COMP3211 03S2 Lecture 25

Pipelined Processor Design

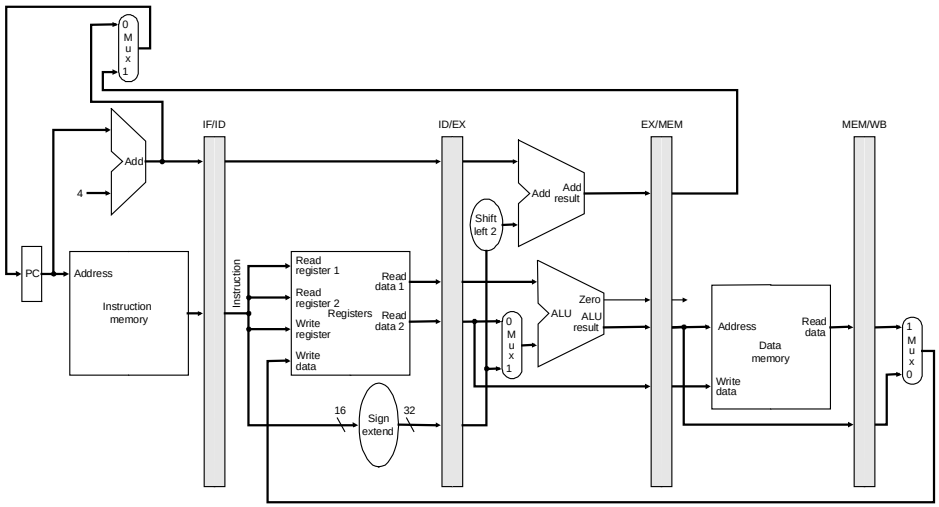
Adapted from

CS152: Computer Architecture and Engineering
Dave Patterson (www.cs.berkeley.edu/~pattrsn)

Recap: How do we pipeline the datapath ?

- Partition the datapath into several stages with
  - Minimize the number of stages given the clock cycle time
  - Balance the delays across the stages
- Associate resources with each stage
- Insert registers between stages
  - To store data transferred from one stage to the next stage
  - To forward control signals for subsequent stages
- Ensure flows do not conflict, or figure out how to resolve conflicts
- Assert control at appropriate stages

Recap: A draft of pipelined datapath



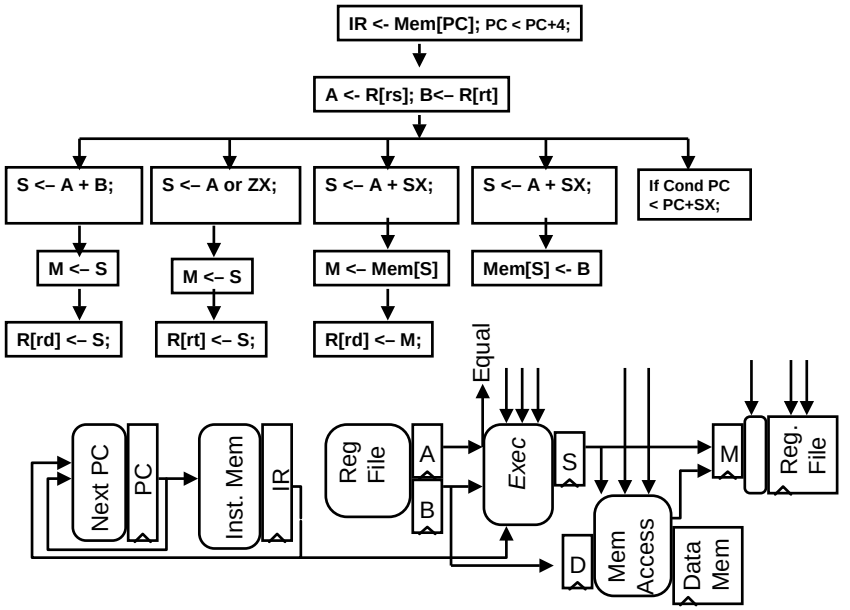
Control design

- Tasks we need to do
  - Determine the control signals for each stage
    - Those that are the same as for the multicycle control
      - Generate a set of signals for each cycle
    - Those also appearing in the single-cycle control
      - Generate them once per instruction
  - Ensure the control signals apply to the right instruction
    - Control signals need to flow through the pipeline together with the corresponding instruction
    - Control signals thus need to be buffered in the pipeline registers as well

Control Signals

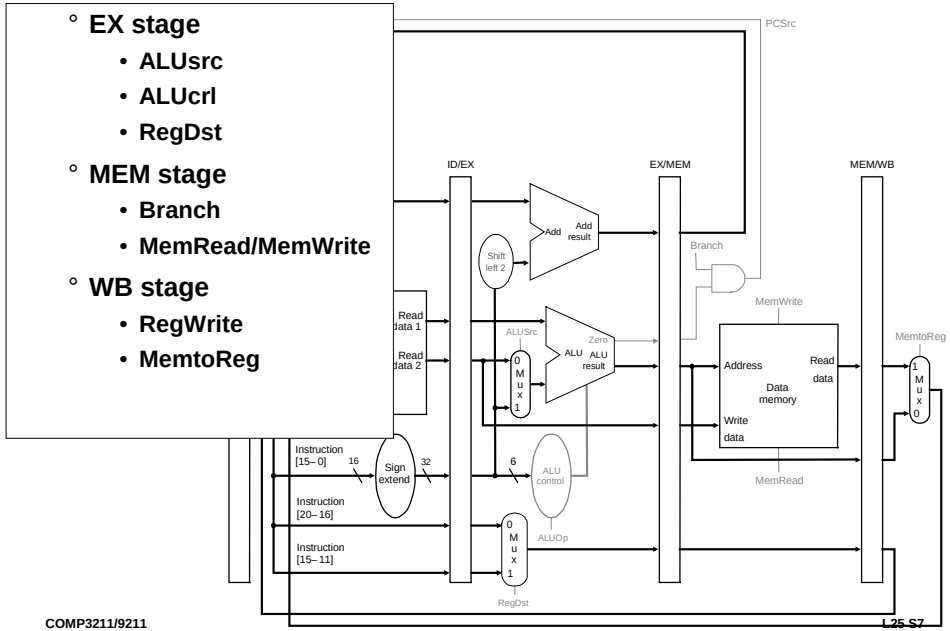
- No control for the pipeline registers
  - The stage registers are written each clock cycle
- Control signals for the first two stages are always asserted
  - They are common for all instructions
- Control signals for the last three stages depend upon the instruction
  - Need to be buffered
  - More pipeline registers required

Control Diagram



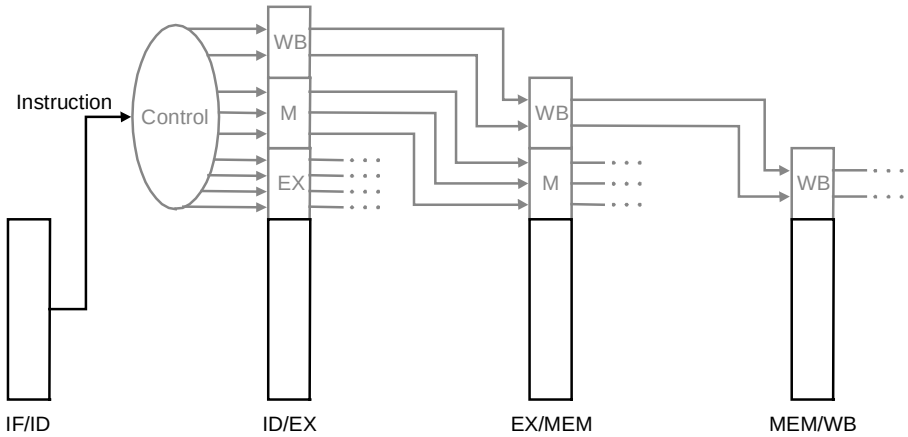
Control signals for the last three stages

- EX stage
  - ALUsrc
  - ALUcrl
  - RegDst
- MEM stage
  - Branch
  - MemRead/MemWrite
- WB stage
  - RegWrite
  - MemtoReg

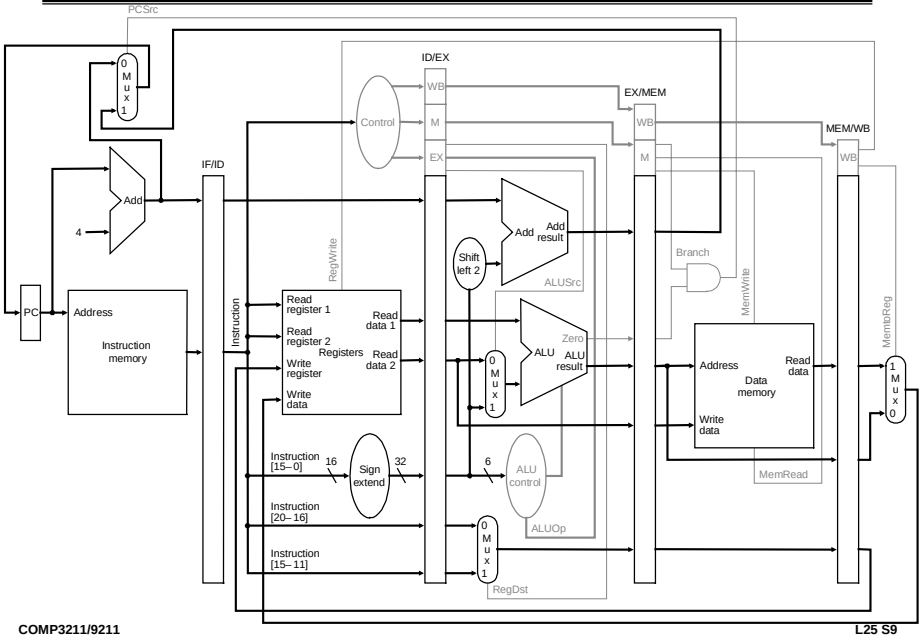


Control signals for the last three stages (cont.)

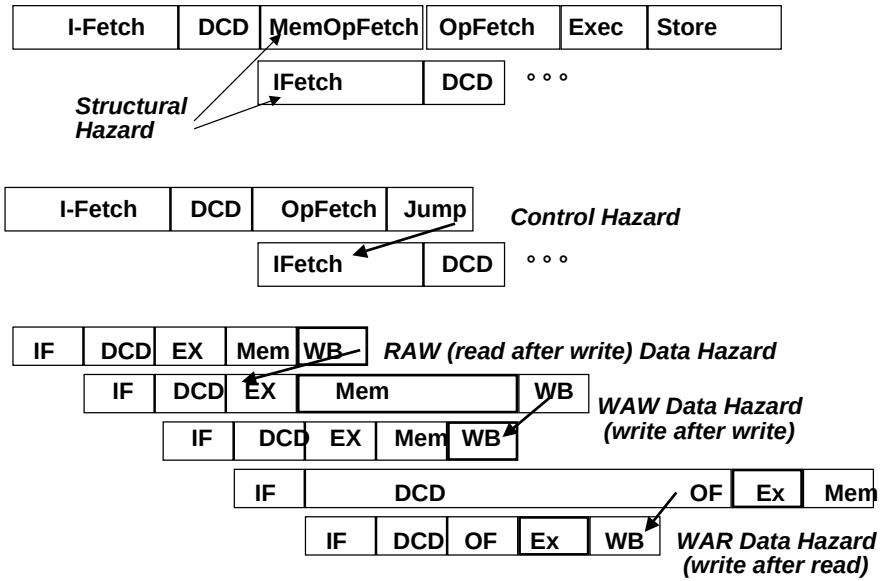
- We need more registers to store the control signals for each stage



Pipelining the control signals

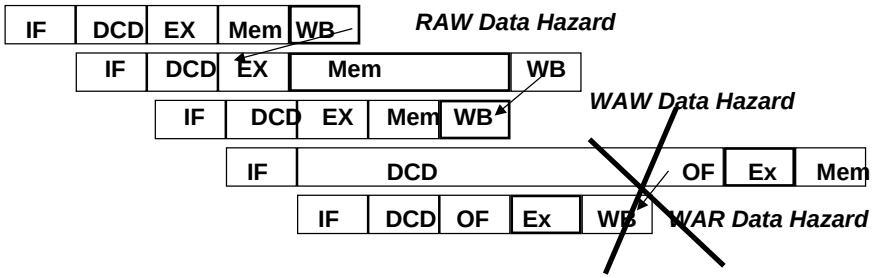


Pipeline Hazards Again



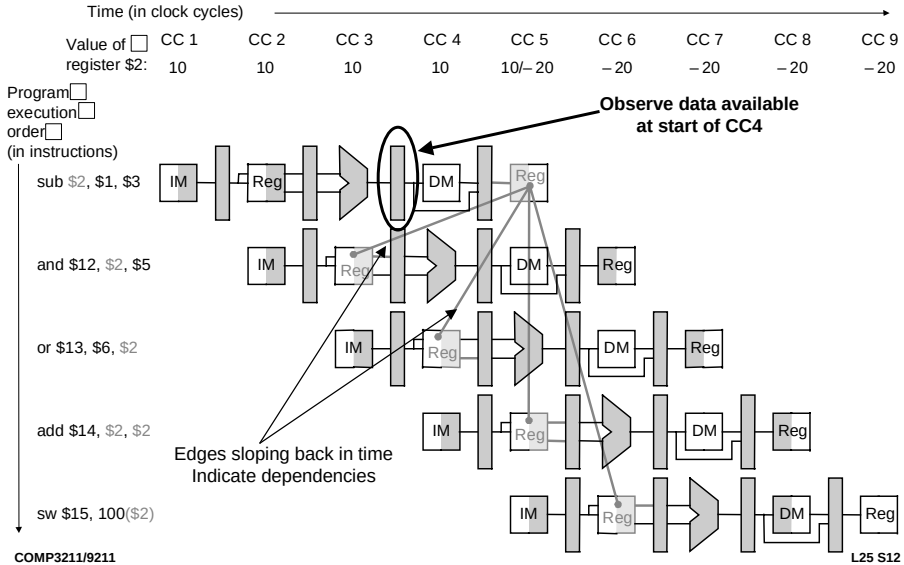
Data Hazards

- Avoid some "by design"
  - eliminate WAR by always fetching operands early (DCD) in pipe
  - eleminate WAW by doing all WBs in order (last stage, static)
- Detect and resolve remaining ones
  - stall or forward (if possible)



Data hazards

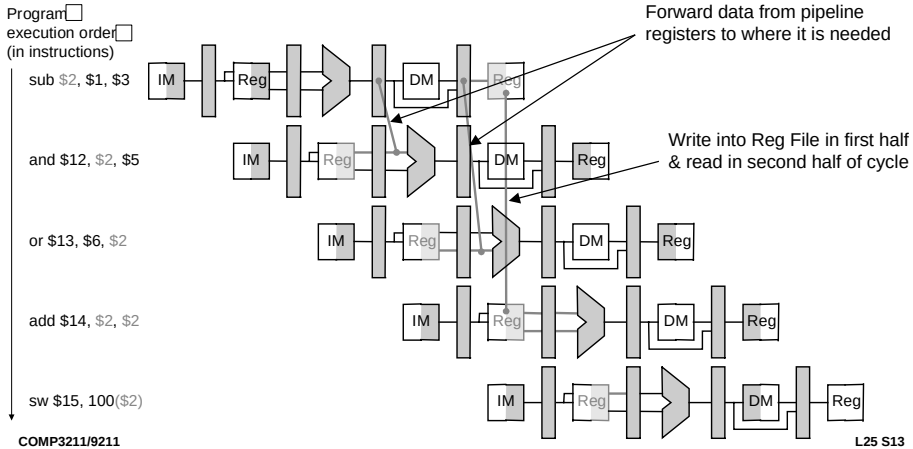
- Happen when there are data dependencies between instructions executing in the pipeline



Solution: Forwarding

- A very efficient approach in avoiding hazard

	Time (in clock cycles) →								
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/– 20	– 20	– 20	– 20	– 20
Value of EX/MEM :	X	X	X	– 20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	– 20	X	X	X	X

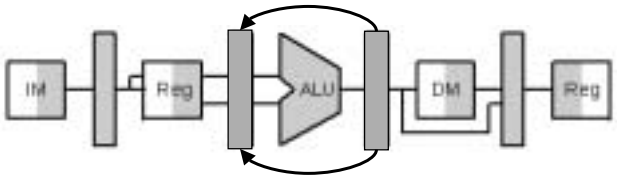


Implementing forwarding

- Detect data hazards
  - Check for data dependencies between each instruction and the preceding instructions in the pipeline
- Forward the proper value

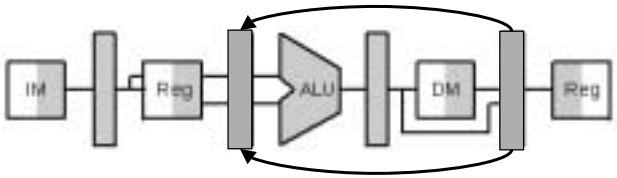
Detecting a data hazard (1)

- Based on pipeline register field contents
- Dependent on the immediately preceding instruction
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
- Dependent on the instruction before that
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt



Detecting a data hazard (2)

- Based on pipeline register field contents
- Dependent on the immediately preceding instruction
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
- Dependent on the instruction before that
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt



Detecting a data hazard – example

```
Sub    $2, $1, $3      IF  ID  EX  ME  WB
And    $12, $2, $5         IF  ID  EX  ME  WB
Or     $13, $6, $2             IF  ID  EX  ME  WB
Add    $14, $2, $2                 IF  ID  EX  ME  WB
Sw     $15, 100($2)                    IF  ID  EX  ME  WB
```


Detecting a data hazard – example

```
Sub    $2, $1, $3      IF  ID  EX | ME  WB
And    $12, $2, $5         IF  ID | EX  ME  WB
Or     $13, $6, $2             IF  ID  EX  ME  WB
Add    $14, $2, $2                 IF  ID  EX  ME  WB
Sw     $15, 100($2)                    IF  ID  EX  ME  WB
```

Dependent instructions: sub-and

Data hazard type: 1a

Detecting a data hazard – example

```
Sub    $2, $1, $3      IF  ID  EX  ME | WB
And    $12, $2, $5         IF  ID  EX | ME  WB
Or     $13, $6, $2             IF  ID | EX  ME  WB
Add    $14, $2, $2                 IF  ID  EX  ME  WB
Sw     $15, 100($2)                    IF  ID  EX  ME  WB
```

Dependent instructions: sub-and, sub-or

Data hazard type: 1a, 2b

Detecting a data hazard – more considerations

- Previous instructions must
  1. Write back a result
  2. If they write to $0, the result is never stored, and hence need not be forwarded
     - $0 is a special register in MIPS that always has value 0

Revised Detect Condition – EX hazard

° Data forwarded from EX stage:

Detection:

```
If (EX/MEM.RegWrite
And (EX/MEM.RegisterRd != 0)
And (EX/MEM.RegisterRd = ID/EX.RegisterRs))
        forward

If (EX/MEM.RegWrite
And (EX/MEM.RegisterRd != 0)
And (EX/MEM.RegisterRd = ID/EX.RegisterRt))
        forward
```


Revised Detect Condition – MEM hazard

° Data forwarded from MEM stage:

Detection:

```
If (MEM/WB.RegWrite
And (MEM/WB.RegisterRd != 0)
And (MEM/WB.RegisterRd = ID/EX.RegisterRs))
        forward

If (MEM/WB.RegWrite
And (MEM/WB.RegisterRd != 0)
And (MEM/WB.RegisterRd = ID/EX.RegisterRt))
        forward
```


Multiple data dependencies

° What if a data is dependent on both EX and MEM results?

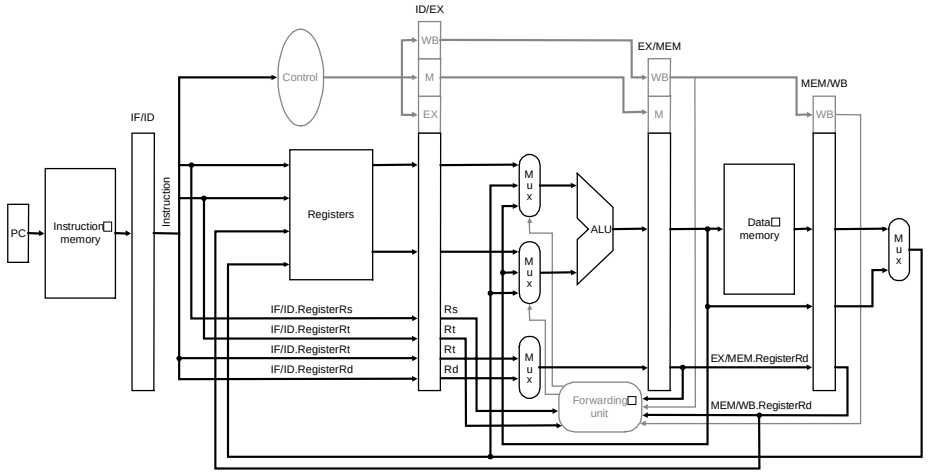
```
Add $1, $1, $2
Add $1, $1, S3
Add $1, $1, $4
```

° Forward the more recent results – forward the EX result

Detection:

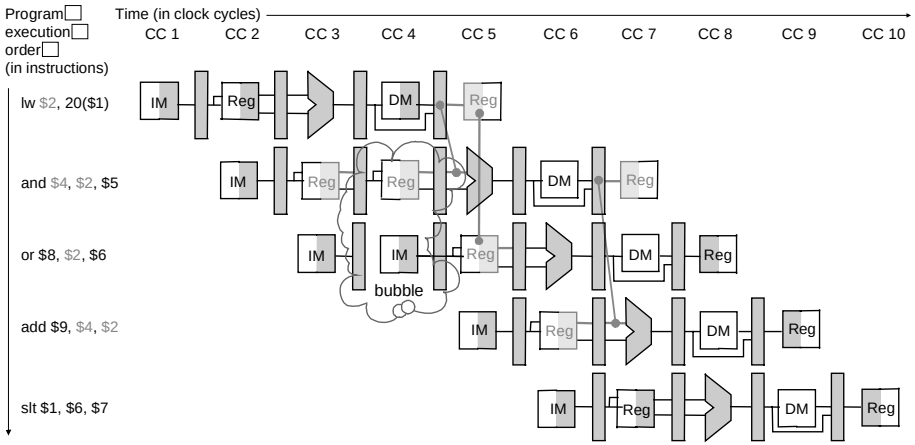
```
If (MEM/WB.RegWrite
And (MEM/WB.RegisterRd != 0)
And (EX/MEM.RegisterRd != ID/EX.RegisterRs
And (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      forward MEM/WB.result

If (MEM/WB.RegWrite
And (MEM/WB.RegisterRd != 0)
And (EX/MEM.RegisterRd != ID/EX.RegisterRt
And (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      forward MEM/WB.result
```


Datapath with modified forwarding unit



When forwarding does not work ⇒ STALL

- Detect hazard
  - Stall at ID stage



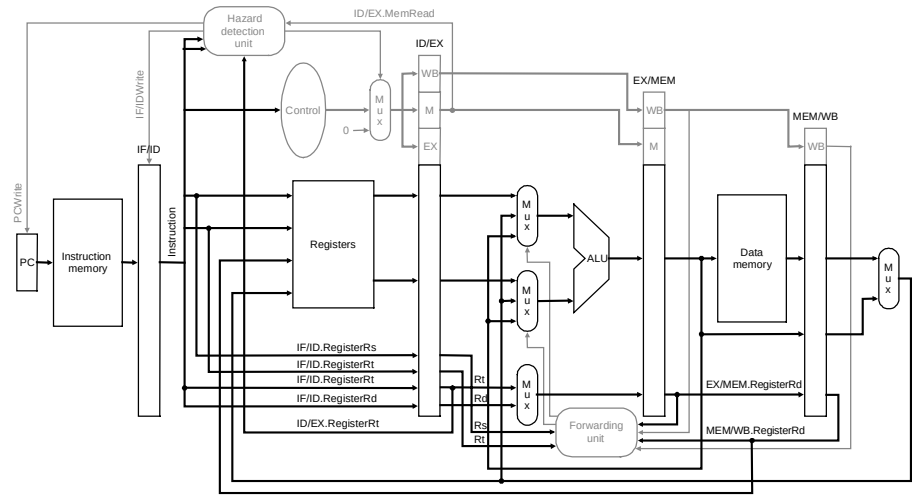
Detecting hazards

- When a load instruction is followed by an instruction dependent on the memory data (load-use hazard)
- We add a hazard detection unit that checks for the following condition:

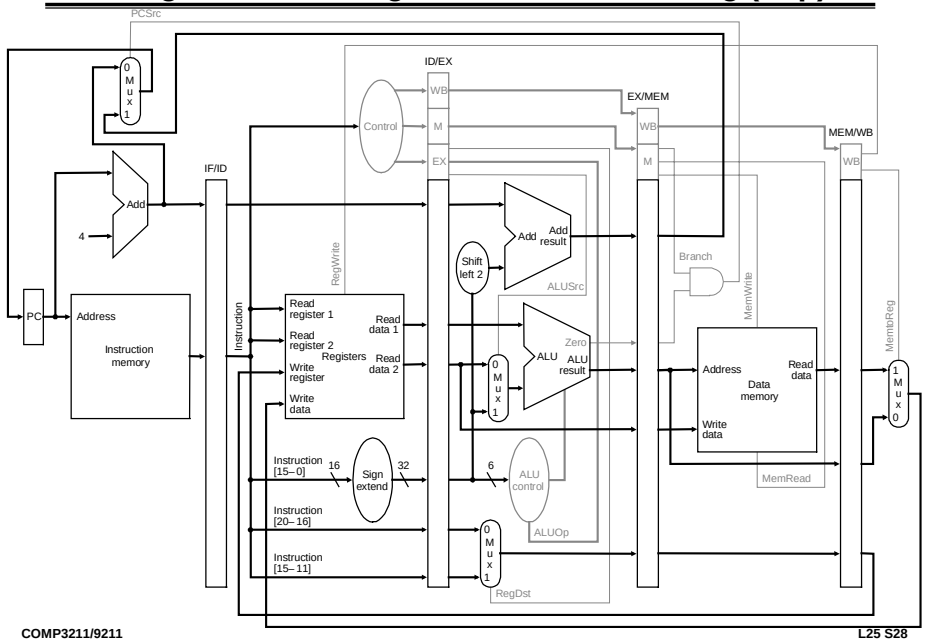
If (ID/EX.MEMRead
and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
or (ID/EX.RegisterRt = IF/ID.RegisterRt)))
stall the pipeline

How do we stall the pipeline?

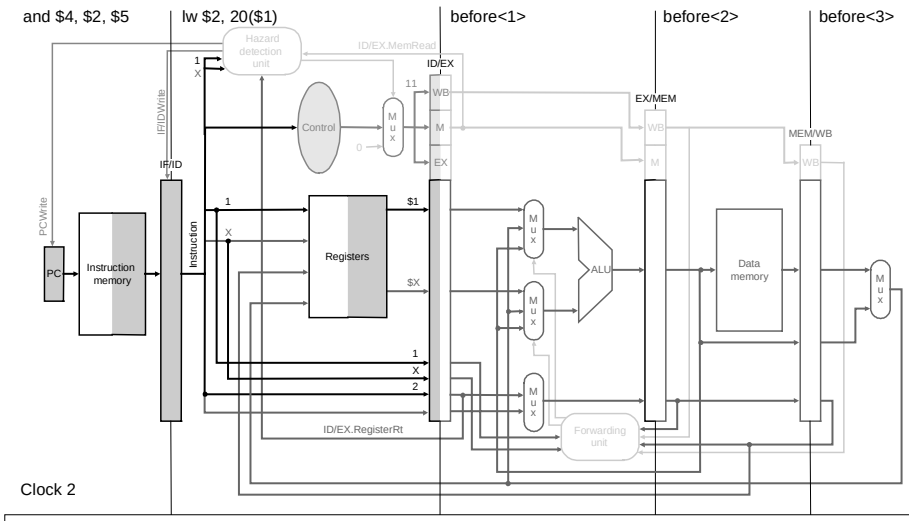
- Prevent the PC and IF/ID pipeline register from changing
- Set the EX, MEM and WB control fields of the ID/EX pipeline register to 0



Setting the control signals to 0 does nothing (nop)

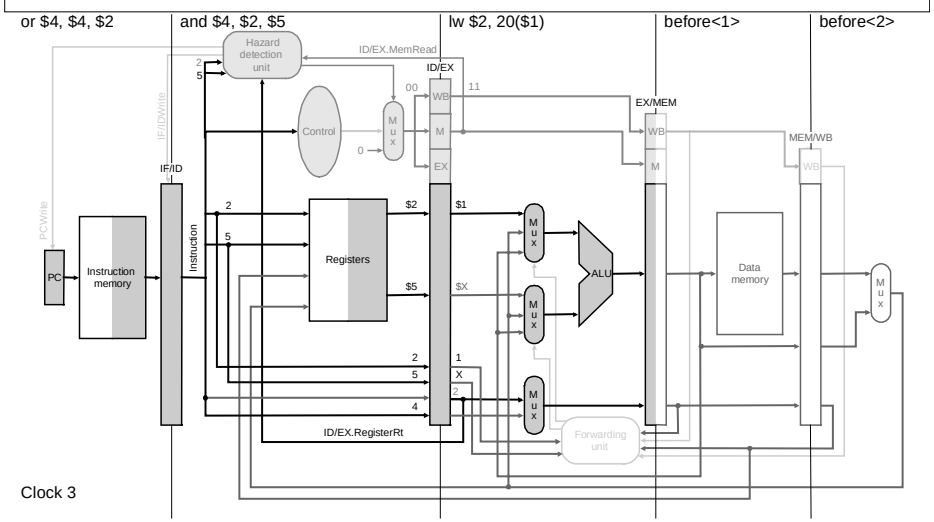


Load-use hazard (LUH) example



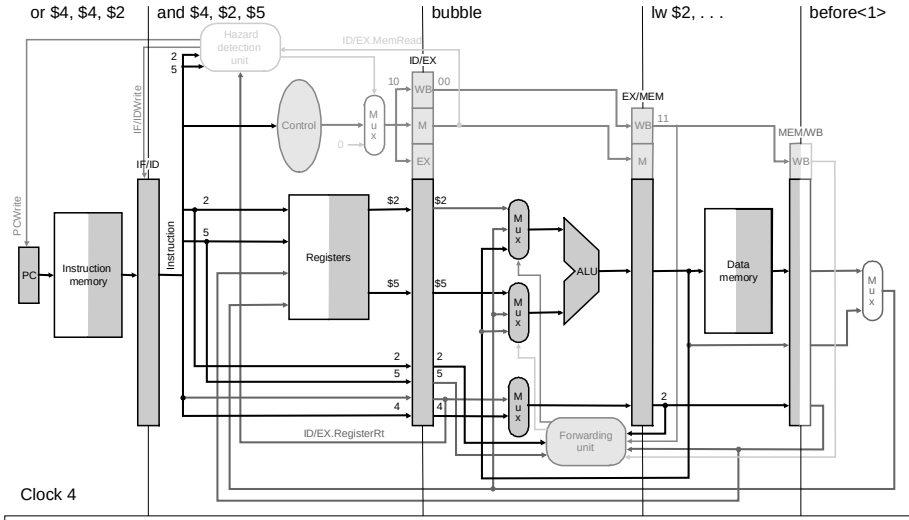
Clock 2

LUH example (2)



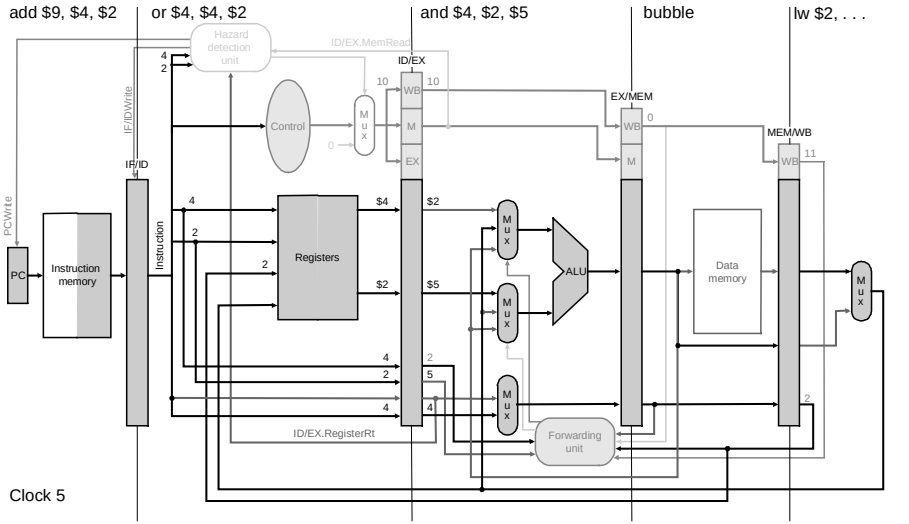
Clock 3

LUH example (3)



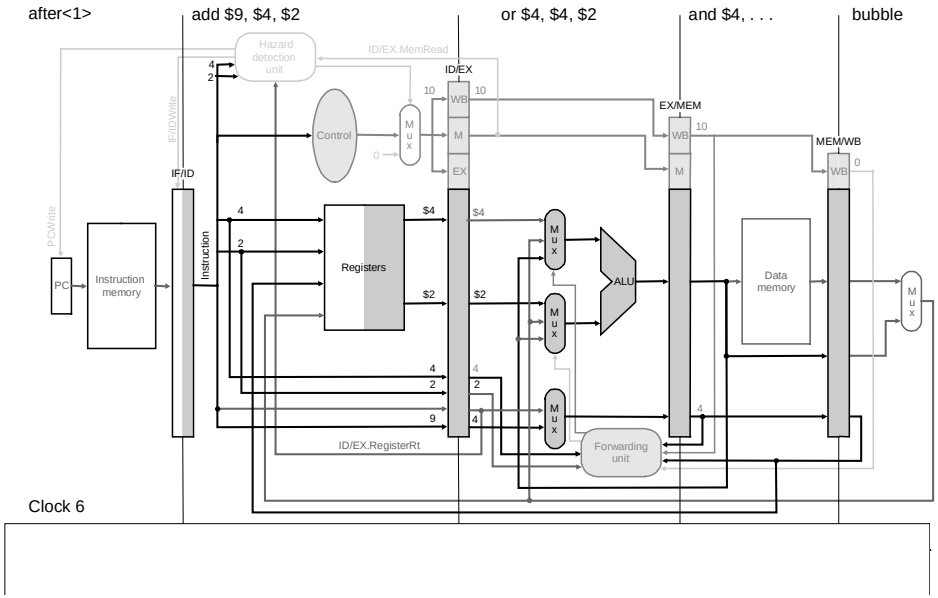
Clock 4

LUH example (4)

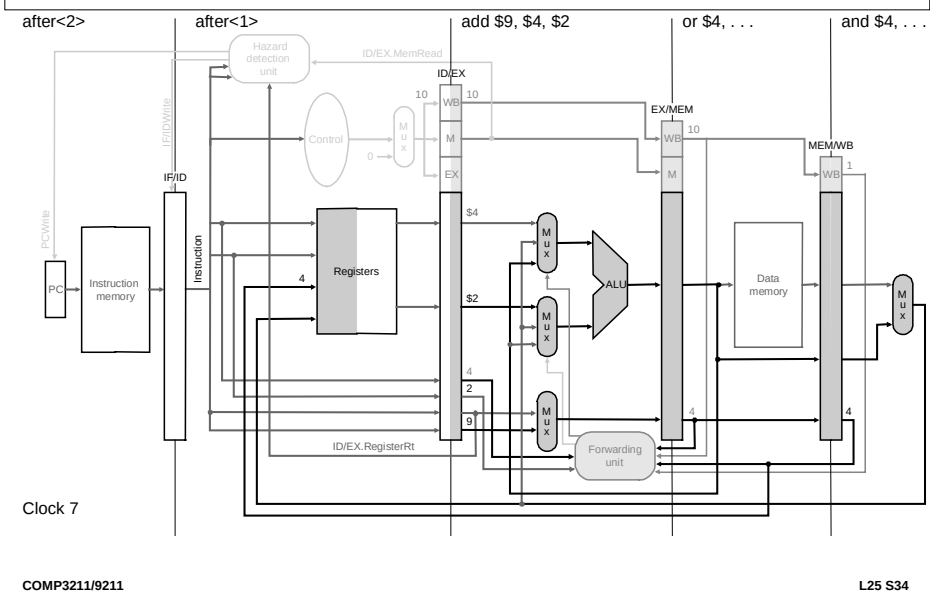


Clock 5

LUH example (5)



LUH example (6)



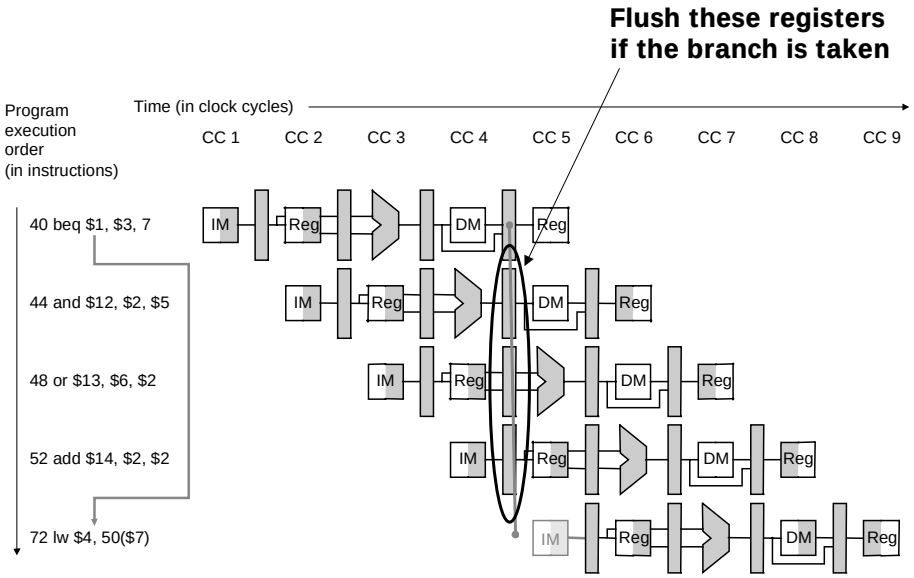
Control hazards

- Delay in determining the proper instruction to fetch is called a control hazard or branch hazard
- Solutions to control hazards
  - Stall
    - Wait until the branch decision is clear
      - Up to three cycles will be wasted for each branch depending upon the design of the datapath (availability of resources)
  - Branch prediction
    - static
    - dynamic
  - Branch delay

Static prediction

- Predict the branch always displays the same behavior
  - Never taken
  - Always taken
- When the prediction is wrong, the instructions being fetched, decoded, and executed after the branch instruction must be discarded (flushed) – here we assume the result of the branch is computed during the EX stage and the PC is updated during the MEM stage
- When the prediction is correct, the pipeline continues processing
- Better than stall
  - If the accuracy of the prediction is x, then there is 3/(3-2x) improvement in branch performance

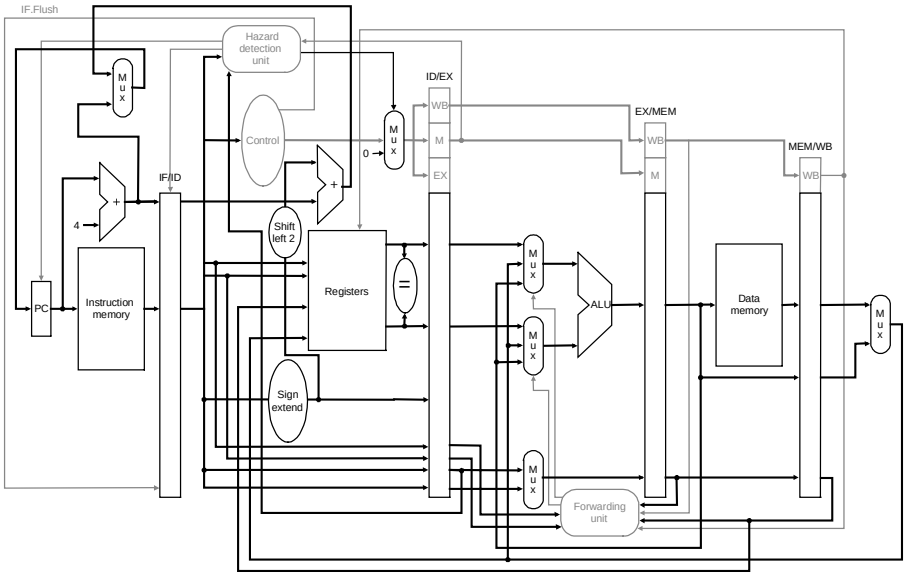
Static prediction – Branch not taken



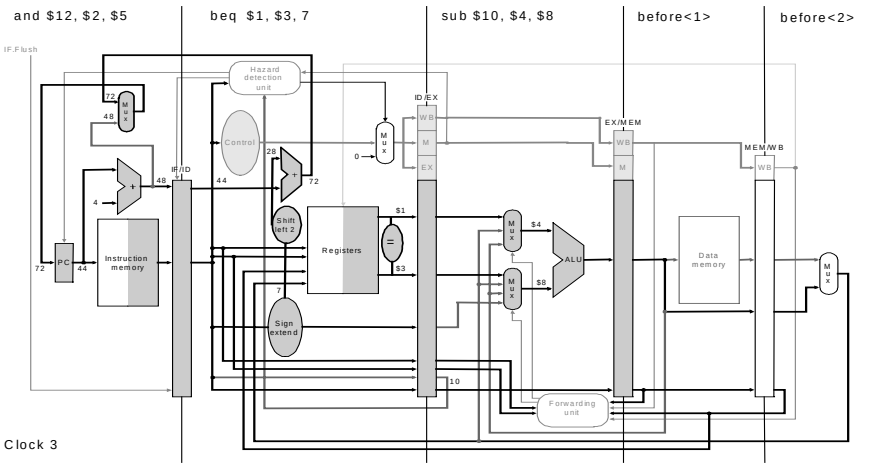
Static prediction – Branch not taken (cont.)

- Reducing the delay of branches – flushing 3 instructions is a heavy penalty!
  - Make branch decision earlier
    - Calculate the target address in the ID stage
    - Compare register contents in the ID stage
      - OR the results of parallel XOR tests on each bit (FAST, since no carry logic required)
  - Require forwarding to the register file and hazard detection if the branch is dependent upon results of an R-type or LOAD instruction still in the pipeline
  - Only one instruction then need be flushed
  - The control signal IF.Flush is used to flush the instruction in the IF/ID register
    - Sets the instruction field of IF/ID register to 0 (nop)

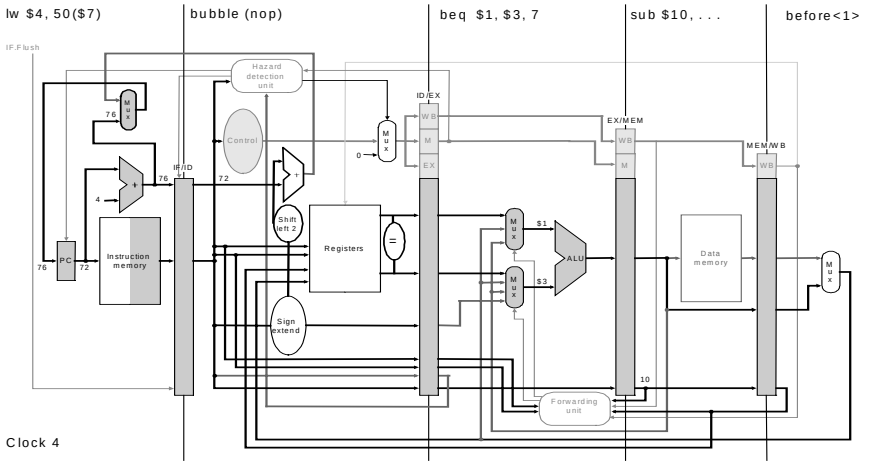
Static prediction – Branch not taken (cont.)



Handling misprediction(1)



Handling misprediction(2)



Dynamic prediction

- The prediction is made on the fly, depending on the history of the branch behavior
- The history is stored in a table or buffer, called the branch history table or branch prediction buffer
- The table consists of a portion of the branch instruction address and a branch history bit field

1-bit prediction scheme

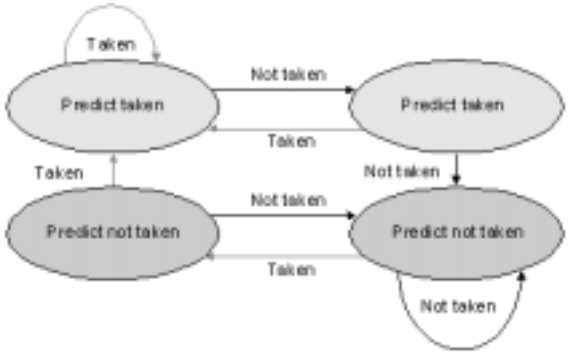
- The history bit field contains only 1 bit
- Prediction takes the bit value recorded from the last evaluation
  - If the prediction is wrong, the bit is changed
- Advantage
  - Simple
- Disadvantage
  - Double mis-prediction if branch is almost always taken:

    Consider loop that is executed several times
    - With 1 bit scheme, we always predict incorrectly when we exit the loop
    - But, when we execute loop again, we mispredict on the first iteration since the bit was set to “not taken” at the end of the last execution

    A 2-bit scheme is often used to overcome this problem

2-bit prediction

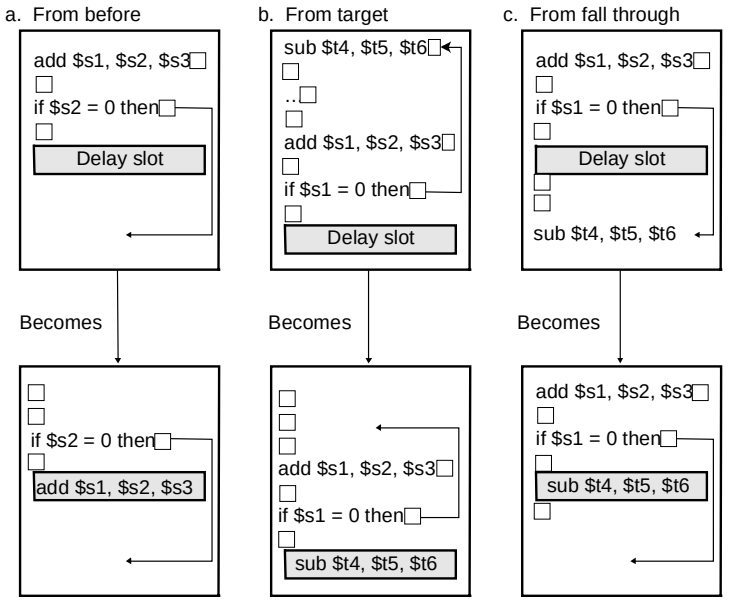
- History field has two bits
- A prediction must be wrong twice before it is changed



Delaying a branch

- A compiler approach
- Avoids the uncertainty of a branch decision by inserting an independent instruction (immediately after the branch instruction) that can always be executed irrespective of the branch decision (does not need to be flushed)
- Is becoming less popular as pipelines become longer and issue more instructions per cycle
- There is a preference for using the increasing availability of transistors to implement dynamic predictors instead

Scheduling a branch delay slot



What about Interrupts, Traps, Faults?

- External Interrupts:
  - Allow pipeline to drain,
  - Load PC with interrupt address
- Faults (within instruction, restartable)
  - Force trap instruction into IF
  - disable writes till trap hits WB
  - must save multiple PCs or PC + state

Refer to MIPS solution

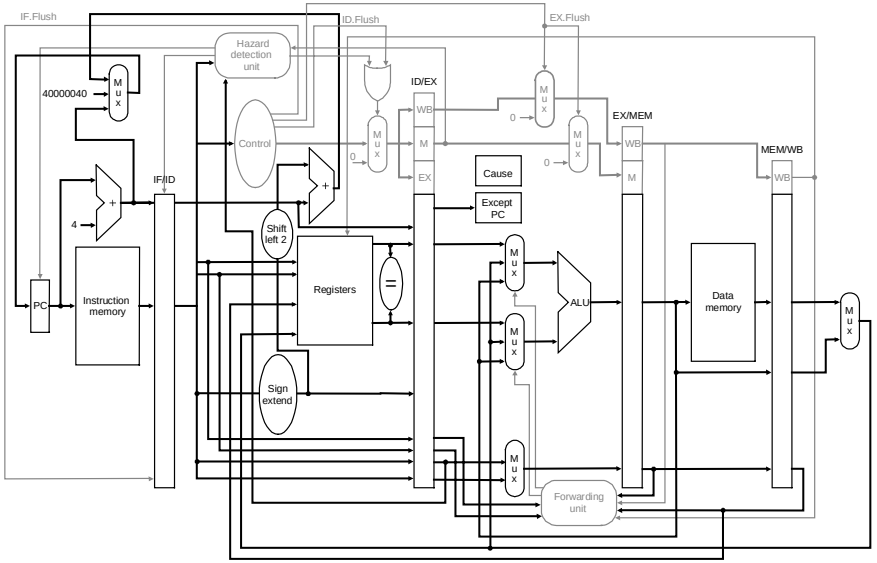
Exception Problem

Exceptions/Interrupts: 5 instructions executing in 5 stage pipeline

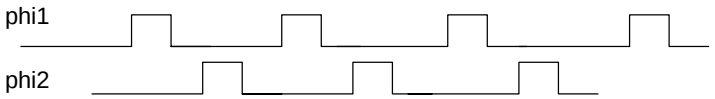
- How to stop the pipeline?
- Restart?
- Who caused the interrupt?

Stage	Problem interrupts occurring
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation; memory error

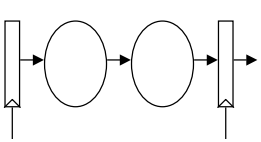
Datapath with controls for handling exceptions



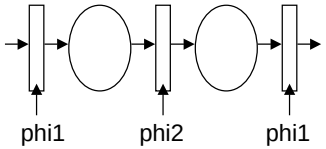
FYI: MIPS (R3000) clocking discipline



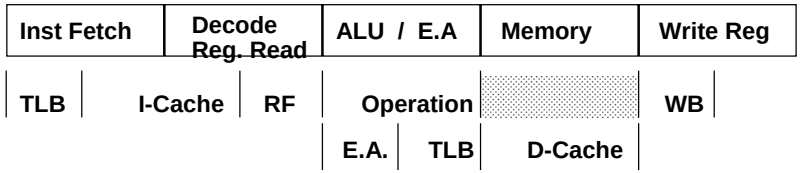
- **2-phase non-overlapping clocks**
- **Pipeline stage is two (level sensitive) latches**



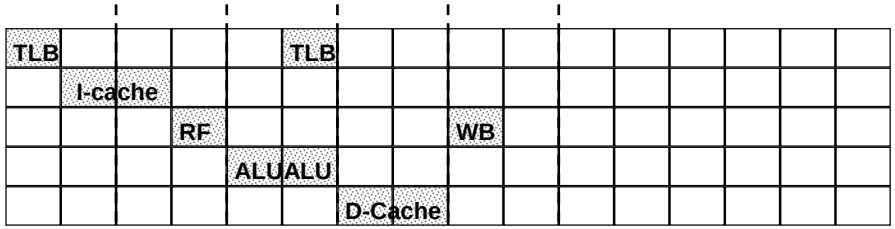
Edge-triggered



MIPS R3000 Instruction Pipeline

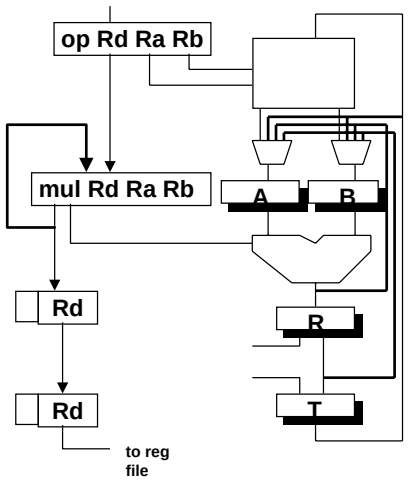


Resource Usage



Write in phase 1, read in phase 2 => eliminates bypass from WB

MIPS R3000 Multicycle Operations



Ex: Multiply, Divide, Cache Miss

Stall all stages above multicycle operation in the pipeline

Drain (bubble) stages below it

Use control word of local stage state to step through multicycle operation

Pipelining review

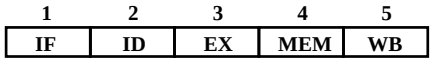
- What is pipelining?
  - allow multiple (m) instructions to be executed at the same time
- Pipelined datapath vs single cycle datapath
  - Similarities
    - CPI = 1 (idea pipelining case)
    - Control signals are generated once per instruction
  - Differences
    - Clock cycle time of the pipelined datapath is smaller
    - Control signals of the pipelined datapath need to be buffered
- Pipelined datapath vs multicycle datapath
  - Similarities
    - Same cycle time if same partitioning of logic
    - Multiple cycles per instruction
  - Differences
    - CPI (multi-cycle datapath) > CPI (pipelined datapath)

Pipelining review (cont.)

- Problems with pipelining
  - Resource competition
    - Structural hazards
  - Dependencies among instructions
    - Data hazards
    - Control hazards
- Solutions
  - For structural hazards
    - Resource replication
      - IMEM and DMEM
    - Resources always used in the same single stage
      - Resources for WB are only ever used in Stage 5

Pipelining review (cont.)

- Solutions
  - For data hazards



    - Hardware approaches
      - Forwarding
        - From EX/MEM (ALU output)
        - From MEM/WB (ALU output or MEM output)
      - Stall + forwarding
        - Load-read: stall read instruction one clock cycle then forwarding data from MEM/WB
    - Software approaches/enhancements
      - Make the two dependent instructions X number of instructions apart
        - X >=3, no data hazards
        - X = 2, no data hazards with proper register write and read control
        - X = 1, no data hazards with forwarding
        - X = 0, no data hazards for R-type instructions with forwarding; one cycle stall for load-read instructions

Pipelining review (cont.)

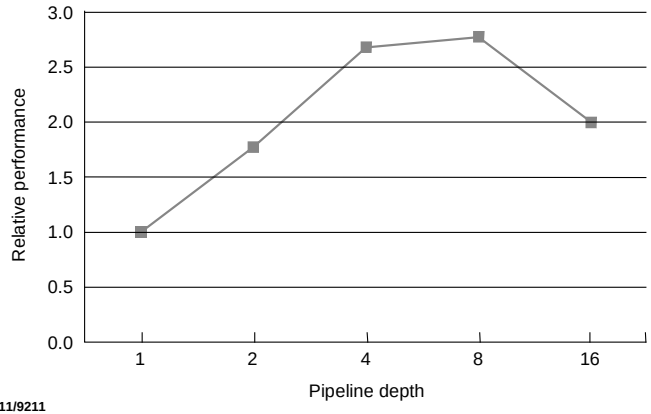
- Solutions
  - For control hazards
    - Hardware approaches
      - Prediction
        - Branch taken/not taken (static and dynamic)
        - Flush if wrong
      - Evaluate and calculate target address earlier
        - Done in ID stage
    - Software approaches
      - Delay branch

Beyond pipelining

- To achieve even greater performance, the simple pipeline concepts has been extended in 3 ways:
  1. Superpipelining
     - longer pipelines, e.g. 8 or more stages
  2. Superscalar
     - replicate datapath components so that multiple instructions can be launched each cycle
     - Allow instruction execution rate to exceed the clock rate, or for CPI < 1
  3. Dynamic pipelining (dynamic pipeline scheduling)
     - Avoids pipeline hazards by bringing forward instructions that otherwise wait on a dependency to be resolved
     - Requires a more sophisticated pipeline control and instruction execution model

Increasing pipeline depth does not always increase performance

- Three factors limit performance improvement gained by increasing pipelining depth
  1. Data hazards in the code increase the time per instruction because a larger percentage of cycles become stalls
  2. Control hazards result in slower branches
  3. Pipeline register overhead limits the decrease in clock period

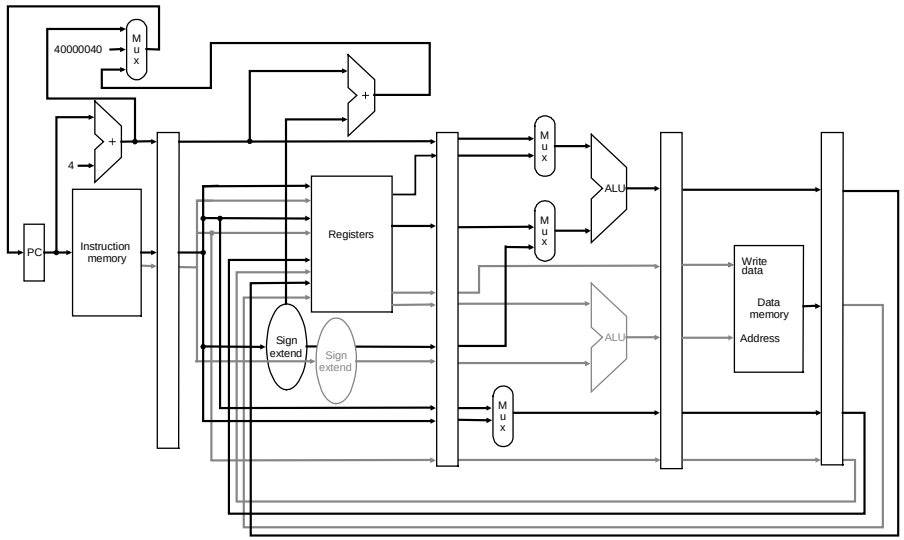


What does a superscalar implementation look like?

- Assume 2 instructions issued per clock cycle
  - One an integer ALU op or branch, the other a load or store
  - Hence, fetch 64 bits of instruction per cycle & assume they are aligned such that the ALU op always appears first
  - Only allow the second instruction to be issued if the first one can be

Instruction Type	Pipe Stages						
ALU/branch	IF	ID	EX	MEM	WB		
Load/store	IF	ID	EX	MEM	WB		
ALU/branch		IF	ID	EX	MEM	WB	
Load/store		IF	ID	EX	MEM	WB	
ALU/branch			IF	ID	EX	MEM	WB
Load/store			IF	ID	EX	MEM	WB

A superscalar datapath



Superscalar datapath hazards

- In order to avoid structural hazards,
  - Need extra read & write ports on the register file to allow ALU and data transfer ops to be issued in parallel
  - Need an additional adder to calculate memory addresses for data transfers
- Data hazards arising from loads cause a stall of two instructions
  - Need to employ more ambitious compiler & hardware scheduling techniques

Simple superscalar code scheduling

- The following loop needs to be reordered to make use of the superscalar resources:

```
Loop: lw    $t0,0($s1)  # $t0=array element
      addu  $t0,$t0,$s2  # add scalar in $s2
      sw    $t0,0($s1)   # store result
      addi  $s1,$s1,-4   # decrement pointer
      bne   $s1,$0,Loop  # branch $s1!=0
```

- The reordered sequence is:

CC		ALU/branch inst		Data transfer	
1	Loop:			lw	$t0,0($s1)
2		addi	$s1,$s1,-4		
3		addu	$t0,$t0,$s2		
4		bne	$s1,$0,Loop	sw	$t0,4($s1)

- Note just one pair of instructions executes in superscalar mode – it takes 4 cycles for 5 instructions – CPI = 0.8 > 0.5!

Use loop unrolling to extract better performance

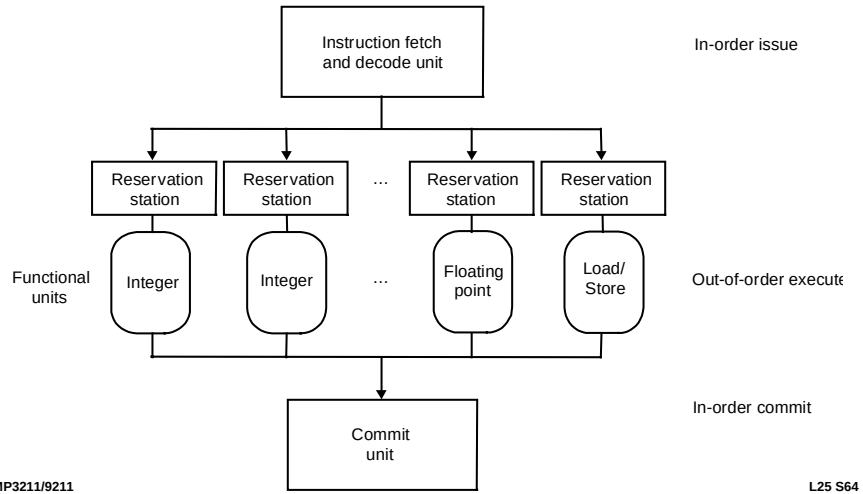
- Assume the loop index is a multiple of 4, then we can unrol as follows:

CC		ALU/Branch inst		Data transfer	
1	Loop:	addi	$s1,$s1,-16	lw	$t0,0($s1)
2				lw	$t1,12($s1)
3		addu	$t0,$t0,$s2	lw	$t2,8($s1)
4		addu	$t1,$t1,$s2	lw	$t3,4($s1)
5		addu	$t2,$t2,$s2	sw	$t0,16($s1)
6		addu	$t3,$t3,$s2	sw	$t1,12($s1)
7				sw	$t2,8($s1)
8		bne	$s1,$0,Loop	sw	$t3,4($s1)

- Notice that 12 of the 14 instructions are executed in superscalar mode and we need 8 cycles for 4 loop iterations – a factor of 2 improvement due to reducing the loop control instructions and from increased superscalar execution

Dynamic pipelining

- Dynamic pipeline scheduling goes past stalls to find later instructions to execute while waiting for the stall to be resolved
- Typically, the pipeline is divided into an instruction fetch and issue unit, several (5-10) execute units, and a commit unit



Dynamic pipeline execution

- The instruction fetch & issue unit fetches & decodes instructions before sending them to a corresponding functional unit in the execute stage
- Each functional unit has buffers, called reservation stations, that hold the operation and operands until all operands are available and the unit is available; then the result is calculated
- The commit unit decides when it is safe to put the result into the register file or memory (for store)
- When the fetch and commit units issue instructions and write results in the original program order, the pipeline is said to operate in a conservative *in-order completion* mode – the alternative, liberal, *out-of-order completion* mode may be more effective at overcoming stalls
- Dynamic scheduling is often combined with branch prediction, which complicates the design of the commit unit because it must discard all the results of a mispredicted branch.
- Combining dynamic scheduling and branch prediction is called *speculative execution*

Motivation for dynamic pipelining

1. Hide memory latency
2. Avoid stalls that the compiler could not schedule
3. Speculatively execute instructions while waiting for hazards to be resolved

Exercise

- How is the following set of instructions (don't worry about the quality of the code) executed in our five-stage pipeline? How can we improve its performance?

```
Loop:   lw      $t0, 0($t1)
        add     $t3, $t0, $t1
        sub     $t4, $t0, $t2
        or      $t5, $t3, $t4
        sw      $t1, 0($t1)
        beq     $t1, $t0, loop
Exit:   add     $t3, $t4, $t5
```