COMP3710 (Class # 5176)
Special Topics in Computer Science
Computer Microarchitecture

Convener: Shoaib Akram
shoaib.akram@anu.edu.au

Australian
National
University

Plan

Week 4: Data and branch hazards, branch prediction

Week 4: Correlating predictors (via an example)

Week 5: Hybrid, Neural, and Tag-based predictors

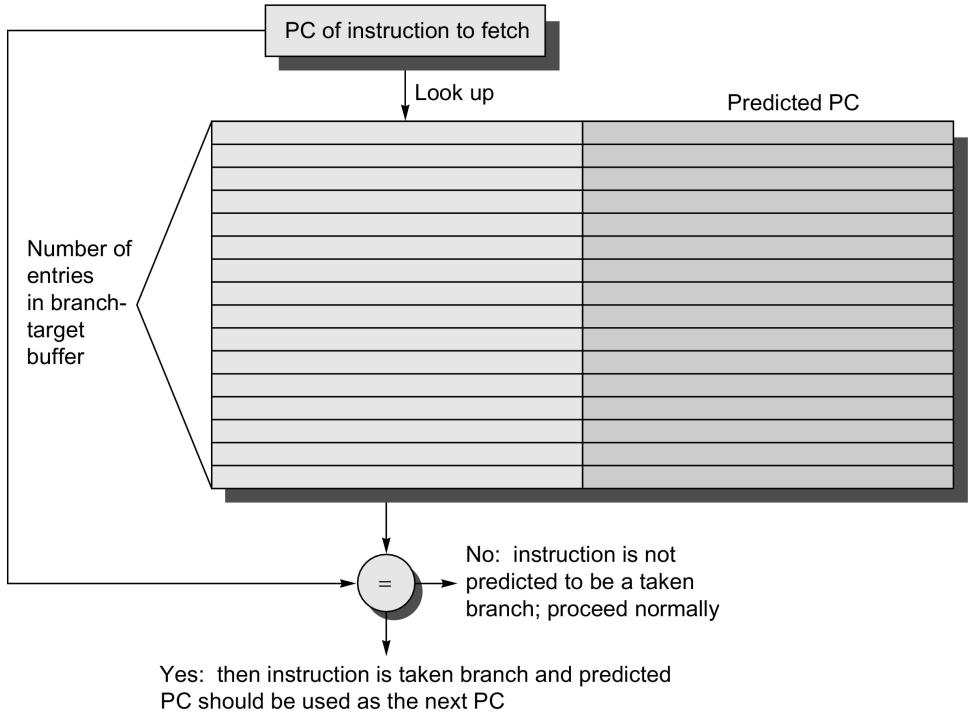
Week 5: BTBs, Exception handling, Multiscalar Pipelines

Week 5: Move towards Out-of-Order

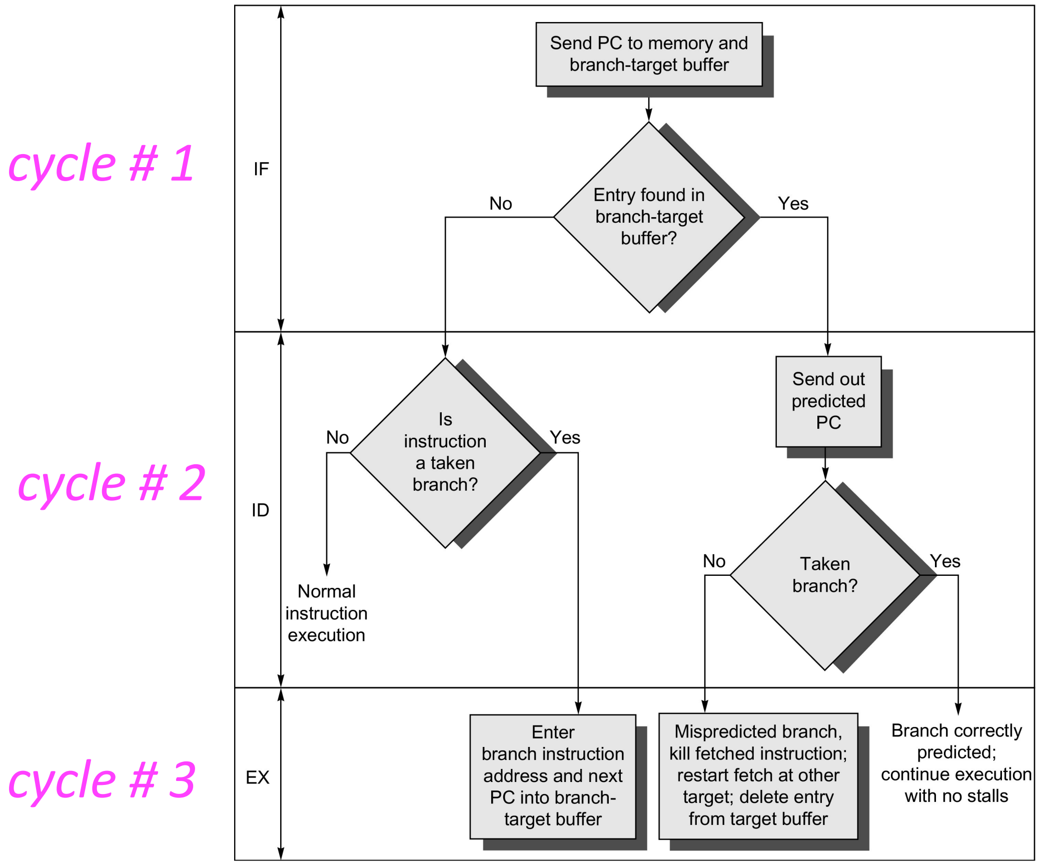
Brach Target Buffer (BTB)

- We need for conditional branches (in the fetch stage)
  - Direction prediction
  - Target address prediction
- Target address
  - Not taken branches: `PC + sizeof(instruction)`
  - Taken: Depends on the branch
    - Two types of branch target: PC-relative and indirect (register + constant)
  - Indirect branches are frequent in OOP, C++ vtable implementation, case statements, and dynamically linked libraries (Section 2.7 and 2.12 of PH1)
  - Must also consider unconditional branches (always taken)
- BTB (also called branch target address cache or BTAC) stores the last seen target address for a branch instruction
  - Taken + hit in BTB → Fetch from predicted target
  - Taken + miss in BTB → Different policies (stall until resolved, non-taken target)

Brach Target Buffer (BTB)



PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be a taken branch; proceed normally
Yes: then instruction is taken branch and predicted PC should be used as the next PC

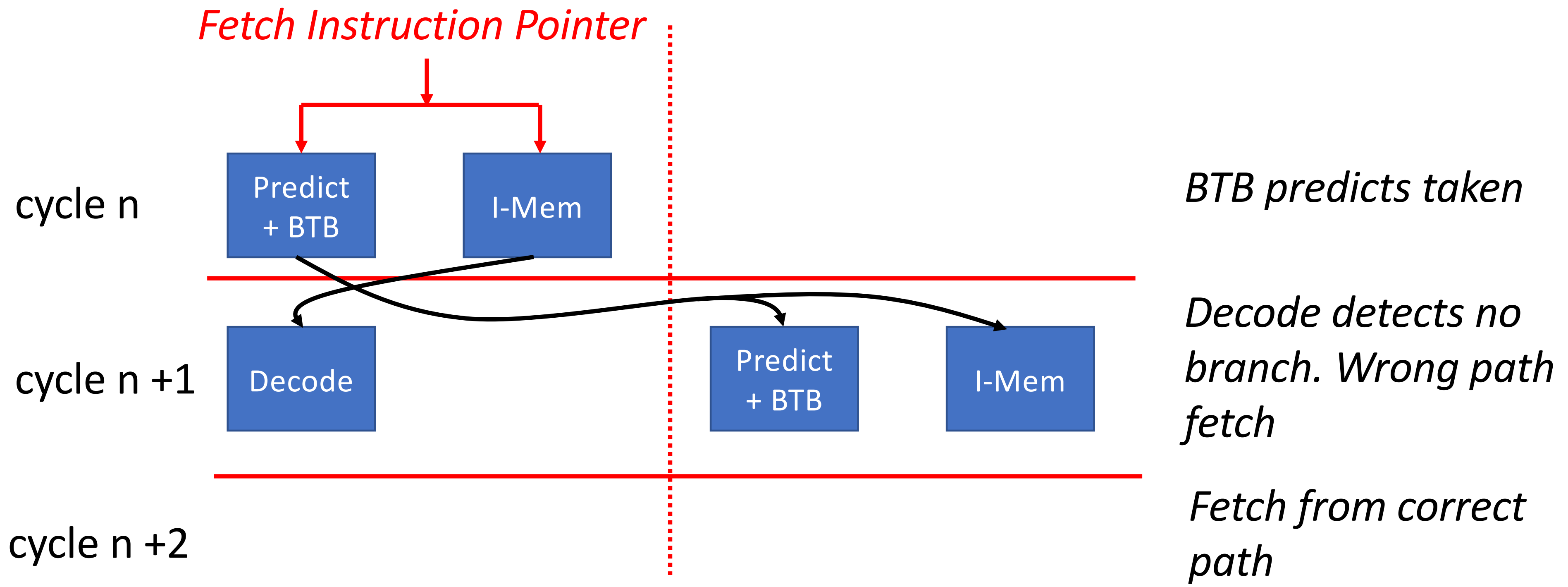


cycle # 1
cycle # 2
cycle # 3
IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

Phantom Branches

A predicted-taken branch that has no corresponding branch instruction

- At fetch time, the BTB can make mistakes (aliasing)
- Typically, the decode logic detects there is no branch and redirect the fetch in the right direction



Fetch Instruction Pointer
cycle n
Predict + BTB
I-Mem
cycle n +1
Decode
Predict + BTB
I-Mem
cycle n +2
BTB predicts taken
Decode detects no branch. Wrong path fetch
Fetch from correct path

Return Address Stack (RAS)

Function call

- Jump into a function
- Return from a function
- Return target harder to detect (e.g., always jump from the same point to *printf()* but *printf() can be called from many program locations*)

ISA support

- Store the return address in a register
- Push it on to the stack (programmer)
- Return instruction (explicit *return instruction* or jump register)

RAS is a branch target predictor that provide target addresses for function returns

Return Address Stack (RAS)

RAS operation

- On a function call, push the return address on top of the RAS
- Pop the entry on a function return and use it as a prediction
- Multiple entries in the RAS to support nested functions

How do we know if an instruction is a return in the fetch stage?

- We do not. BTB provides an intitial target prediction for both the jump into a function the return from a function
- After the instruction is decoded, RAS provides the target (typically after decode)
- Without the RAS, target misprediction is not detected until the return address is loaded from program stack into a register and the return instruction is executed

Return Address Stack (RAS)

Question: Why do we need RAS if we have the return address on the method stack?

- We do not want to wait until the return address is popped from the stack and the return instruction (or the jump and link instruction) has executed
- In multi-issue processors, pop and link instructions may be fetched in the same issue packet
  - So to start fetching as soon as we can, we use an RAS
- Furthermore, we use an RAS in the decode stage (or after) so we can detect at least the branch instruction is a return
- Meanwhile in the fetch stage the BTB can give an initial prediction for the target address

Exceptions

Exceptions (or interrupts) are a form of control flow hazard

- Disrupt the normal control flow due to an unexpected event
- User to kernel swicth, I/O request, arithmetic overflow, undefined instruction, malfunction

Need to do two things

- Save the address of the offending instruction in an Exception Program Counter (EPC)
- Transfer control to the OS at some specified address
- Restart execution or terminate

Exception Handling

Two techniques

- Vectored interrupts
- Non-vectored interrupts (MIPS)

Vectored Interrupts

- The interrupting device provides enough information to switch control to the correct target address
- The OS knows the reason for the interrupt when invoked

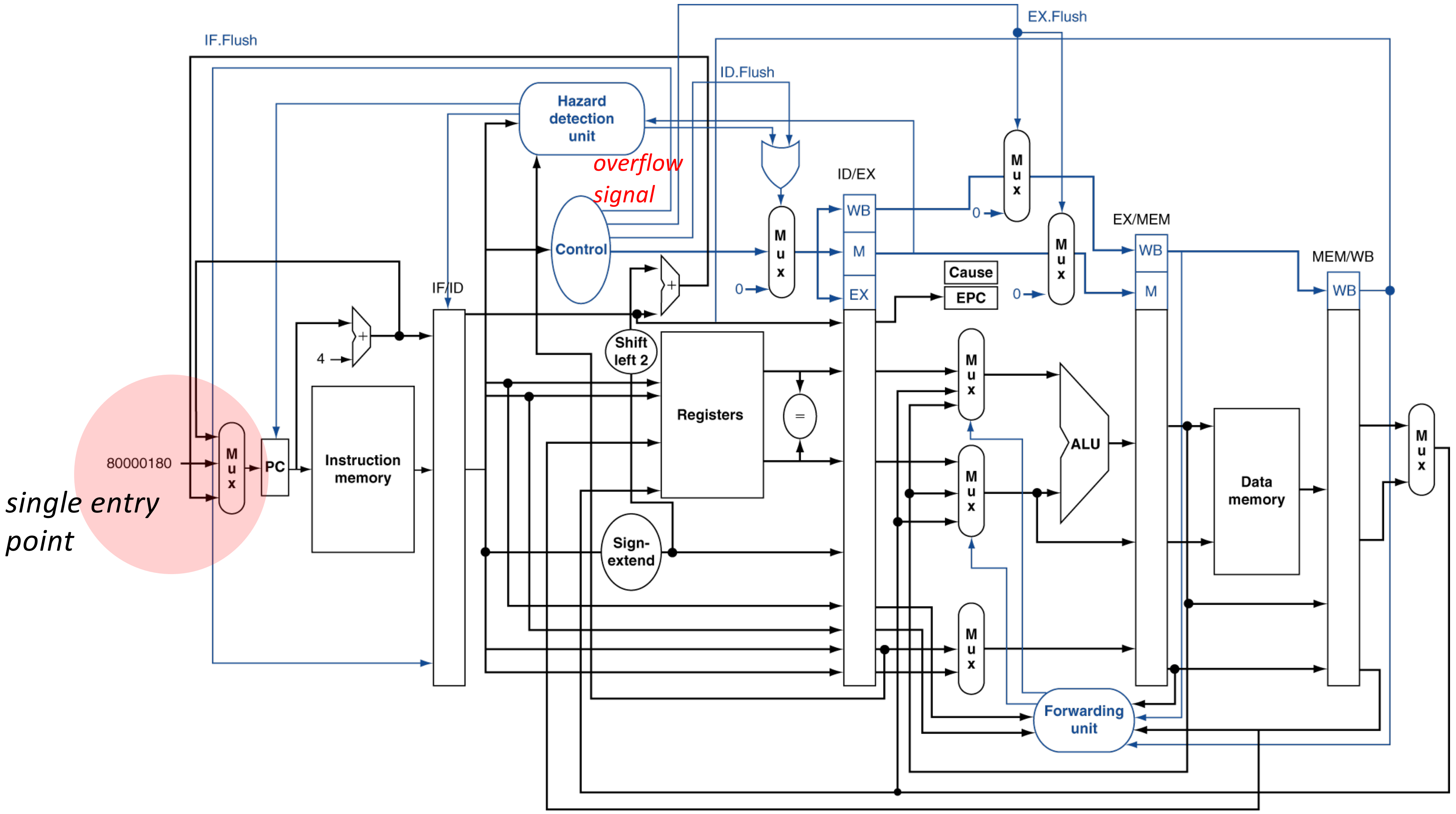
Non-Vectored Interrupts

- Single point of entry regardless of the exception type
- Note down the reason for the interrupt in a special cause register

Two additions to the pipeline

- 32-bit EPC register (vectored or non-vectored)
- 32-bit cause register (several bits unused)

Pipeline with Exceptions



Example

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

overflow exception!

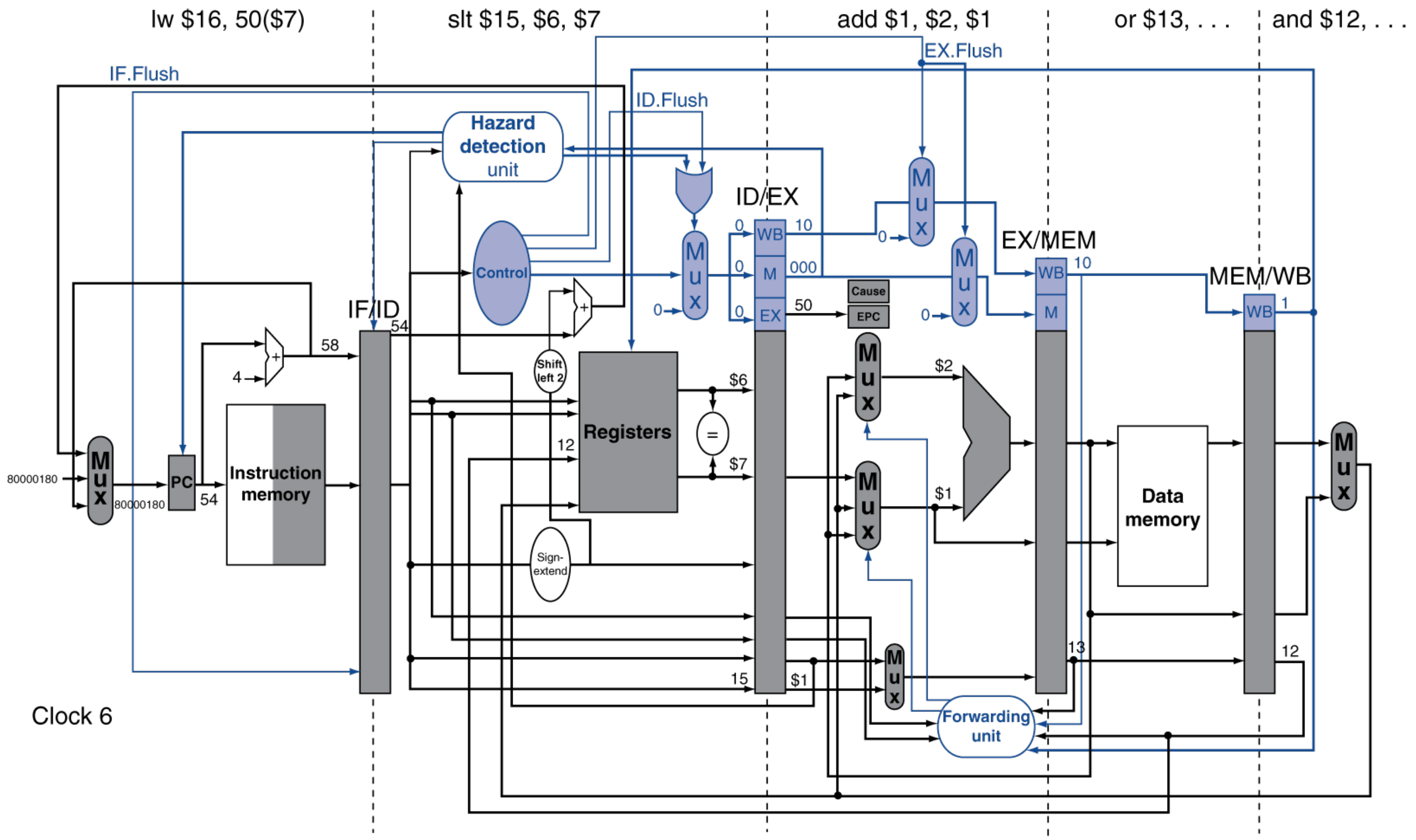
```
80000180hex   sw     $26, 1000($0)
80000184hex   sw     $27, 1004($0)
. . .
```

instructions invoked on an exception

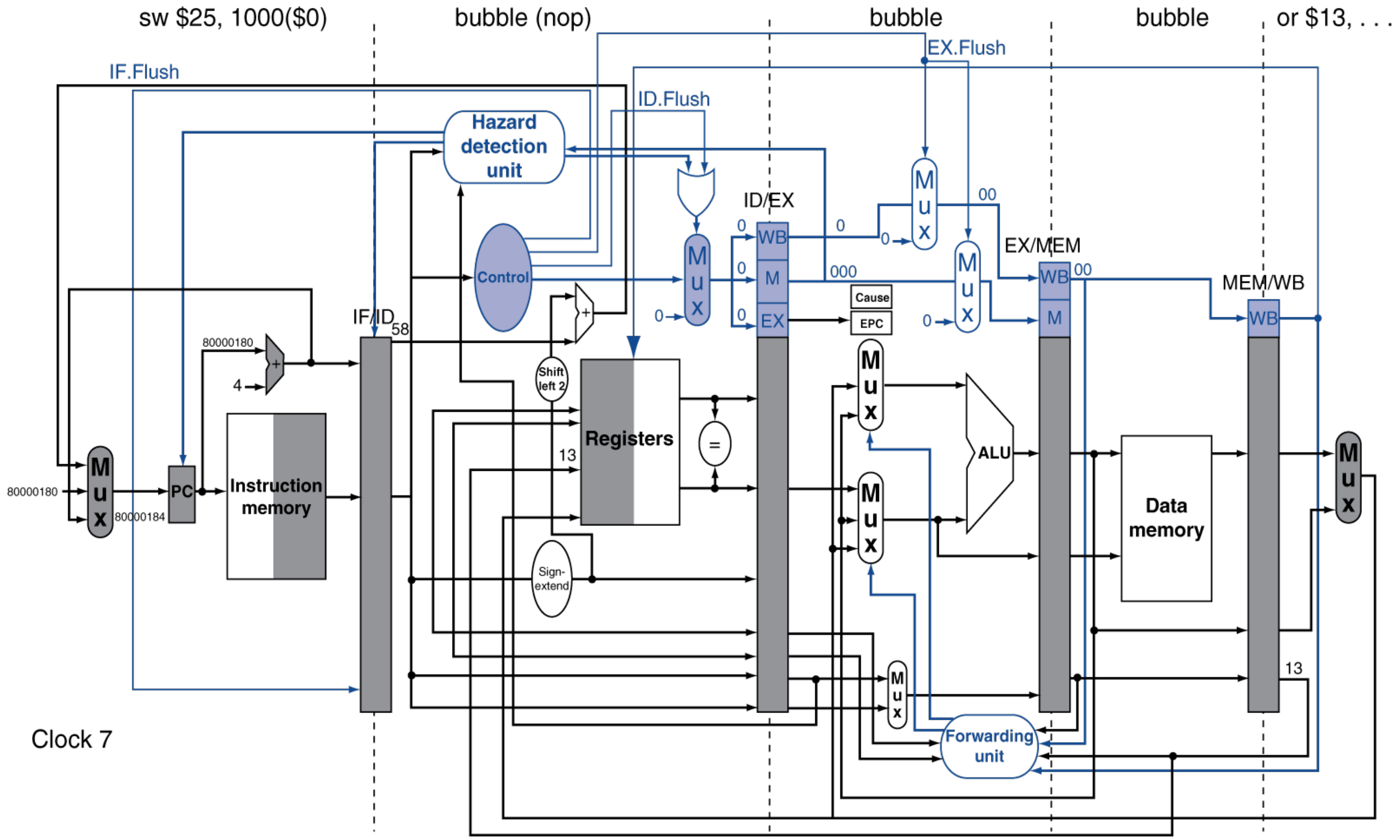
Key Ideas to Handle Overflow Exception

- Execution must be stopped in the middle of the instruction
  - We must preserve the value of $1 to make the exception precise
- At then end of the cycle in which the overflow is detected all Flush signals must be asserted
  - Turn `add` into a `nop`
- Fetch the first instruction for the exception routine
- All instructions prior to `add` still complete
- The ALU overflow signal is an input to the control unit

Exceptions Example



Exceptions Example



Instruction-Level Parallelism (ILP)

- Pipelining exploits the parallelism among instructions

Two ways to increase parallelism

- Increase the depth of the pipeline to overlap more instructions
  - Shorter clock cycle would potentially lead to greater performance
- Launch multiple instructions in every pipeline stage
  - Multiple-issue pipelines
  - Need replication of components to launch multiple instructions in a single clock cycle
  - CPI < 1 and IPC > 1

Multiple-Issue Pipelines

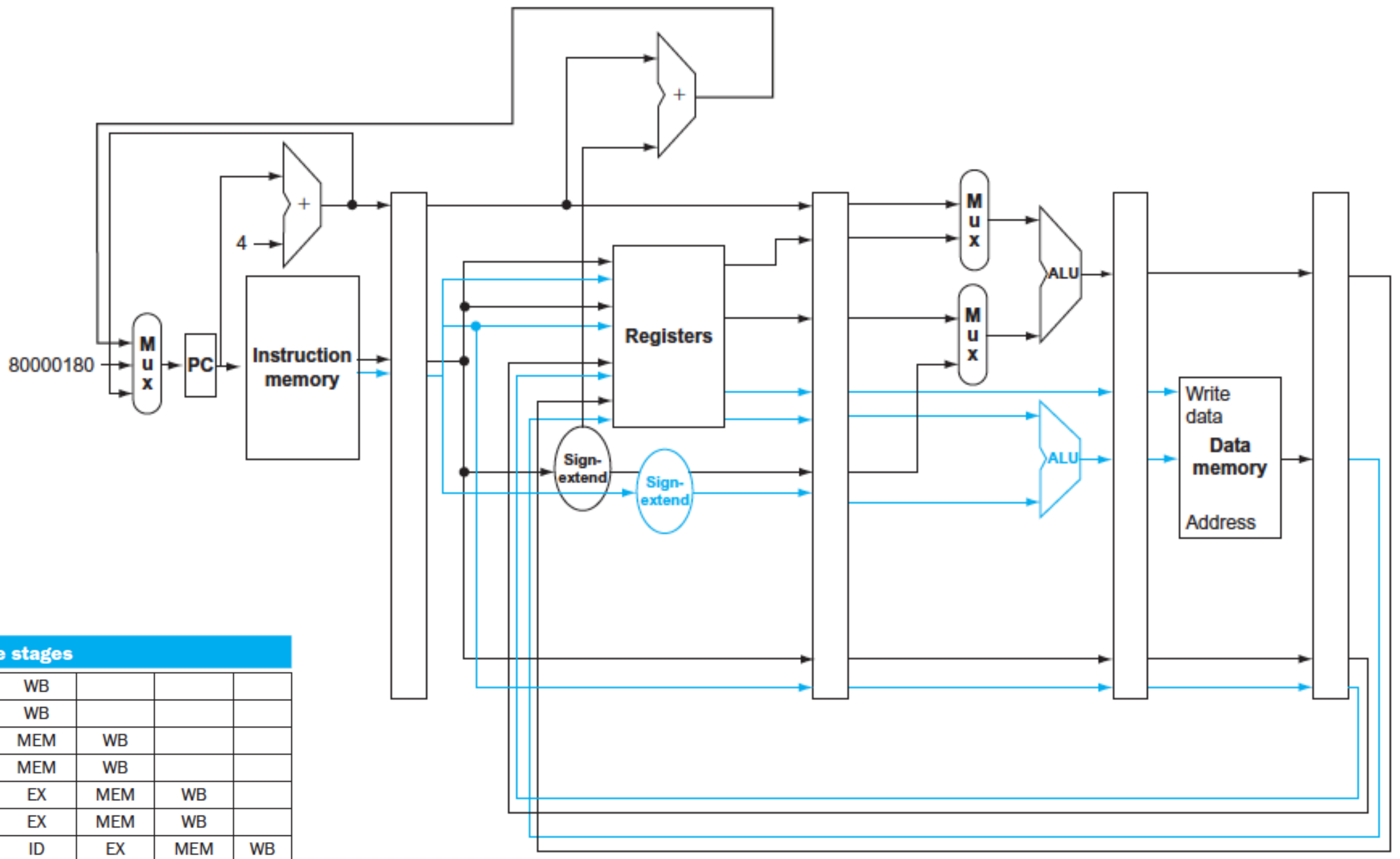
Two ways to issue multiple instruction in a cycle

- Compile-time scheduling (e.g., very long instruction word or VLIW)
    - Statically pick set of instructions that issue together (called issue packet)
- Dynamic scheduling (superscalar)

Static instruction scheduling

- Need to limit the co-executing instruction pairs (e.g., 1 ALU + 1 load)
- **Question:** What changes do we need to the MIPS pipeline to support dual-issue?
- **Question:** How best to fill the issue packet (i.e., instruction pairs)?
- **Question:** Who detects hazards and inserts stalls?
    - Compiler or hardware

Statically Scheduled Dual-Issue Pipeline



Example Scheduling

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Static Multiple Issue + Loop Unrolling

Static multiple issue

- Compiler packs instructions into a single long instruction word
- Hardware fetches/executes *issue packets*: 2+ instructions
- **Very Long Instruction Word (VLIW):** 4 or more instructions
- Compiler picks the instruction mix

addi $1, $1, -4	**lw $0, 0($1)**

- Compiler (typically) handles data/branch hazards
- Hardware places constraints on the mix, e.g., ALU + Load

Loop unrolling

- Compilation technique for exploiting instruction level parallelism
- Make multiple copies of the loop body
- Hardware schedules instructions from different iterations

Example

Schedule this loop on a static two-issue pipeline for MIPS

```
Loop:  lw    $t0   0($s1)
       addu  $t0   $t0     $s2
       sw    $t0   0($s1)
       addi  $s1   $s1     -4
       bne   $s1   $zero   Loop
```

reordering $\Rightarrow$

```
Loop:  lw    $t0   0($s1)
       addi  $s1   $s1     -4
       addu  $t0   $t0     $s2
       bne   $s1   $zero   Loop
       sw    $t0   4($s1)
```

separate the dependent instructions addi *and* bne

Change the constant

	ALU or Branch Instruction	Load/Store Instruction	Clock Cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1, -4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

IPC (ideal) = 2
IPC (real) = 1.25
Inefficient scheduling
→ Three empty slots
→ Not enough ILP
→ ILP is found across large instruction windows

Example

Schedule this loop on a static two-issue pipeline for MIPS

```
Loop:   addi    $s1     $s1     -8
        lw      $t0     8($s1)
        addu    $t0     $t0     $s2
        sw      $t0     8($s1)
        lw      $t1     4($s1)
        addu    $t1     $t1     $s2
        sw      $t1     4($s1)
        bne     $s1     $zero   Loop
```

- **Register renaming:** Use $t1 in addition to $t0 to avoid WAR and WAW anti-dependences.
- **Unrolling overhead:** Subtract **8** at the beginning and use constants to reduce unrolling overhead

	ALU or Branch Instruction	Load/Store Instruction	Clock Cycle
Loop:	addi $s1, $s1, -8	lw $t0, 0($s1)	1
	nop	lw $t1, 4($s1)	2
	addu $t0, $t0, $s2	nop	3
	addu $t1, $t1, $s2	sw $t0, 8($s1)	4
	sw $t1, 4($s1)	bne $s1, $zero, Loop	5

IPC (ideal) = 2
IPC (real) = 1.6
Inefficient scheduling
→ Three empty slots
→ Not enough ILP
→ ILP is found across large instruction windows

Example

Schedule this loop on a static two-issue pipeline for MIPS

```
Loop:   addi    $s1     $s1       -16
        lw      $t0     16($s1)
        addu    $t0     $t0       $s2
        sw      $t0     16($s1)
        lw      $t1     12($s1)
        addu    $t1     $t1       $s2
        sw      $t1     12($s1)
        lw      $t2     8($s1)
        addu    $t2     $t2       $s2
        sw      $t2     8($s1)
        lw      $t3     4($s1)
        addu    $t3     $t3       $s2
        sw      $t3     4($s1)
        bne     $s1     $zero     Loop
```

	ALU or Branch Instruction	Load/Store Instruction	Clock Cycle
Loop:	addi $s1, $s1, -16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t3, $s2	sw $t1, 12($s1)	6
		sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

IPC (ideal) = 2
IPC (real) = 14/8 = 1.75

From In Order to Out of Order

A critical limitation of the 5-stage MIPS pipeline is the *blocking execute stage*

- Single universal unpipelined ALU
- Execute stage blocks for multi-cycle operations
- Cache misses block the execute stage

structural hazards

The blocking execute stage hides a critical difference between the in order and out-of-order issue policy

Structural hazards supersede data hazards

- How RAW/WAR/WAW hazards are handled is not obvious

From In Order to Out of Order

Towards a more aggressive in-order scalar pipeline

- Non-blocking execute stage (eliminate structural hazards)
- State the nature of *in-order issue policy*

In-order issue policy

- Younger instruction has a RAW hazard with an older instruction (must stall and it's ok!)
- What about instructions after it? Some of the younger instructions may be independent (*this is where the problem lies*)

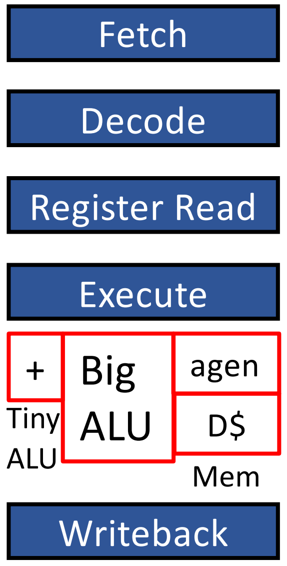
From In Order to Out of Order

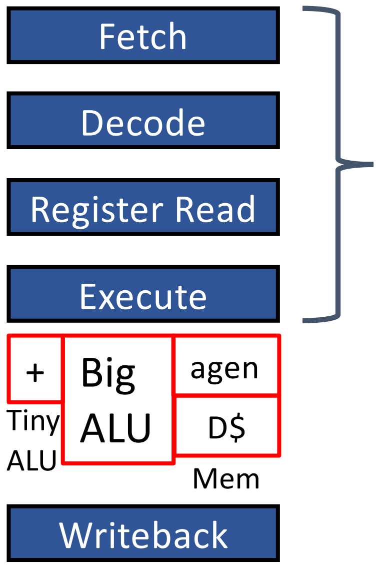
Out of order pipeline

- An instruction stalls if it has a RAW hazard with a previous instruction (that's ok)
- Independent instructions after it do not stall: they may issue out of program order

Two alternatives for handling WAR and WAW

- Stall the pipeline (in-order-style)
- Register renaming (optional optimization)

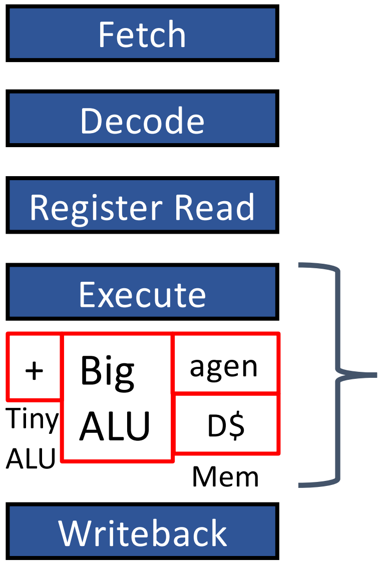

Fetch
Decode
Register Read
Execute
+
Tiny ALU
Big ALU
agen
D$
Mem
Writeback



Assumptions

Scalar:

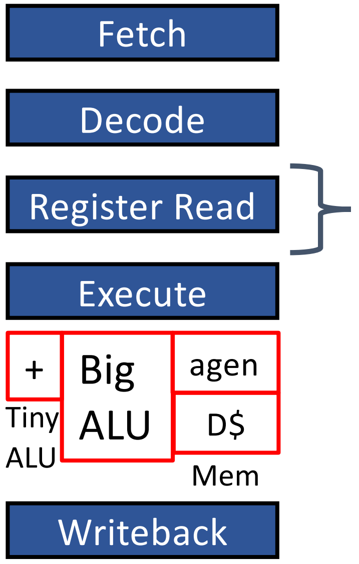
- *fetch 1 inst/cycle*
- *decode 1 inst/cycle*
- *issue 1 inst/cycle to a function unit*



Assumptions

Execute stage:

- *Contains multiple functional units (FUs) to support different instruction classes*
- *Multi-cycle function units are pipelined (FP mul, MAC)*
- *May observe multiple instructions executing concurrently, yet only 1 new instruction may begin executing in a cycle (scalar issue)*



Assumptions

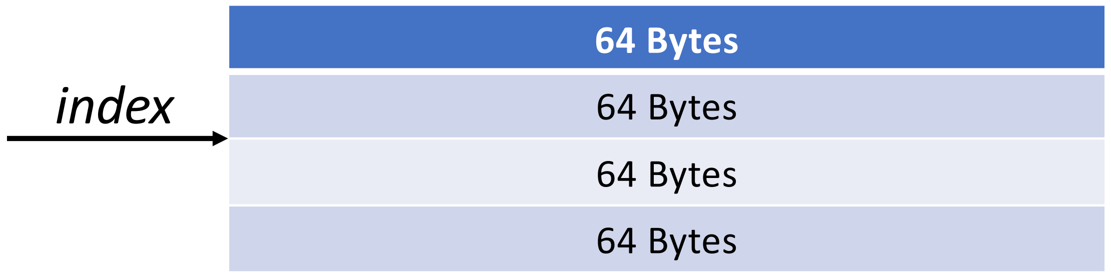
Issue logic:

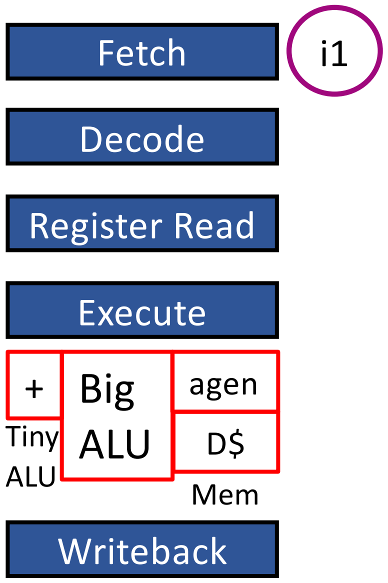
- *RAW hazard: Instruction stalls if its source registers are not ready*
- *WAW hazard: Non-blocking execute stage plus variable FU latencies introduce out-of-order writeback. Ok if writes to different registers. Not Ok if writes are to the same register.*
- *Instruction stalls if its destination register is "busy", i.e., conflicts with destination register of older instruction in Execute stage*
- *WAR hazard: Not a problem in in-order pipelines. In-order issue ensures read by first instruction happens before write by second instruction*

Scenario 1: load miss followed by independent instructions

Cache Memory 101

- Spatial Locality: *If you access a memory location, likely to access a nearby location in the near future*
- Temporal Locality: *If you access a memory location, likely to access it again in the near future*





Scenario 1: load miss followed by independent instructions

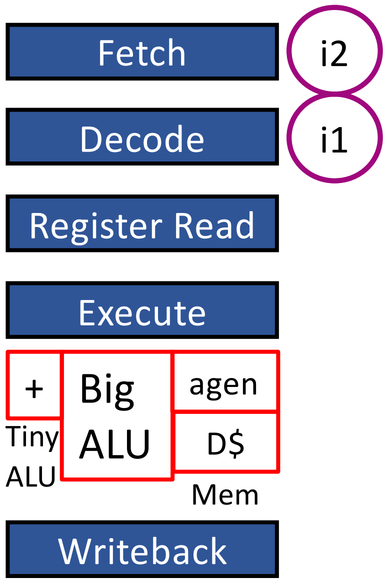
i1: load r2, #0(r1)

i2: add r4, r3, #1

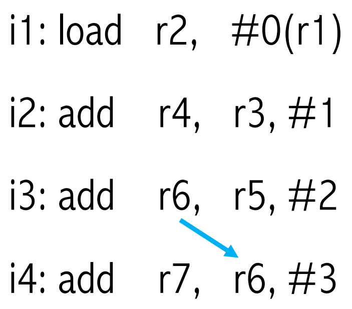
i3: add r6, r5, #2

i4: add r7, r6, #3

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE												
i2													
i3													
i4													

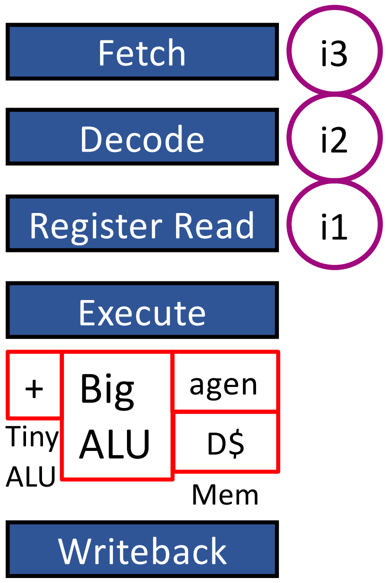


Scenario 1: load miss followed by independent instructions



```
i1: load   r2,   #0(r1)
i2: add    r4,   r3, #1
i3: add    r6,   r5, #2
i4: add    r7,   r6, #3
```

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE											
i2		FE											
i3													
i4													



Scenario 1: load miss followed by independent instructions

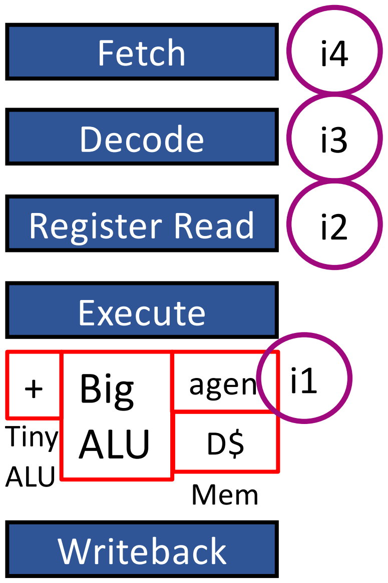
i1: load r2, #0(r1)

i2: add r4, r3, #1

i3: add r6, r5, #2

i4: add r7, r6, #3

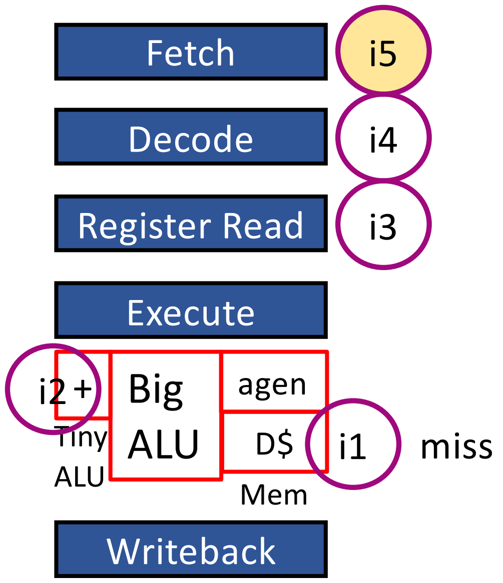
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR										
i2		FE	DE										
i3			FE										
i4													



Scenario 1: load miss followed by independent instructions

```
i1: load  r2,  #0(r1)
i2: add   r4,  r3, #1
i3: add   r6,  r5, #2
i4: add   r7,  r6, #3
```

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_@$									
i2		FE	DE	RR									
i3			FE	DE									
i4				FE									



Scenario 1: load miss followed by independent instructions

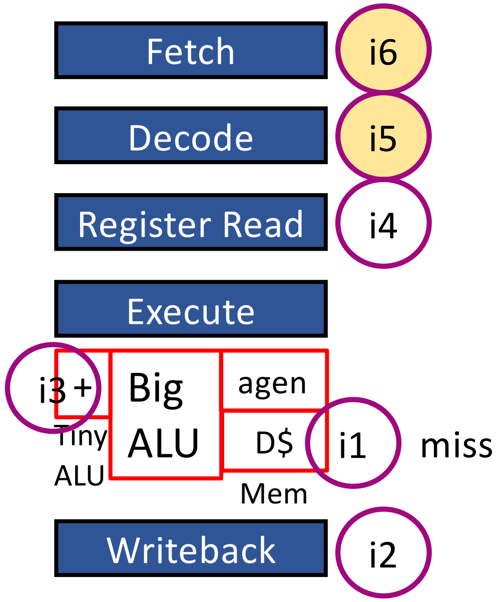
i1: load r2, #0(r1)

i2: add r4, r3, #1

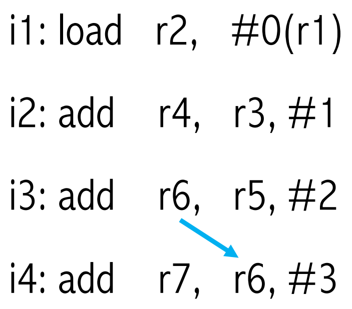
i3: add r6, r5, #2

i4: add r7, r6, #3

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$		…miss…						
i2		FE	DE	RR	EX								
i3			FE	DE	RR								
i4				FE	DE								



Scenario 1: load miss followed by independent instructions



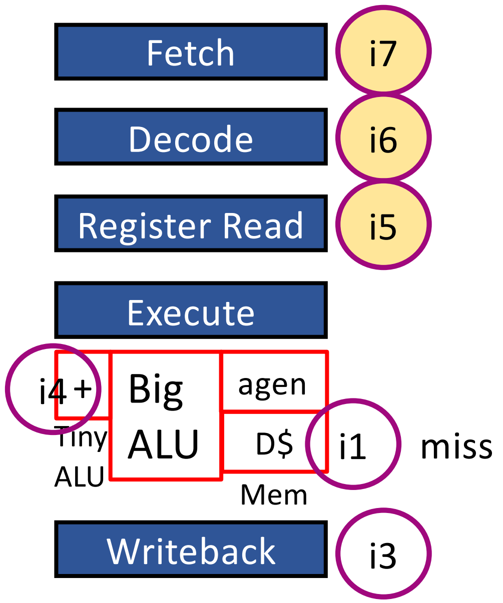
i1: load r2, #0(r1)

i2: add r4, r3, #1

i3: add r6, r5, #2

i4: add r7, r6, #3

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$	…miss…							
i2		FE	DE	RR	EX	WB							
i3			FE	DE	RR	EX							
i4				FE	DE	RR							



Scenario 1: load miss followed by independent instructions

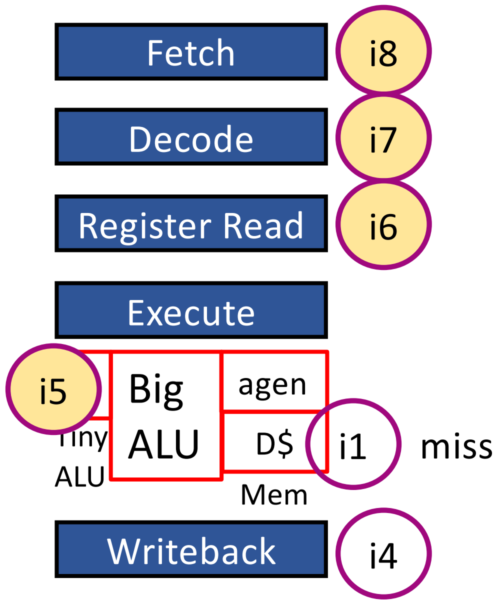
i1: load r2, #0(r1)

i2: add r4, r3, #1

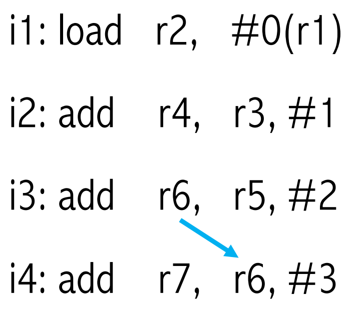
i3: add r6, r5, #2

i4: add r7, r6, #3

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$	...miss...							
i2		FE	DE	RR	EX	WB							
i3			FE	DE	RR	EX	WB						
i4				FE	DE	RR	EX						



Scenario 1: load miss followed by independent instructions



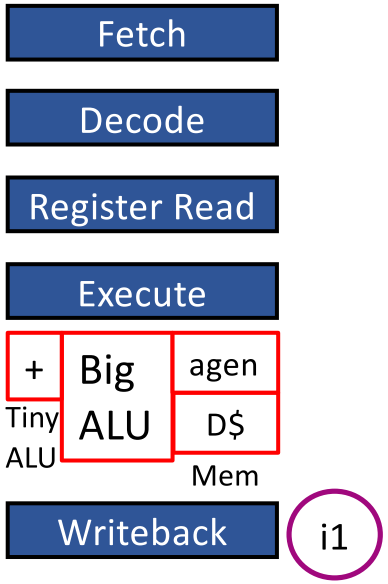
i1: load r2, #0(r1)

i2: add r4, r3, #1

i3: add r6, r5, #2

i4: add r7, r6, #3

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$	...miss...							
i2		FE	DE	RR	EX	WB							
i3			FE	DE	RR	EX	WB						
i4				FE	DE	RR	EX	WB					

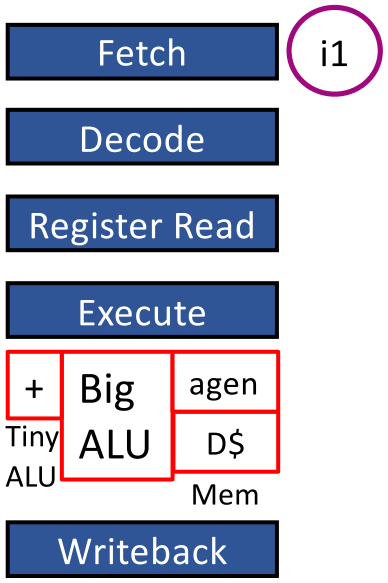


Scenario 1: load miss followed by independent instructions

```
i1: load   r2,   #0(r1)
i2: add    r4,   r3, #1
i3: add    r6,   r5, #2
i4: add    r7,   r6, #3
```

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$		…miss…			WB			
i2		FE	DE	RR	EX	WB							
i3			FE	DE	RR	EX	WB						
i4				FE	DE	RR	EX	WB					

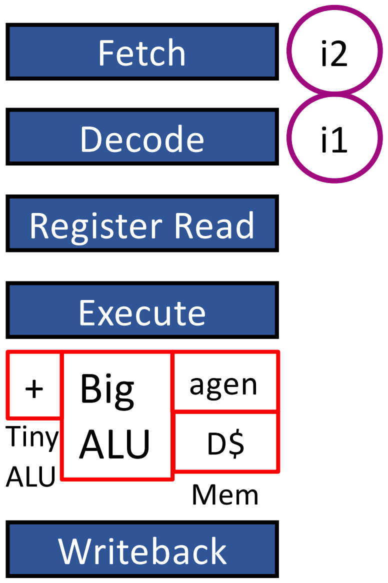
Scenario 2: Load miss followed by dependent instruction, followed by independent instructions



Scenario 2: load miss followed by dependent instruction, followed by independent instructions

```
i1: load  r2,  #0(r1)
i2: add   r4,  r2, #1
i3: add   r6,  r5, #2
i4: add   r7,  r6, #3
```

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE												
i2													
i3													
i4													



Scenario 2: load miss followed by dependent instruction, followed by independent instructions

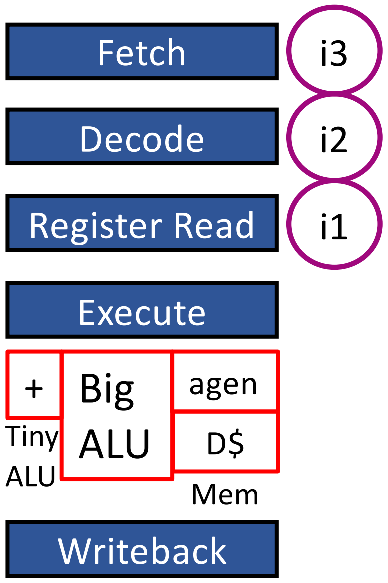
i1: load r2, #0(r1)

i2: add r4, r2, #1

i3: add r6, r5, #2

i4: add r7, r6, #3

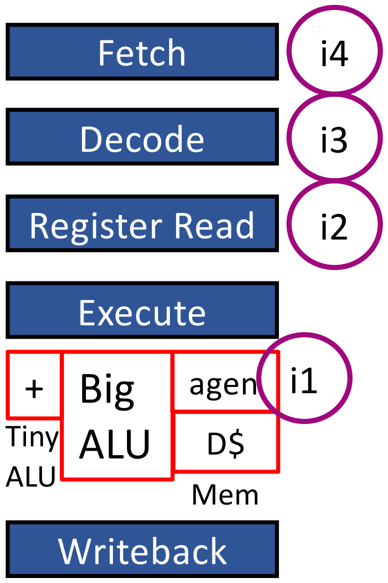
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE											
i2		FE											
i3													
i4													



Scenario 2: load miss followed by dependent instruction, followed by independent instructions

```
i1: load  r2,  #0(r1)
i2: add   r4,  r2, #1
i3: add   r6,  r5, #2
i4: add   r7,  r6, #3
```

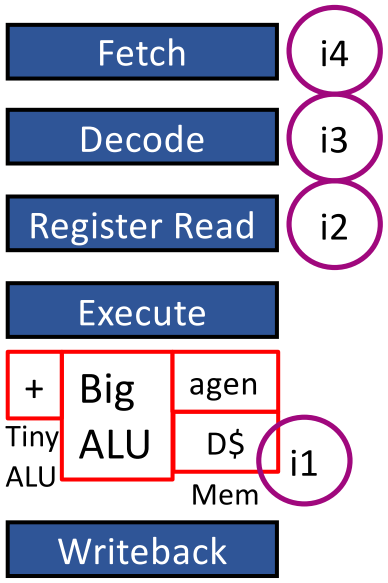
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR										
i2		FE	DE										
i3			FE										
i4													



Scenario 2: load miss followed by dependent instruction, followed by independent instructions

```
i1: load  r2,  #0(r1)
i2: add   r4,  r2, #1
i3: add   r6,  r5, #2
i4: add   r7,  r6, #3
```

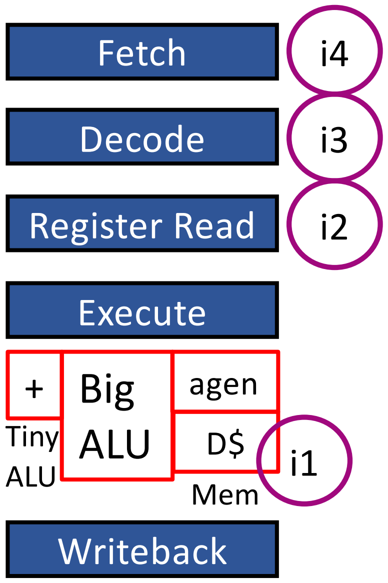
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$									
i2		FE	DE	RR									
i3			FE	DE									
i4				FE									



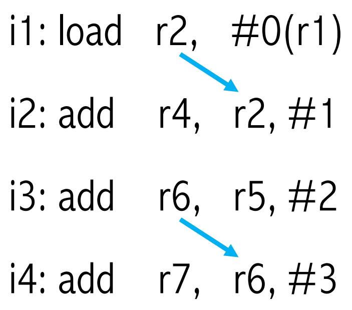
Scenario 2: load miss followed by dependent instruction, followed by independent instructions

```
i1: load  r2,  #0(r1)
i2: add   r4,  r2, #1
i3: add   r6,  r5, #2
i4: add   r7,  r6, #3
```

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$	…miss…							
i2		FE	DE	RR	RR								
i3			FE	DE	DE								
i4				FE	FE								

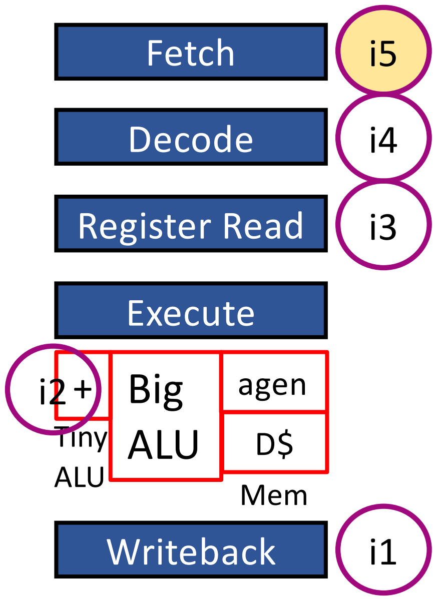


Scenario 2: load miss followed by dependent instruction, followed by independent instructions



```
i1: load  r2,  #0(r1)
i2: add   r4,  r2, #1
i3: add   r6,  r5, #2
i4: add   r7,  r6, #3
```

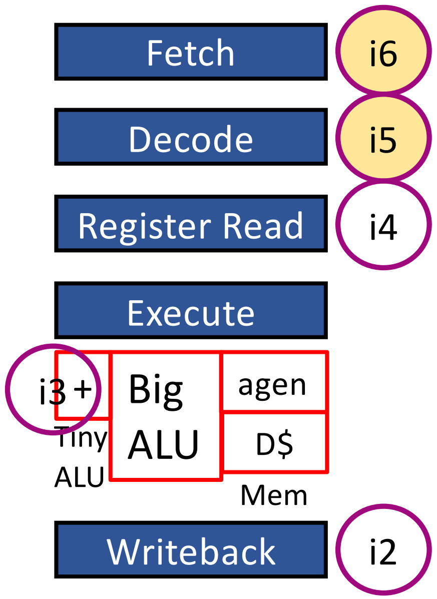
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$			…miss…					
i2		FE	DE	RR	RR	RR	RR	RR	RR				
i3			FE	DE	DE	DE	DE	DE	DE				
i4				FE	FE	FE	FE	FE	FE				



Scenario 2: load miss followed by dependent instruction, followed by independent instructions

```
i1: load  r2,  #0(r1)
i2: add   r4,  r2, #1
i3: add   r6,  r5, #2
i4: add   r7,  r6, #3
```

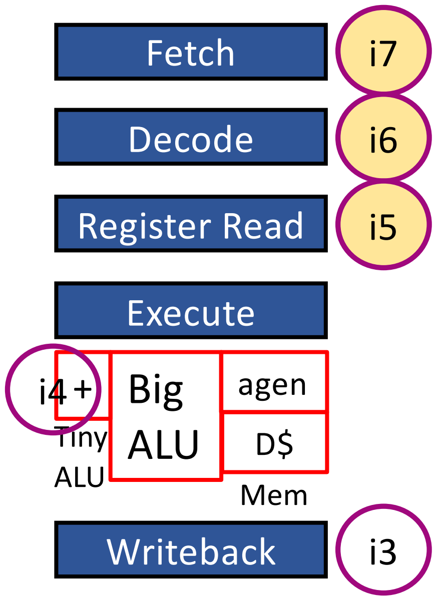
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$	…miss…				WB			
i2		FE	DE	RR	RR	RR	RR	RR	RR	EX			
i3			FE	DE	DE	DE	DE	DE	DE	RR			
i4				FE	FE	FE	FE	FE	FE	DE			



Scenario 2: load miss followed by dependent instruction, followed by independent instructions

$$\begin{tikzcd}[row sep=small, column sep=small]
\text{i1: load} & \text{r2,} \arrow[dr] & \text{\#0(r1)} \\
\text{i2: add} & \text{r4,} & \text{r2, \#1} \\
\text{i3: add} & \text{r6,} \arrow[dr] & \text{r5, \#2} \\
\text{i4: add} & \text{r7,} & \text{r6, \#3}
\end{tikzcd}$$

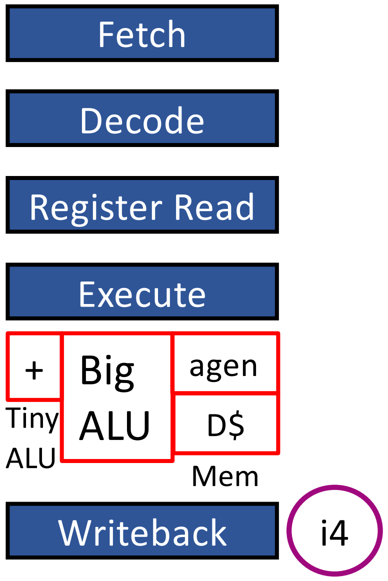
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_@$	$EX_{D\$}$	…miss…				WB			
i2		FE	DE	RR	RR	RR	RR	RR	RR	EX	WB		
i3			FE	DE	DE	DE	DE	DE	DE	RR	EX		
i4				FE	FE	FE	FE	FE	FE	DE	RR		



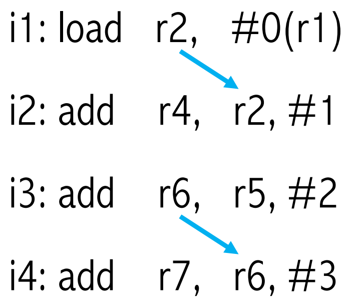
Scenario 2: load miss followed by dependent instruction, followed by independent instructions

```
i1: load   r2,   #0(r1)
i2: add    r4,   r2, #1
i3: add    r6,   r5, #2
i4: add    r7,   r6, #3
```

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$		...miss...			WB			
i2		FE	DE	RR	RR	RR	RR	RR	RR	EX	WB		
i3			FE	DE	DE	DE	DE	DE	DE	RR	EX	WB	
i4				FE	FE	FE	FE	FE	FE	DE	RR	EX	



Scenario 2: load miss followed by dependent instruction, followed by independent instructions



```
i1: load  r2,  #0(r1)
i2: add   r4,  r2, #1
i3: add   r6,  r5, #2
i4: add   r7,  r6, #3
```

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$		…miss…			WB			
i2		FE	DE	RR	RR	RR	RR	RR	RR	EX	WB		
i3			FE	DE	DE	DE	DE	DE	DE	RR	EX	WB	
i4				FE	FE	FE	FE	FE	FE	DE	RR	EX	WB

In-Order Issue Bottleneck

- `i2` must wait for `i1`
  - `i2` depends on `i1` (chain of dependent instructions)
- `i3`, `i4` need not wait for the `i1–i2` chain
  - They are independent
- But the `i3–i4` chain stalls
  - Key insight: *In-order issue translates into a structural hazard*
  - *RR stage (issue stage) blocked by the stalled `i2`*

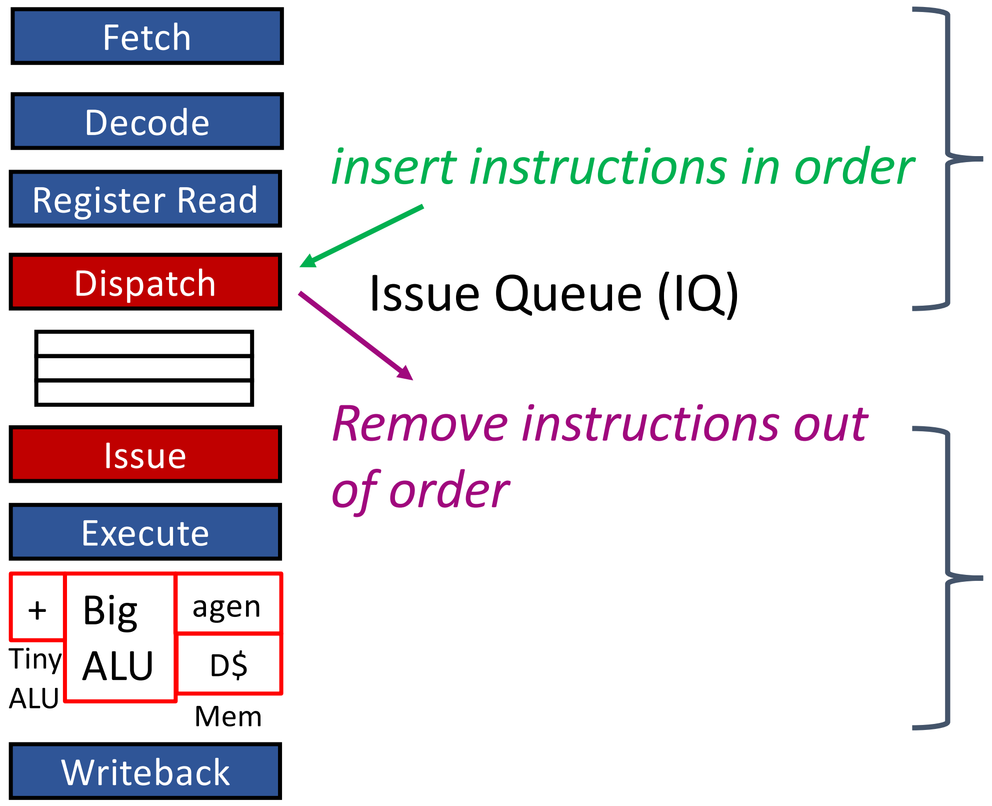
OOO pipeline unblocks RR (issue) using a new instruction buffer for stalled data-dependent instructions

- *A structure with many names: “Reservation stations”, “issue buffer”, “issue queue”, “scheduler”, “scheduling window”*

Issue Queue

- Stalled instructions do not impede instruction fetch
- Younger **ready** instructions issue and execute out of order with respect to older **non-ready** instructions
- Issue queue opens up the pipeline to future independent instructions
  - Tolerate long latencies (cache misses, floating point)
  - Exploit ILP (critical for superscalar)

Out-of-Order Scalar Pipeline (v.1)



In-order fetch/dispatch engine

OOO issue/execute engine

Summary and Exercises

In-order to OOO Transformation

Naïve in-order suffers from structural hazards

Aggressive in-order shows the real problem with in-order

In-order issue bottleneck: *RAW hazards turn into structural hazards*

Independent instructions after the dependent instruction stall

OOO pipeline unblocks RR (issue) using a new instruction buffer for stalled data-dependent instructions

- *A structure with many names: "Reservation stations", "issue buffer", "issue queue", "scheduler", "scheduling window"*

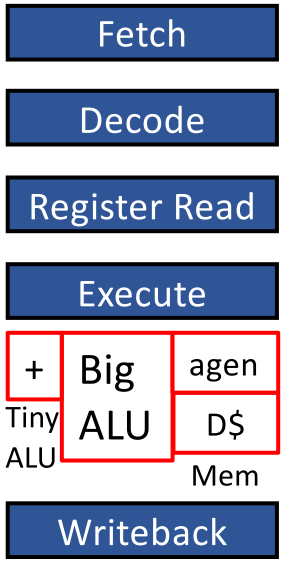
Types of In-Order

Stall on miss (*simple issue policy*)

- Stall the pipeline on a long-latency event such as a cache miss
- Naïve, simplest, extremely low power environments
- Can still have forwarding
- MIPS 5-stage pipeline is stall-on-miss (*although we avoided structural hazards with simplified assumptions*)

Stall on use (*aggressive issue policy*)

- Stall the pipeline on a RAW hazard (when the “use” instruction is encountered in the register read stage → issue logic)
- Need book-keeping and hazard detection logic to track busy registers (WAW) and outstanding events in the execute stage
- Extra care for managing pipeline registers



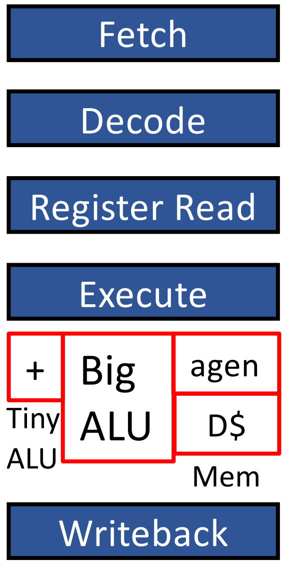
Exercise 1

Scenario 1: load miss followed by dependent instruction, followed by load miss and a dependent instruction

```
i1: load  r2,  #0(r1)
i2: add   r4,  r2, #1
i3: load  r6,  #0(r5)
i4: add   r7,  r6, #3
```

Fill the table below for stall-on-miss and stall-on-use.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
i1																				
i2																				
i3																				
i4																				



Scenario 1: load miss followed by dependent instruction, followed by load miss and a dependent instruction

```
i1: load   r2,   #0(r1)
i2: add    r4,   r2, #1
i3: load   r6,   #0(r5)
i4: add    r7,   r6, #3
```

Both scenarios respond similarly

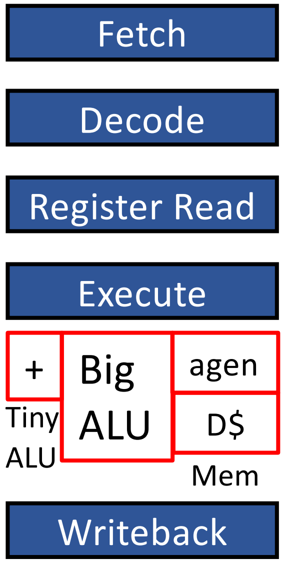
cache miss is resolved

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$		...miss...			WB										
i2		FE	DE	RR	RR	RR	RR	RR	RR	EX	WB									
i3			FE	DE	DE	DE	DE	DE	DE	RR	$EX_{@}$	$EX_{D\$}$		...miss...			WB			
i4				FE	FE	FE	FE	FE	FE	DE	RR	RR	RR	RR	RR	RR	EX	WB		

wasted fetch cycles

wasted fetch cycles

The load-use scenario obfuscates the key distinction between stall-on-miss and stall-on-use issue policies



Exercise 2

Scenario 2 (*statically reordered*): load miss followed by dependent instruction, followed by load miss + dependent instruction

i1: load r2, #0(r1)

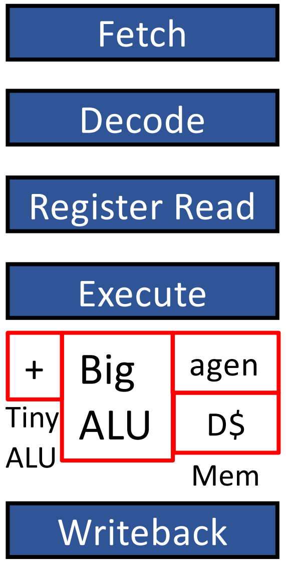
i2: load r6, #0(r5)

i3: add r4, r2, #1

i4: add r7, r6, #3

Fill the table below for stall-on-miss and stall-on-use.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
i1																				
i2																				
i3																				
i4																				



Scenario 2 (*statically reordered*): load miss followed by dependent instruction, followed by load miss + dependent instruction

i1: load r2, #0(r1)

i2: load r6, #0(r5)

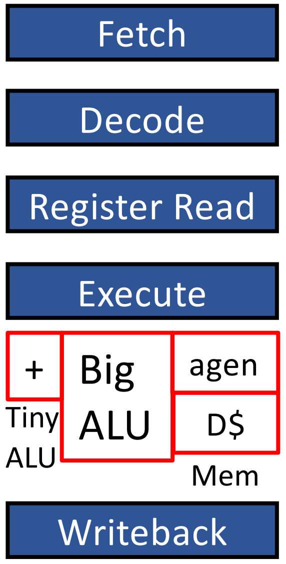
i3: add r4, r2, #1

i4: add r7, r6, #3

miss *miss*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
i1	FE	DE	RR	EX$_@$	EX$_{D\$}$		...miss...			WB										
i2		FE	DE	RR	RR	RR	RR	RR	RR	EX$_@$	EX$_{D\$}$		...miss...			WB				
i3			FE	DE	DE	DE	DE	DE	DE	RR	RR	RR	RR	RR	RR	EX	WB			
i4				FE	FE	FE	FE	FE	FE	DE	DE	DE	DE	DE	DE	RR	EX	WB		

Stall-on-miss *is unable to execute the two loads simultaneously (missed opportunity to exploit memory-level parallelism)*



Scenario 2 (*statically reordered*): load miss followed by dependent instruction, followed by load miss + dependent instruction

```
i1: load   r2,   #0(r1)

i2: load   r6,   #0(r5)

i3: add    r4,   r2, #1

i4: add    r7,   r6, #3
```

miss

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
i1	FE	DE	RR	$\text{EX}_{@}$	$\text{EX}_{D\$}$		…miss…			WB										
i2		FE	DE	RR	$\text{EX}_{@}$	$\text{EX}_{D\$}$		…miss…			WB									
i3			FE	DE	RR	RR	RR	RR	RR	EX	WB									
i4				FE	DE	DE	DE	DE	DE	DE	RR	EX	WB							

***Stall-on-use** can execute the two loads simultaneously and exploit memory-level parallelism (with a little extra book-keeping hardware)*