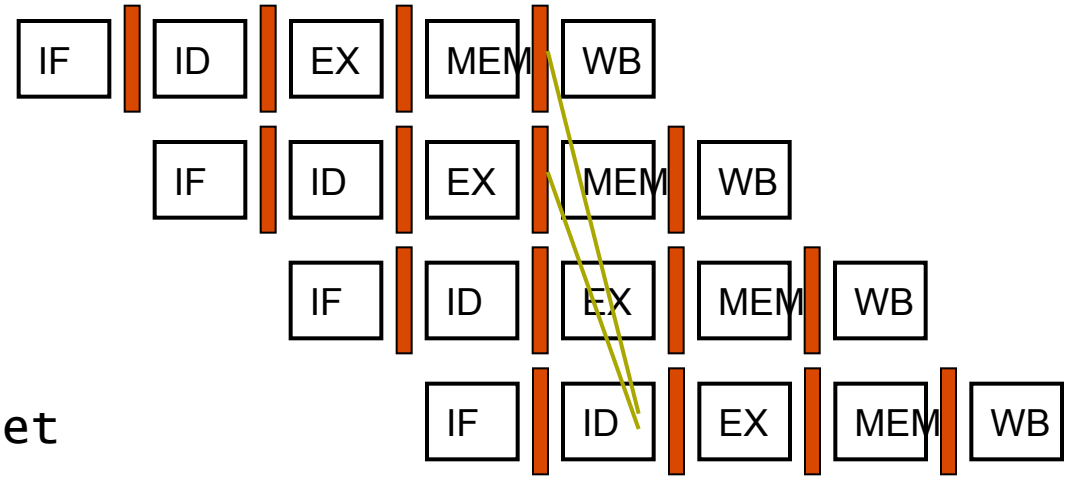
It's All Connected

- We moved the branch control point to ID
- Which means we consume two registers in ID
  - earlier in the pipeline as before (EX)
- What does this mean for forwarding/stalls?
  - Need to check!!

Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```

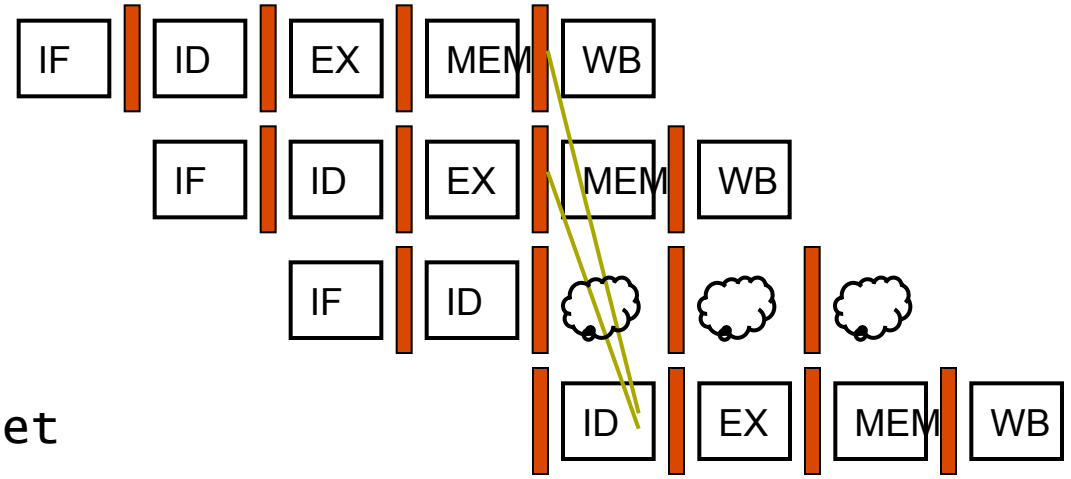


- Can resolve using forwarding
  - Additional datapaths and control

Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

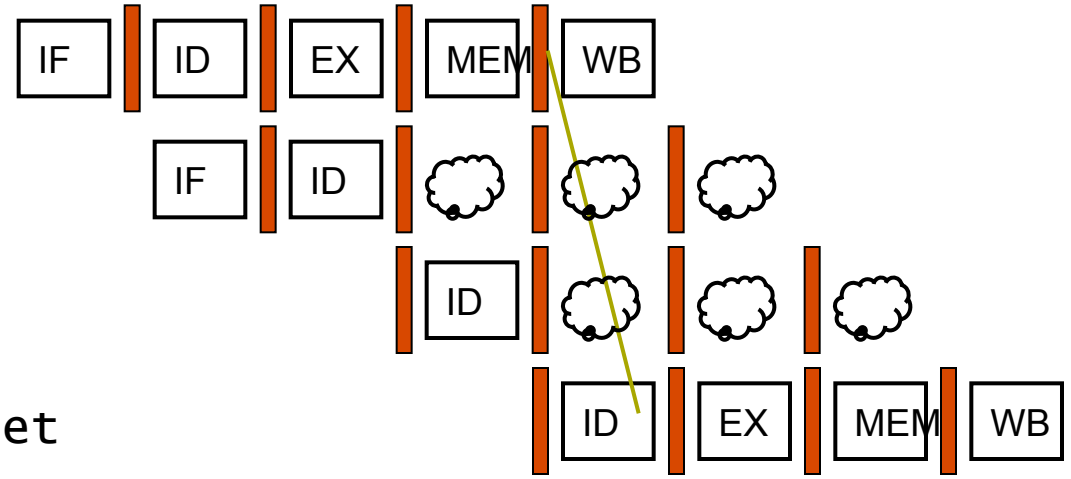
```
lw   $1, addr
add  $4, $5, $6
beq  stalled
beq  $1, $4, target
```



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

```
lw   $1, addr
beq  stalled
beq  stalled
beq  $1, $0, target
```



Branch CPI

15% Branch frequency

65% branches are taken

50% of single delay slots are filled usefully

Control point	Branch strategy	Branch Stall CPI
MEM	stall	
ID	stall	
ID	Predict taken	
ID	Predict Not taken	
ID (target) / MEM (==)	Predict taken	
ID	Delayed branch	

Branch Prediction

- Static prediction vs. dynamic prediction

- Static prediction schemes:
  - always predict taken
  - always predict not-taken
  - compiler/programmer hint
  - if (target < PC)
    predict taken
    else
    predict not-taken

What's the rationale behind this?

Dynamic Branch Prediction

- Branch History Table (BHT)
  - One entry for each branch PC
  - Taken/Not taken bit
- Branch Target Buffer (BTB)
  - One entry for each branch PC
  - Target address
- Increasingly important for long pipelines (IDx)
  - x86 vs. RISC-V instruction decode

Branch Prediction Example


```
void foo() {
  for(e=0;e<4;e++) {
    stuff …
  }
}

foo:j E
L:  stuff..
    ..
E:  bne $t3,$t4,L
```

How many times will bne at E: be predicted correctly?

Two-bit saturating counter:
strongly taken, weakly taken, weakly n-t, strongly n-t

Exceptions and Interrupts

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Switch from user to privileged (kernel) mode
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
  - E.g., undefined opcode, overflow, syscall, …
- Interrupt
  - E.g., from an external I/O controller (network card)
- Dealing with them without sacrificing performance is hard

Handling Exceptions

- Something bad happens to an instruction
  - Need to stop the machine
  - Fix up the problem (with privileged code like the OS)
  - Start the machine up again

- MIPS: managed with System Control Coprocessor (CP0)
  - Save PC of offending (or interrupted) instruction
    - In MIPS: Exception Program Counter (EPC)
  - Save indication of the problem
    - In MIPS: Cause register
    - We'll assume 1-bit for the examples
      - 0 for undefined opcode, 1 for overflow
  - Jump to handler code at hardwired address (e.g., 8000 00180)

An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
  - Avoids the software overhead of selecting the code to run based on cause reg
  - Jump table

- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - …: C000 0040

- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

Handler Actions

- Read cause, and transfer to relevant handler
  - A software switch statement
  - May be necessary even if vectored interrupts are available
- Determine action required
- If restartable
  - Take corrective action
  - Use EPC to return to program
- Otherwise
  - Terminate program (e.g., segfault, …)
  - Report error using EPC, cause, …

Precise Exceptions

- Definition: precise exceptions
  - All previous instructions had completed
  - The faulting instruction was not started
  - None of the following instructions were started
    - No changes to the architecture state (registers, memory)
- Why are they desirable by OS developers?
- With single cycle design, precise exceptions are easy
  - Why?
- With pipelined design, they can be tricky
  - Why?

Exceptions in a Pipeline

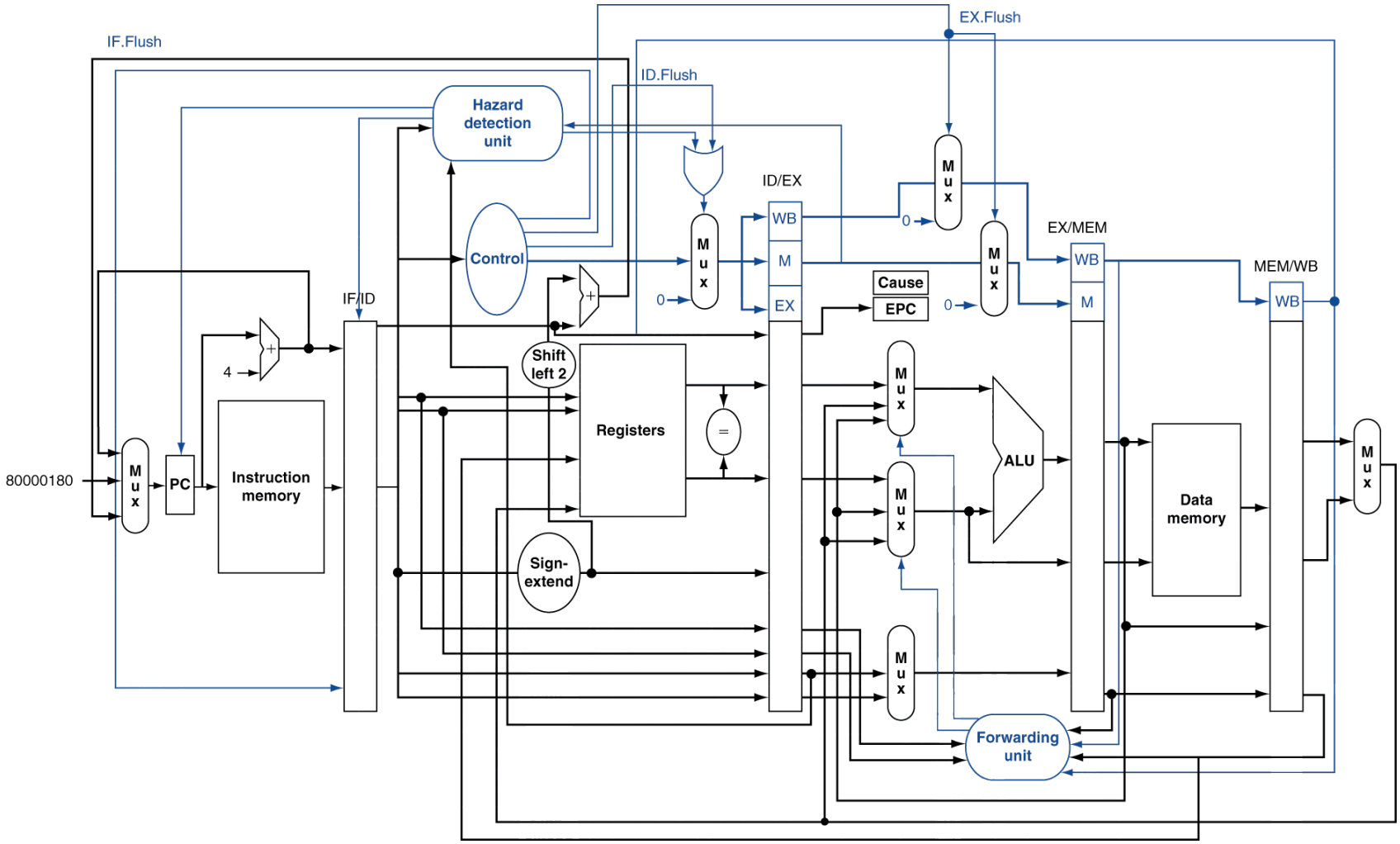
- Another form of control hazard
- Consider overflow on add in EX stage

  `add $1, $2, $1`
  - Prevent $1 from being clobbered by add
  - Complete previous instructions
  - Nullify subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Uses much of the same hardware

Nullifying Instructions

- Nullify = turn an instruction into a nop (or bubble)
  - Reset its RegWrite and MemWrite signals
    - Does not affect the state of the system
  - Resets its branch and jump signals
    - Does not cause unexpected flow control
  - Mark that it should not raise any exceptions of its own
    - Does not cause unexpected flow control
  - Let it flow down the pipeline

5-stage Pipeline with Exceptions



Exception Example

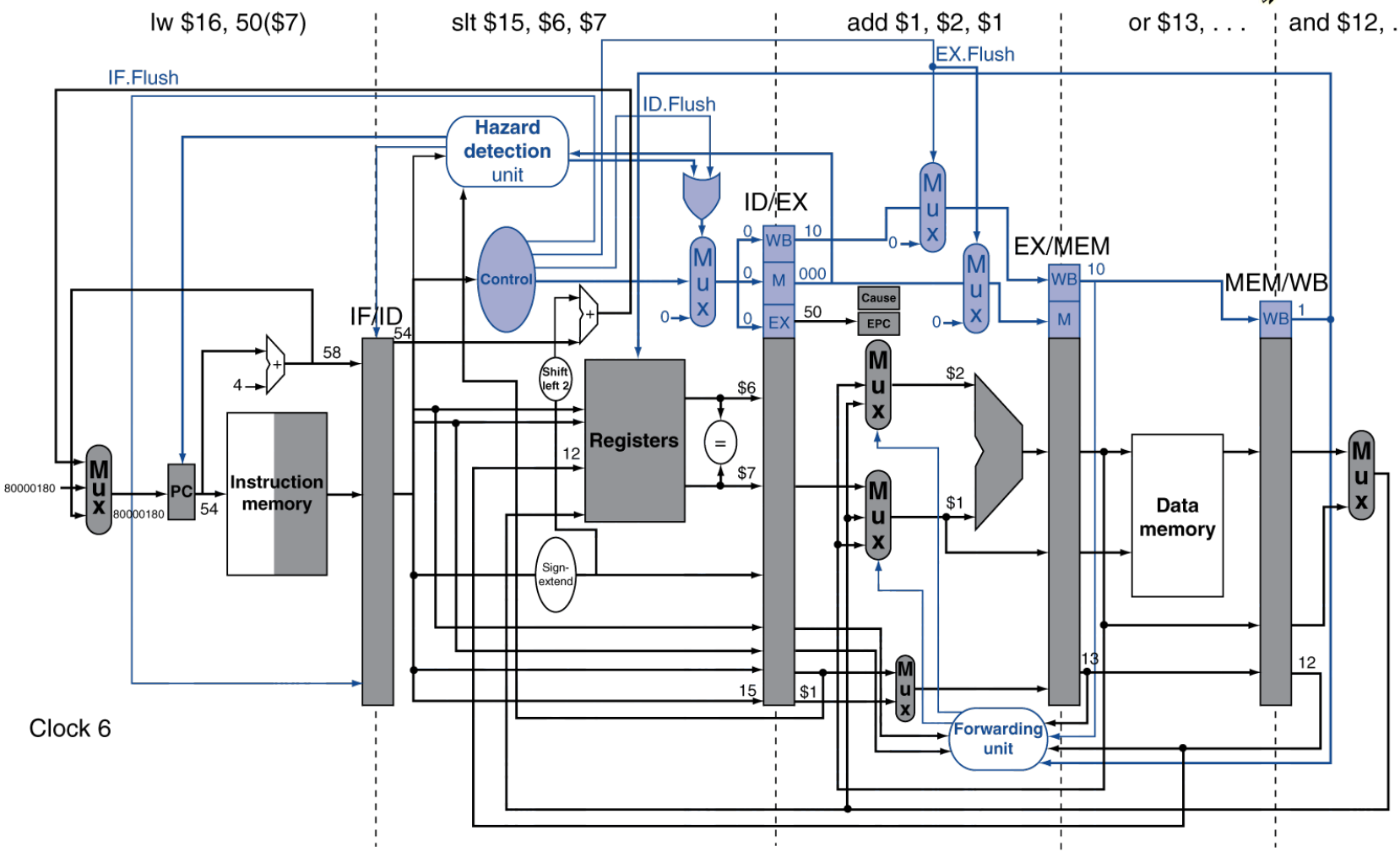
- Exception on add in

```
40      sub   $11, $2, $4
44      and   $12, $2, $5
48      or    $13, $2, $6
4C      add   $1,  $2, $1
50      slt   $15, $6, $7
54      lw    $16, 50($7)
…
```

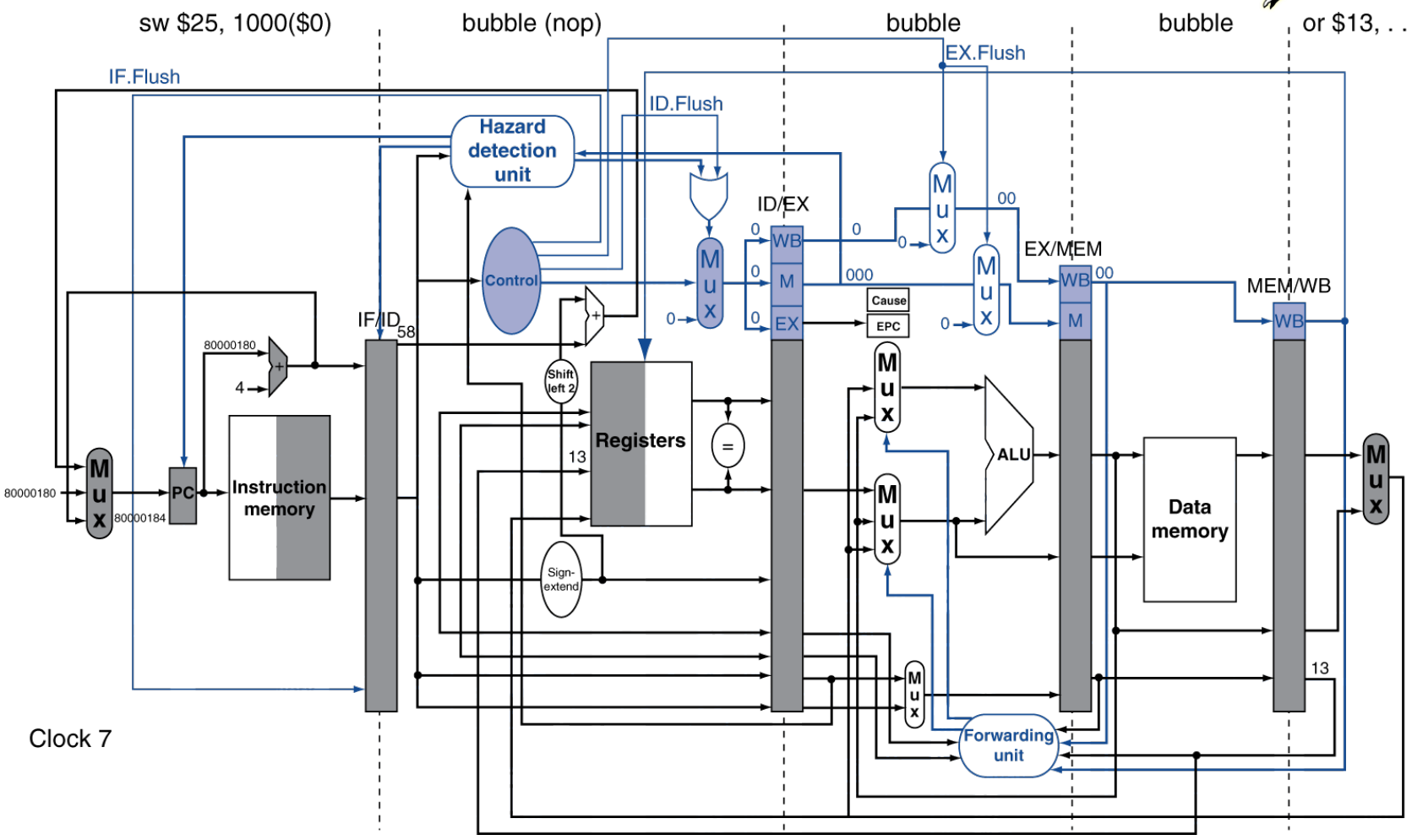
- Handler

```
80000180    sw    $25, 1000($0)
80000184    sw    $26, 1004($0)
…
```

Exception Example



Exception Example



Dealing with Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception in program order
  - Single commit point (MEM or WB stage)
  - Flush subsequent instructions
  - Necessary for “precise” exceptions

Beyond the 5-stage Pipeline

And why you should take CS316 ☺

- Advanced Multi-core Systems
- Topics: advanced core design, GPU architecture, multi-threading, support for parallel programming, …
- And a research project

Beyond the 5-stage Pipeline

Desirable properties

- Higher clock frequency
- $CPI_{base} < 1$
- Avoid in-order stalls
  - When instruction stalls, independent instructions behind it stall
- Avoid stalls due to branches

How do we go about it?

Reminder:
Key Tools for System Architects

1. Pipelining
2. Parallelism
3. Out-of-order execution
4. Prediction
5. Caching
6. Indirection
7. Amortization
8. Redundancy
9. Specialization
10. Focus on the common case

Modern Processors

High clock frequency → deep pipelines

10 – 20 stages

$CPI_{base} < 1$ → superscalar pipelines

Launch 2, 3, or 4 instructions every cycle

Avoid in-order stalls → out-of-order execution

Re-order instructions based on dependencies

Avoid stalls due to branches → branch prediction

Keep history about direction and target of branches

Why Does it Work: Instruction-level Parallelism

Independence among instructions

Example with ILP

```
add $t0, $t1, $t2
or  $t3, $t1, $t2
sub $t4, $t1, $t2
and $t5, $t1, $t2
```

Most programs have ILP
But it is not infinite
Scope also matters a lot

Example with no ILP

```
add   $t0, $t1, $t2
or  $t3, $t0, $t2
sub $t4, $t3, $t2
and $t5, $t4, $t2
```

Question

How much ILP is there in common programs?

Multiple Instruction Issue (Superscalar)

Fetch and execute multiple instructions per cycle => CPI < 1

Example: fetch 2 instructions/clock cycle

Dynamically decide if they can go down the pipe together or not (hazard)

Pipe Stages

IF	ID	EX	MEM	WB		
IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB	
Ideal CPI = 0.5	IF	ID	EX	MEM	WB	
		IF	ID	EX	MEM	WB
		IF	ID	EX	MEM	WB

Resources: double amount of hardware (FUs, Register file ports)

Issues: hazards, branch delay, load delay

Deeper Pipelining

Increase number of pipeline stages

- Fewer levels of logic per pipe stage
- Higher clock frequency

Example 9-stage pipeline

Pipe Stages

IF1	IF2	ID	EX	MEM1	MEM2	MEM3	WB	
	IF1	IF2	ID	EX	MEM1	MEM2	MEM3	WB

Issues: branch delay, load delay: CPI = 1.4 - 2.0, cost of pipeline registers

Modern pipelines

- Pentium: 5 stages
- Original Pentium Pro: 12 stages
- Pentium 4: 31 stages, 3+ GHz
- Tejas: 40-50 stages, 10 GHz, cancelled!
- Core 2: 14 stages

Dynamic Scheduling

Execute instructions out-of-order

Fetch multiple instructions per cycle using branch prediction

Figure out which are independent and execute them in parallel

Example

```
add   $t0, $t1, $t2
or  $t3, $t0, $t2
sub $t0, $t1, $t2
and $t5, $t0, $t2
```

Superscalar + Dynamic scheduling

```
add   $t0, $t1, $t2        sub $t0, $t1, $t2
or  $t3, $t0, $t2          and $t5, $t0, $t2
```

What's wrong with this?

End of Uniprocessor Performance Improvements