EEL 4768
Computer Architecture

Instruction Level Parallelism

Register Renaming/ Out of order Ex.

Types of Parallelism

- Hardware level:
  - Uniprocessor
    - Pipelining
    - Superscalar, VLIW
    - Multithreading
  - Multiprocessor architecture
  - Distributed computer architecture
- Software level:
  - Task level Parallelism
  - Data level Parallelism
  - Memory level Parallelism
  - Instruction level Parallelism

Instruction-Level Parallelism (ILP)

- *Definition:* Instruction-Level Parallelism (ILP) is a general concept that implies multiple instructions execute in parallel

- Pipelining become universal technique in 1985
  - Overlaps execution of instructions
  - Exploits "Instruction Level Parallelism"

Pipeline:

- 5-stage pipeline: up to five instructions in the datapath
- Therefore, the pipelined datapath is a form of ILP
- The overlap in the pipelined datapath is called '**partial overlap**' since each instruction is using a different part of the pipeline (e.g: we can't have two instructions using the ALU in the same clock cycle since there's one ALU)

Instruction-Level Parallelism

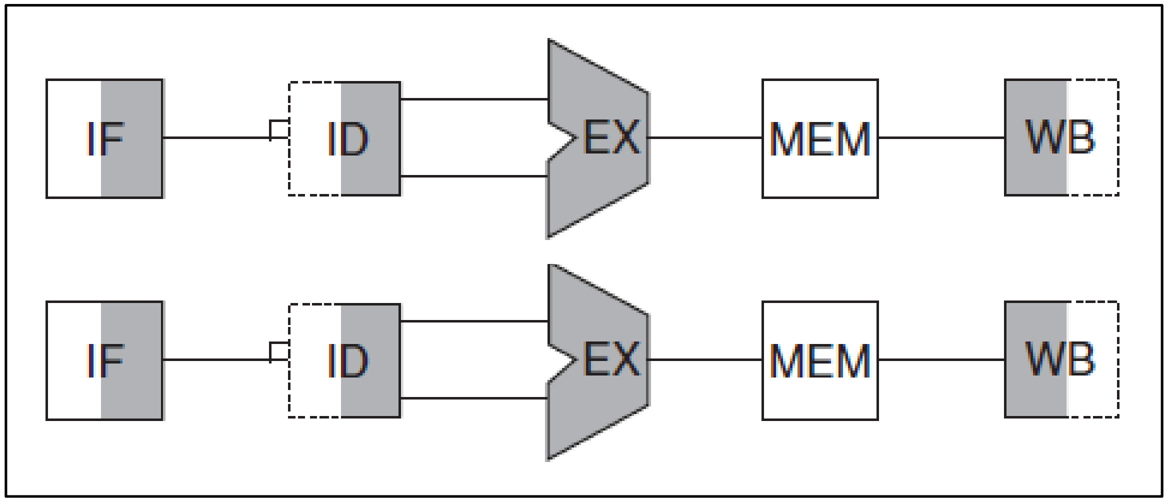
- When exploiting instruction-level parallelism, goal is to minimize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls
  - Ideal pipeline CPI: 1
- Goal: To achieve CPI < 1

Multiple-Issue Datapath

- Multiple parallel pipelines:
  - Multiple instructions can be issued (started) in a clock cycle

- **'Two-way multiple issue CPU/datapath':**
  - The two pipelines contain at most 10 instructions at a time
  - The two pipelines are synchronized to resolve data dependences

Two pipelines in the CPU



CPI (Clock Per Instruction) vs. IPC (Instruction Per Clock)

- A single-issue pipelined datapath can finish one instruction per clock cycle when there are no stalls
- Therefore, it can achieve: **CPI = 1**

- *Note: the single-cycle datapath can achieve a CPI=1, but its clock cycle is much longer than the pipelined datapath's*

- A multiple-issue pipelined datapath can finish multiple instructions per clock cycle
- The two-way multiple issue CPU can finish two instructions per cycle, therefore, it can achieve: **CPI = 0.5**

- In this case, instead of measuring CPI (Clocks Per Instruction), we measure Instructions Per Clock (IPC)
- Therefore, the two-way multiple issue CPU can do: **IPC = 2**

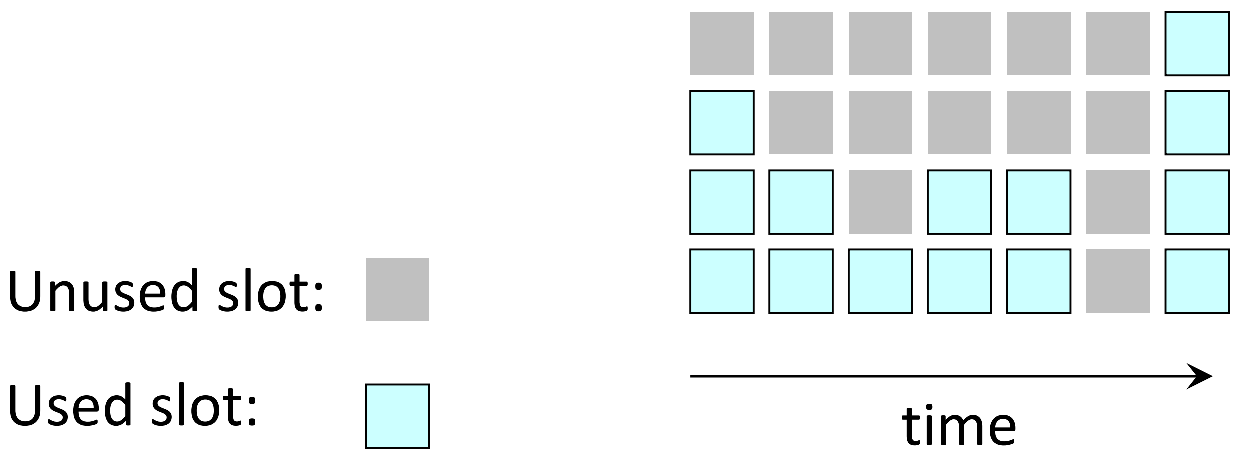
Multiple-Issue CPU

- Most of today's advanced CPUs are multiple-issue
  - Some basic CPUs used in embedded systems might not be multiple-issue
  - But CPUs used in desktops, servers, smartphones are multiple-issue

- Today's CPUs attempt to **issue from 3 to 6 instructions** per cycle
- However, due to data dependences, there is some limitation on which instructions can execute in parallel
- Stalls also happen, so this might slow down the CPU

Multiple-Issue CPU

- The figure below is one way to represent a 4-way multiple issue CPU
- Each box is called an **'issue slot'**
- In the first cycle, we only issued 3 instructions since we can't find a 4th one that doesn't have a dependence with the other 3 instructions
- Therefore, not all the four issue slots can always be filled
- In one slot in the figure, there's a stall and no instruction was issued



Multiple-Issue CPU

- *What are the tasks needed to support multiple-issue CPUs?*

- We should determine which instructions can execute in parallel
- Instructions with data dependences can't run in parallel
  - They could have partial overlap and we might possibly need to do forwarding

Who can do this task?

- Hardware-based dynamic approaches
  - The hardware groups the instruction at run-time, hence it's the **dynamic** approach
  - Used in server and desktop processors
- Static: Compiler-based static approaches
  - The compiler groups the instructions during compilation, hence it's the **static** approach
  - Not as successful outside of scientific applications

Summary: Static vs. Dynamic ILP

Compiler approach	**Hardware approach**
Static also called **VLIW** (Very Long Instruction Word)	**Dynamic** also called **superscalar**
(+) Analyze the code multiple times at compilation and package instructions	Hardware looks at a window of ~100 instructions and runs all ready
(-) Can't see run-time event, eg: cache missed, exceptions	(+) Can see run-time events and adjust instruction execution
(-) Compile code for a specific hardware	(+) Code runs fast on all hardware since the hardware is in charge
	* Usually the preferred approach
Speculation: execute code even though it's not sure this code must be executed (is done instead of idling)	

Multiple-Issue CPU: Compiler

- The compiler is a good candidate for grouping the instructions
- This is because the compiler can traverse the code and analyze it multiple times during compilation
- The compiler can group instructions into issue slots, a process referred to as **packaging**

- The compiler’s strength is the ability to see all the code at compile time
- However, the compiler can’t see run-time events that affect the optimum packaging, e.g.:
  - cache misses, exceptions, branch stalls
- Accordingly, the CPU is allowed to alter the packaging that the compiler made to adapt to run time events

Multiple-Issue CPU: Compiler

- The compiler can easily see the hazard in (Code 1) and will create enough separation between the load and the add

<u>Code 1</u>

```
lw     $t0, 12($s0)
add    $a0, $a0, $t0
```

<u>Code 2</u>

```
lw     $t0, 0($s0)
add    $t2, $t3, $t4
add    $t2, $t2, $t5
add    $t2, $t2, $t6
sub    $t2, $t2, $t0
```

- However, (Code 2) might seem to be fine since the dependence on t0 is separated by multiple instructions
- But, if $t0 misses in the cache and it took 100s of cycles to access, the sub instruction would have to be stalled
- The hardware is better positioned to observe the cache miss and re-schedule the sub accordingly

Multiple-Issue CPU: Compiler

- There are other situations that the hardware is better positioned to observe than the compiler, e.g.:
  - Exceptions
  - Stalls caused by memory read or write
  - Branch stalls

Conclusion

- Even if the compiler is primarily in charge, the compiler and the hardware collaborate
  - The compiler deals with the hazards that can be seen at compile time, then the hardware deals with the other hazards that are observed at run time
  - The compiler packages the instructions, then the hardware might have to modify the packages

Multiple-Issue CPU: Hardware

- Another approach is to have the hardware primarily in charge
- The hardware determines which instructions execute in parallel

- Limitation: cannot look at all the code and analyze
- It usually looks at a window of ~100 instructions and analyze them
- Therefore, the compiler will arrange the instructions in a ‘beneficial order’ by separating the dependences
- This helps the hardware in executing more instructions in parallel

Conclusion

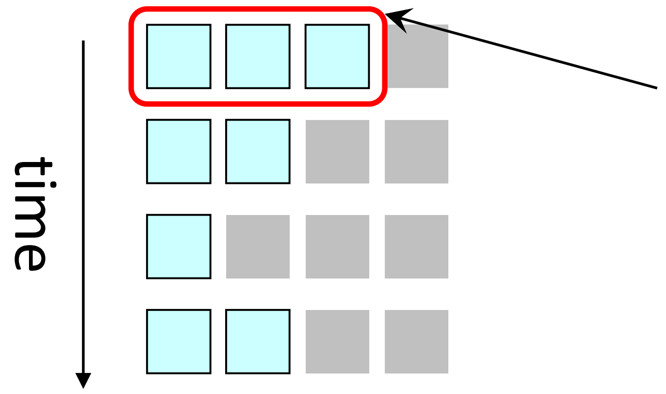
- Whether the compiler or the hardware is in charge, they always collaborate in the multiple-issue CPU

Static Multiple Issue

- The compiler groups instructions into an ‘**issue packet**’
- In the figure below, the first issue packet contains 3 instructions
- We can think of an issue packet as a very long instruction
- The issue packet with 3 instructions is like a large instruction with 96 bits (32-bit instruction x 3)
- Therefore, the issue packet is called a **Very Long Instruction Word (VLIW)**
- Therefore, static multiple issue CPUs are called **VLIW CPUs**

Unused slot: ▨

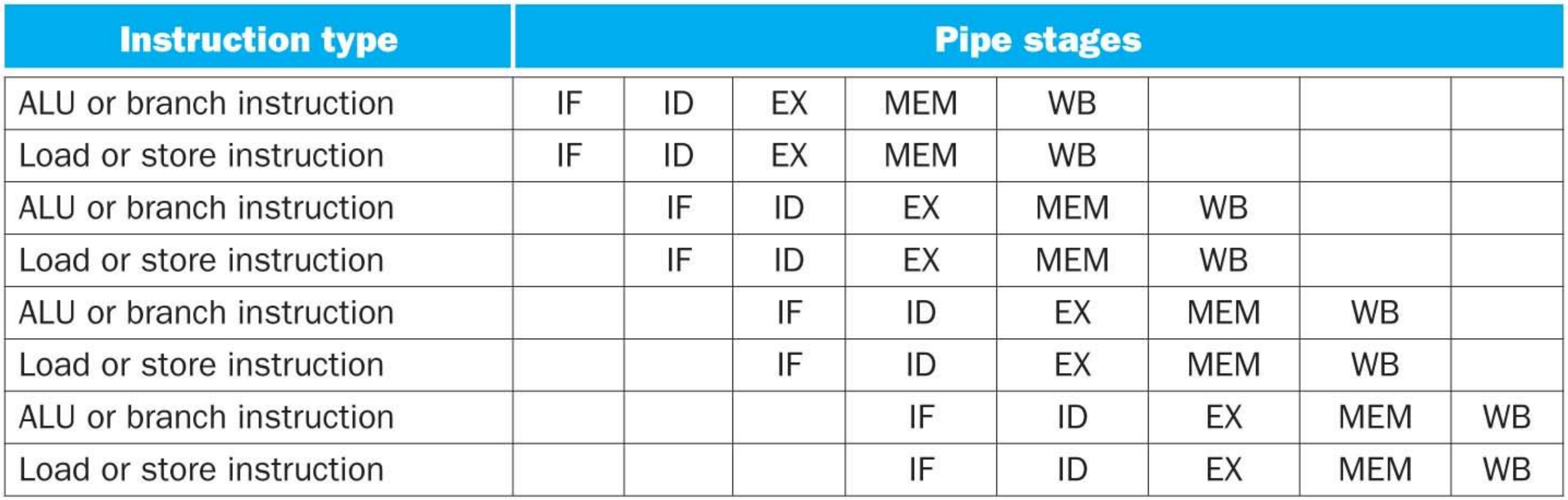
Used slot: ▢



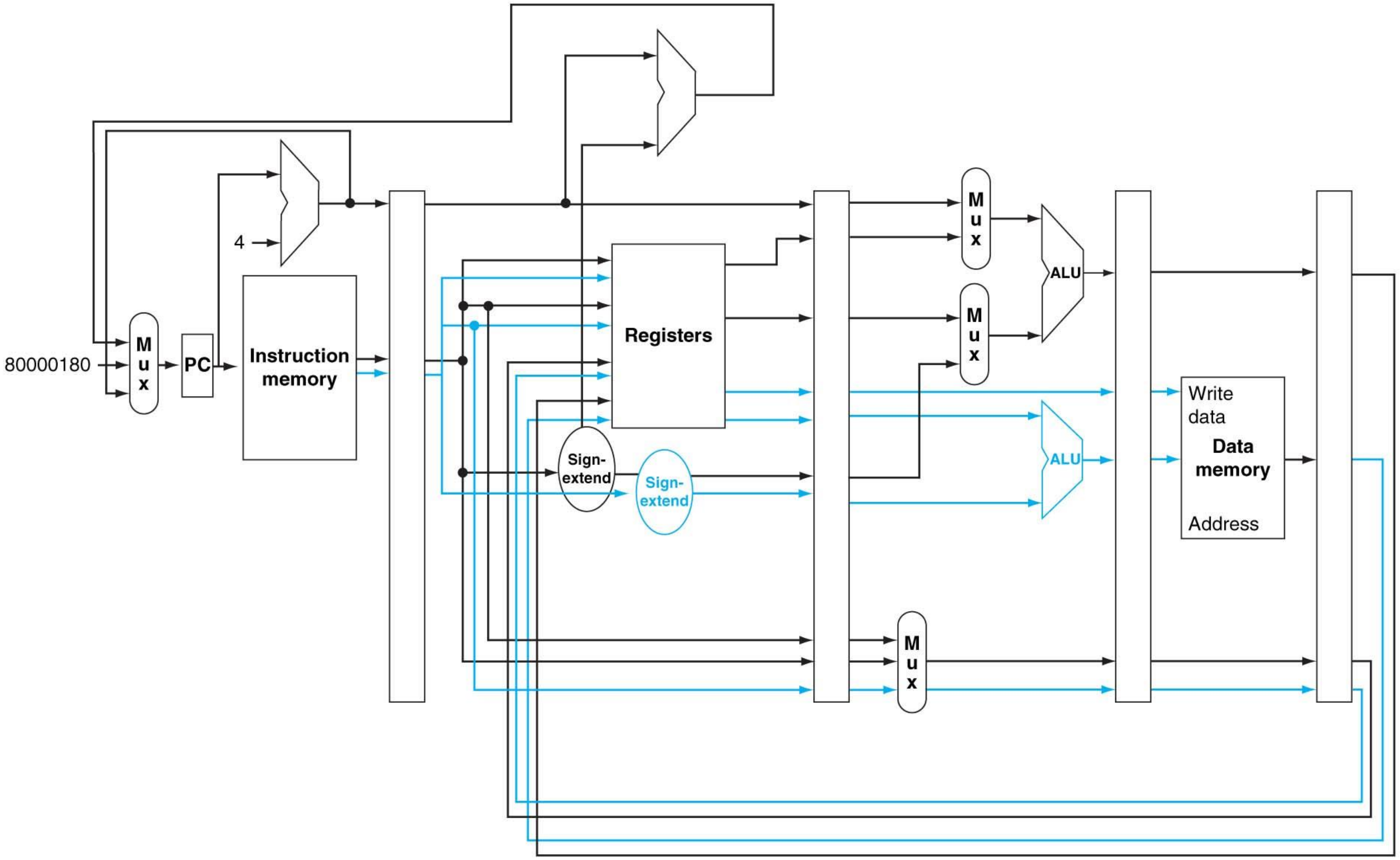
These multiple instructions issued together can be viewed as a ‘Very Long Instruction Word’ (VLIW)

Static Multiple Issue

- The table represents a static two-issue MIPS CPU
- **An issue packet contains (an ALU or branch) and (a load or store)**
- *We can't have two 'loads' or two R-types in one* issue packet*
- This condition reduces the number of components in the datapath
- In every clock cycle, the CPU fetches 64 bits of instruction that are aligned on the 64-bit boundary



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



80000180
M u x
PC
4
Instruction memory
Registers
Sign-extend
Sign-extend
M u x
M u x
ALU
ALU
M u x
Write data
Data memory
Address

Static Multiple Issue

- The datapath figure corresponds to the static two-issue MIPS CPU
- This is the VLIW content:

VLIW	ALU or Branch	Load or Store

How many registers could we want to read?

- At most (R-type: 2 and Store: 2), three registers
- Therefore, the register file is modified to allow reading four registers

How many register could we want to write?

- At most (R-type: 1 and Load: 1), two registers
- Therefore, the register file is modified to allow writing two registers

How many ALU computations in the EX stage?

- The ALU processes the (R-type or Branch) and the extra adder processes the (load or store)

Static Multiple Issue

- The MIPS code below loads a value from an array into $t0, then adds $s2 to it and stores the result in the same array location
- The loop stops when the address becomes zero (s1 is initialized as the address of the last element in the array and goes down to zero)

- How can this code be scheduled on the two-issue CPU?

```
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, $zero, Loop # branch $s1!=0
```

Static Multiple Issue

- This is one way to schedule the code on the two-issue CPU
- *Remember, the first instruction in the VLIW is ALU or branch and the second instruction is load or store*

```
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, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	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

- This schedule achieves an IPC (Instruction-Per-Cycle) value of 5/4=1.25 out of a maximum IPC of 2; there are too many nops!

Static Multiple Issue: Loop Unrolling

- How can we schedule the previous code to get a better performance?
- <u>Observation:</u> Usually in a loop code, there is a lot of dependence; however, in the previous code there are no dependences between different iterations of the loop
  - Every iteration of the loop reads an array element, adds $s2 to it and stores it back
  - Therefore, different iterations use different data

- A technique that can be used here is called ‘**loop unrolling**’
- Let’s unroll the loop so we have fewer iterations in the loop code; the total number of iterations decreases; the loop code becomes larger but it’s possible to overlap more instructions

Loop Unrolling

- The original loop has four instructions and iterates 100 times
- The loop code is written twice, back-to-back, making the loop twice as large, but now iterates 50 times
- The two codes are logically equivalent

```
Loop 100 times
  Instruction 1
  Instruction 2
  Instruction 3
  Instruction 4
```

```
Loop 50 times
  Instruction 1
  Instruction 2
  Instruction 3
  Instruction 4
  Instruction 1'
  Instruction 2'
  Instruction 3'
  Instruction 4'
```

Loop Unrolling

- Now each loop iteration processes **four** elements in the array
- The address register $s1 is decreased by 16 bytes to jump over 4 words

```
Loop: lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      lw   $t0, -4($s1)
      addu $t0, $t0, $s2
      sw   $t0, -4($s1)
      lw   $t0, -8($s1)
      addu $t0, $t0, $s2
      sw   $t0, -8($s1)
      lw   $t0, -12($s1)
      addu $t0, $t0, $s2
      sw   $t0, -12($s1)
      addi $s1, $s1,-16
      bne  $s1, $zero, Loop
```

Most of the instructions use the register $t0 which means we can't reorder the instructions.

However, there is no real dependence between the 4 operations in the loop's code.

A technique called '**register renaming**' allows us to deal with this.

Static Multiple Issue: Register Renaming

- Instead of using $t0 for the four iteration codes, we're also using the registers $t1, $t2 and $t3

```
Loop: lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      lw   $t1, -4($s1)
      addu $t1, $t1, $s2
      sw   $t1, -4($s1)
      lw   $t2, -8($s1)
      addu $t2, $t2, $s2
      sw   $t2, -8($s1)
      lw   $t3, -12($s1)
      addu $t3, $t3, $s2
      sw   $t3, -12($s1)
      addi $s1, $s1,-16
      bne  $s1, $zero, Loop
```

Register renaming is used when there is a dependence on the name of the register but there is no real dependence on the data between the instructions.

Static Multiple Issue: Reordering

- The code on the previous slide had ‘lw’ followed by ‘addu’ with a data dependency; the code also had dependency between the ‘addu’ and the ‘store’
- ‘nops’ can be avoided by reordering the instructions as shown below

```
Loop: lw   $t0, 0($s1)
      lw   $t1, -4($s1)
      lw   $t2, -8($s1)
      lw   $t3, -12($s1)
      addu $t0, $t0, $s2
      addu $t1, $t1, $s2
      addu $t2, $t2, $s2
      addu $t3, $t3, $s2
      sw   $t0, 0($s1)
      sw   $t1, -4($s1)
      sw   $t2, -8($s1)
      sw   $t3, -12($s1)
      addi $s1, $s1,-16
      bne  $s1, $zero, Loop
```

This code can benefit more from instruction overlapping and achieves a better performance

Static Multiple Issue

- The offset in the 'lw' and 'sw' instructions have also changed
- In Cycle 1, the 'lw' will use $s1 before it's incremented so it uses the original address in $s1

	ALU/branch	Load/store	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, $t4, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

- In 8 clock cycles, there are only two nops
- The IPC achieved by this code is: 14/8 = 1.75
- This is much better than the IPC of 1.25 without unrolling the loop

Static Multiple Issue

- Unrolling the loop made the code run 1.4 times faster
  - IPC=1.25 vs. IPC=1.75
- However, unrolling the loop made the code larger
- It also used more registers ($t0 to $t3) instead of using $t0 only
- To mitigate this, the CPUs usually contain special registers that are used for register renaming

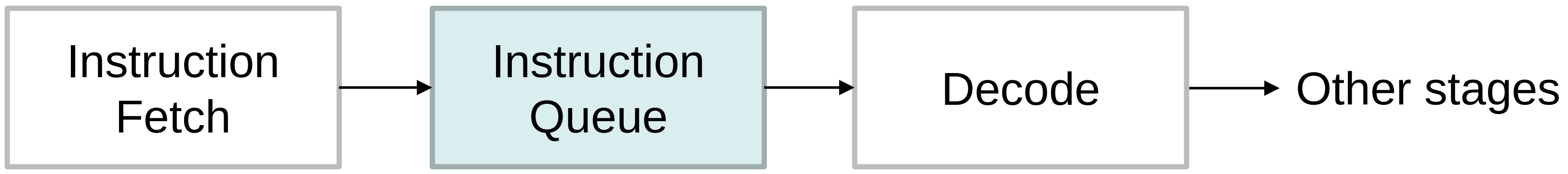
Dynamic Multiple Issue

- Dynamic multiple-issue CPUs are called **superscalar CPUs**
- A **scalar CPU** fetches one instruction at a time

- In superscalar CPUs, the hardware determines which instructions execute in parallel
- The approach is slightly different from the VLIW/static CPU
- Superscalar doesn’t package instructions
- Instead, the execution is based on queues

Superscalar CPU: Instruction Queue

- Instructions are fetched and stored in an instruction queue.
- Hardware finds out the dependencies between the instructions in the queues.



Dynamic Multiple Issue & Compiler

- The compiler helps the hardware by spreading out dependences
- This helps the hardware become more successful at overlapping instructions

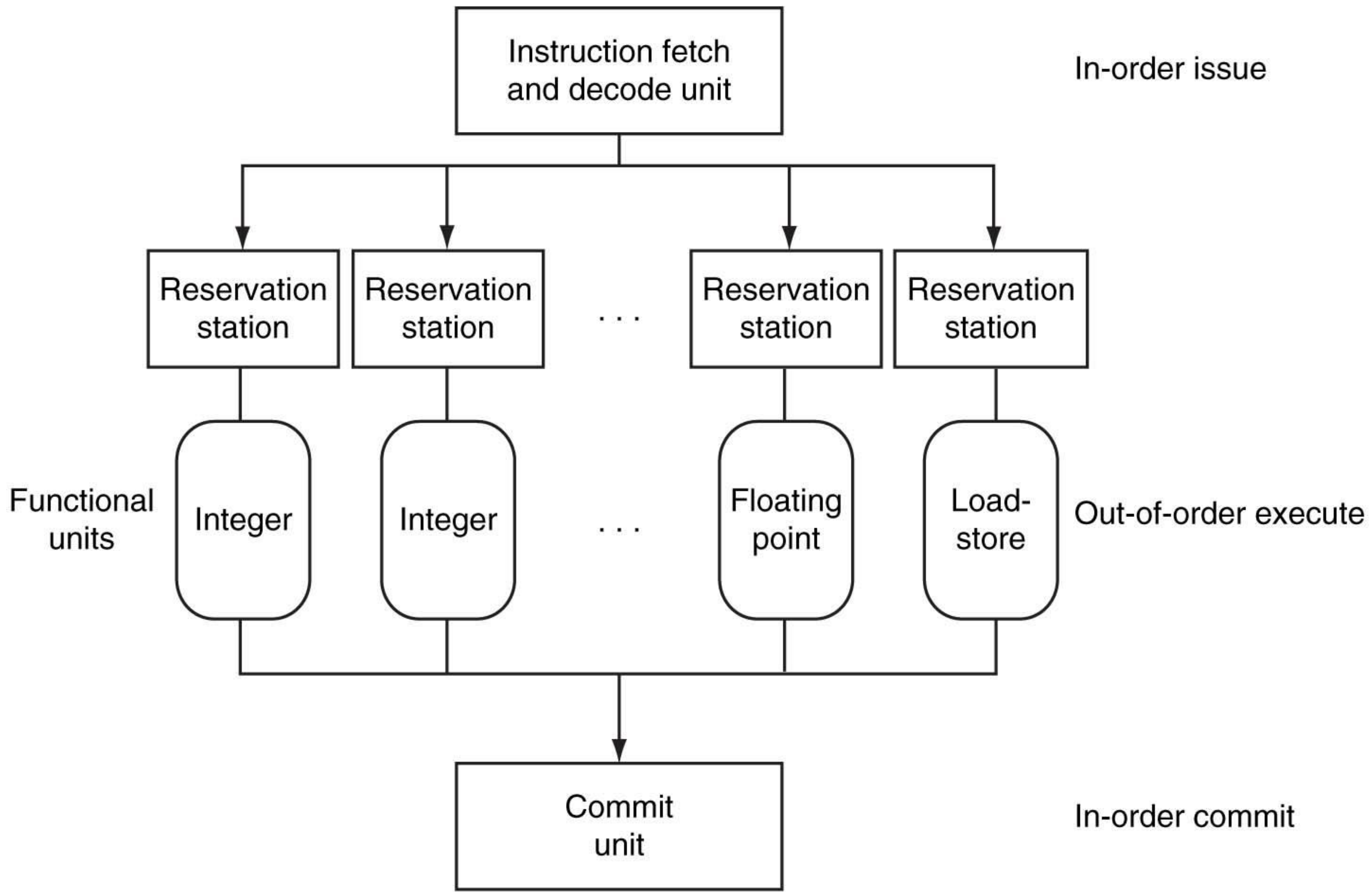
- The code is compiled in the same way independently of the hardware structure
- For example, the compiler compiles in the same way for a 5-stage pipeline or a 12-stage pipeline
- The compiler separates dependencies as much as it can
- Then, it's up to each hardware to discover which instructions can execute in parallel

Dynamic Multiple Issue

- The instructions issue in-order **so the dependences can be tracked**
- They execute in-order or out-of-order **(out-of-order speeds up the execution)**
- They commit the result in-order **(so the code executes correctly)**

Stages	**Tasks**	**Strategies (Based on order of instructions in the code)**
Issue	Fetch the instructions	In-order
Execute	ALU operations / memory access	In-order / out-of-order
Commit	Write the result to a register / Write to the memory	In-order

Superscalar Datapath



Dynamic Multiple Issue

Instruction fetch and decode unit

- Fetches the instructions and tracks the dependences
- This is done in-order so the dependences can be observed

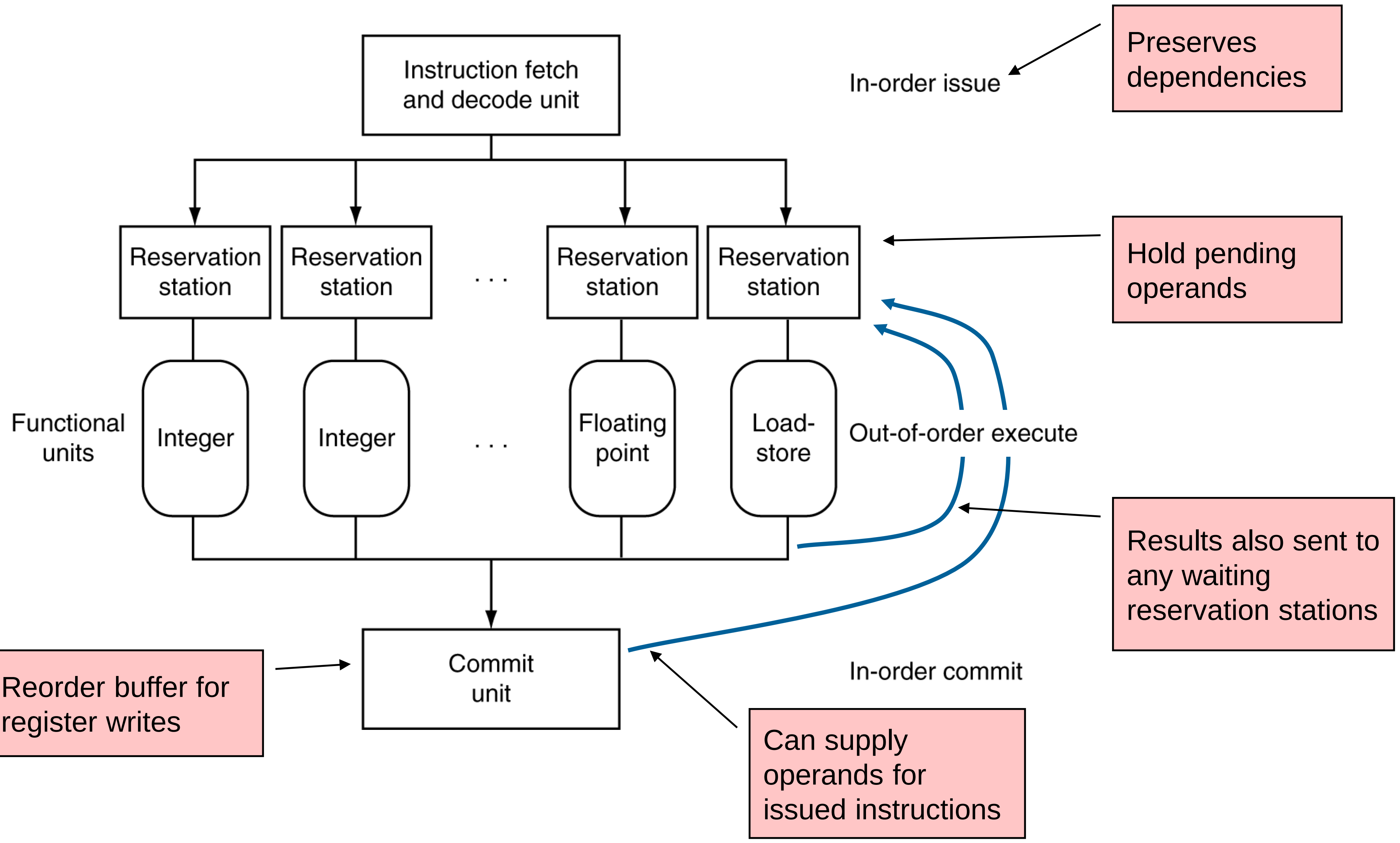
Reservation station & function units

- A queue where the instruction waits until all of its operands are ready
- The operands may be hampered by a cache miss or by another instruction waiting on the cache miss
- The function units execute the instruction

Commit unit

- Writes the results (commits them) to registers and to memory
- The commit unit applies the results in-order so the code **gives the impression that it has executed sequentially**
- This ensures the correctness of the code
- The commit unit contains a part called **the reorder buffer** which holds the result until they can be committed

Dynamic Multiple Issue: Forwarding



Dynamic Multiple Issue

- When the function unit produces a result, it forwards it to reservation stations where there might be instructions waiting on this result
- Example: a sequence of instructions with a large chain of dependence

- When new instructions arrive in the reservation stations, they may need results that are in the commit/reorder buffer
- Therefore, the commit buffer forwards the data it’s holding with the reservation stations

Dynamic Multiple Issue

- The code is issued **in-order** so as to establish the dependences between the variables as shown below

```
Code
addi        t1, s0, 40          # Line 1
lw          t2, 0(t1)           # Line 2
add         t3, t3, t2          # Line 3
or          t2, t5, t6          # Line 4
and         t3, t3, t2          # Line 5
add         a0, zero, zero      # Line 6
```

```
Read After Write (RAW) Dependencies
t1     Lines 1 & 2
t2     Lines 2 & 3
t2     Lines 4 & 5
```

Code: Read a value from the array, add it to **t3**. Then, AND the result with **(t5 OR t6).**

Dynamic Scheduling: Out-of-order Execution

- The code execute **out-of-order:**
  - operands are available → can execute
  - operands are not available → wait

```
Code
addi        t1, s0, 40
lw          t2, 0(t1)    # Cache miss
add         t3, t3, t2   # On hold
or          t2, t5, t6   # Ok to execute
and         t3, t3, t2   # On hold
add         a0, zero, zero    # Ok to execute
```

- The load is a miss, therefore, the 'add' cannot execute
- Also, the 'and' cannot execute neither (it uses the result of 'add')
- However, the 'or' and second 'add' can execute
- Out-order-order execution is called **dynamic pipeline scheduling**

Dynamic Scheduling

- The code commits **in-order** to ensure correctness

```
Code
addi        t1, s0, 40
lw          t2, 0(t1)    # Executed later
add         t3, t3, t2
or          t2, t5, t6   # Execute earlier
and         t3, t3, t2
add         a0, zero, zero
```

- The 'or' executed while the 'load' was waiting on the miss event
- However, the 'or' can't commit its result to register t2 before the 'load'
- The 'load' commits first, then the 'or' commits later
- Any instruction further on that uses t2 gets the result of 'or'

Dynamic Scheduling

- This is another example

Code

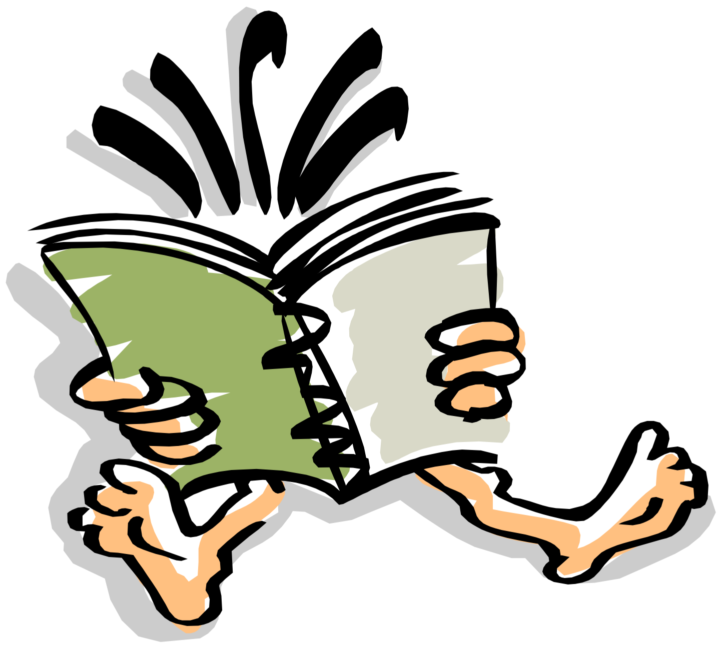
```
lw     t0,0(s0)          # Cache miss      #Commits first
add    t1,t0,t2          # On hold
sub    t3,t4,t5          # Ok to execute
and    t0,t5,t6          # Ok to execute #Commits later
...
sw     t0,0(s1)
```

- ‘lw’ experiences a cache miss; ‘add’ waits for the miss handling
- The ‘sub’ and ‘and’ can execute meanwhile
- The ‘load’ should commit before the ‘and’
- Accordingly, ‘sw’ uses the result of the ‘and’

Dynamic Multiple Issue

- Superscalar CPUs usually support hardware-based speculation
- Branch instructions are especially supported in hardware speculation
- The CPU can use dynamic branch prediction to speculate on branches
- Results computed under speculation are kept in the commit unit until it’s sure the speculation was correct; otherwise, these results are flushed

Readings



- H&P CA
  - Chapter 3.1-3.8