CCSYA

The Processor

Increasing ILP

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

How to Increase Performance

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- **Decrease the number of instructions in the program**
- **Increase the clock frequency**
- **Increase the number of instructions executed by clock cycle**

Instruction-Level Parallelism (ILP)

- **Pipelining exploits the potential parallelism among instructions**
  - This parallelism is called **instruction-level parallelism** (ILP)

- **Increasing the potential amount of instruction-level parallelism**
  - Increasing the depth of the pipeline to overlap more instructions
    - Less work per stage $\Rightarrow$ shorter clock cycle
  - Replicate the internal components of the computer so that it can launch multiple instructions in every pipeline stage
    - The general name for this technique is **multiple issue**

Limitations of Deep Pipelines

- **In the late 1990s and early 2000s, microprocessors were marketed largely based on clock frequency**
  - This pushed microprocessors to use very deep pipelines (20–31 stages on the Pentium 4) to maximize the clock frequency, even if the benefits for overall performance were questionable
- **Longer pipelines introduce more dependencies**
  - Some of the dependencies can be solved by forwarding but others require stalls, which increase the CPI
- **Adding more stages increases the cost**
  - Extra pipeline registers and hardware required to handle hazards

Multiple Issue

- **Replicate pipeline stages ⇒ multiple pipelines**
  - Start multiple instructions per clock cycle
- **CPI < 1, so use Instructions Per Cycle (IPC)**
  - E.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
  - But dependencies reduce this in practice
- **Today's high-end microprocessors attempt to issue from 3 to 6 instructions in every clock cycle**
  - Even moderate designs will aim at a peak IPC of 2

Multiple Issue

- **Static multiple issue processors**
  - Use the compiler to assist with packaging instructions and handling hazards
- **Dynamic multiple issue processors**
  - Are also known as **superscalar** processors
  - Enables the processor to execute more than one instruction per clock cycle by selecting them during execution
- **Common to static and dynamic multiple issue is the concept of speculation**

Speculation

- **"Guess" the outcome of an instruction**
  - To remove it as a dependence in executing other instructions
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing

- **Examples**
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

Compiler/Hardware Speculation

- **Compiler can reorder instructions**
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess
- **Hardware can look ahead for instructions to execute**
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

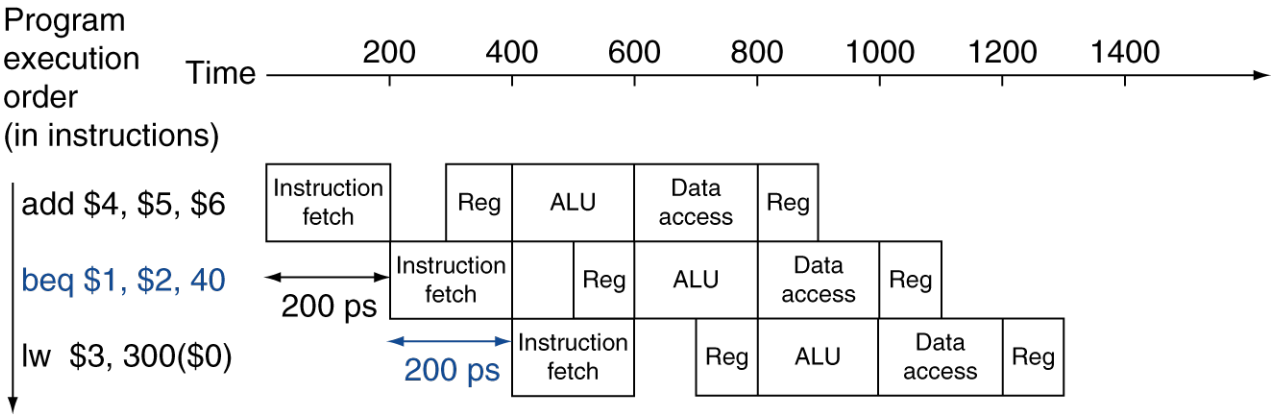
Branch Prediction

- **Predict outcome of branch and proceed from that assumption**
  - Rather than waiting to be certain of the actual outcome
  - Only stall if prediction is wrong

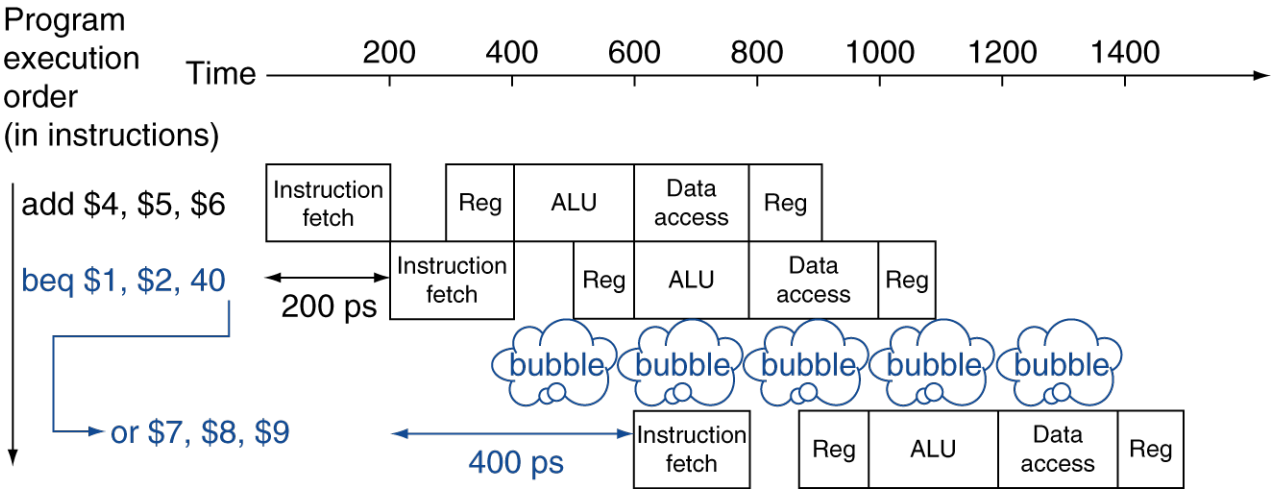
- **Assuming a branch is not taken is one simple form of branch prediction**
  - Fetch instruction after branch, with no delay
  - In that case, we predict that branches are untaken, flushing the pipeline when we are wrong
  - For the simple five-stage pipeline, coupled with compiler-based prediction, is probably adequate

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



Why branch prediction?

- **Superscalar processors with deep pipelines**
  - Intel Core 2 Duo: 14 stages
  - AMD Athlon 64: 12 stages
  - Intel Pentium 4: 31 stages

- **Many cycles before branch is resolved**
  - Wasting time if wait
  - Would be good if CPU could do some useful work

More-Realistic Branch Prediction

- **Static branch prediction (at compile time)**
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken

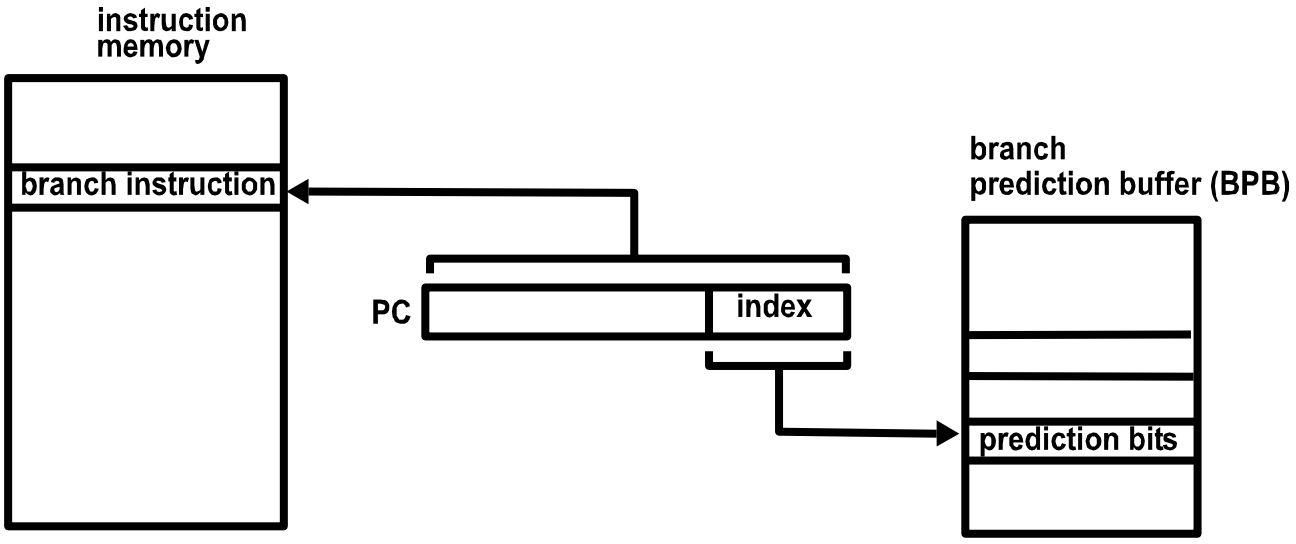
- **Dynamic branch prediction (at execution time)**
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Dynamic Branch Prediction

- **With more hardware it is possible to try to predict branch behavior during program execution**
- **Branch prediction buffer (aka branch history table)**
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
- **To execute a branch**
  - Check table, expect the same outcome
  - Start fetching from fall-through or target
  - If wrong, flush pipeline and flip prediction

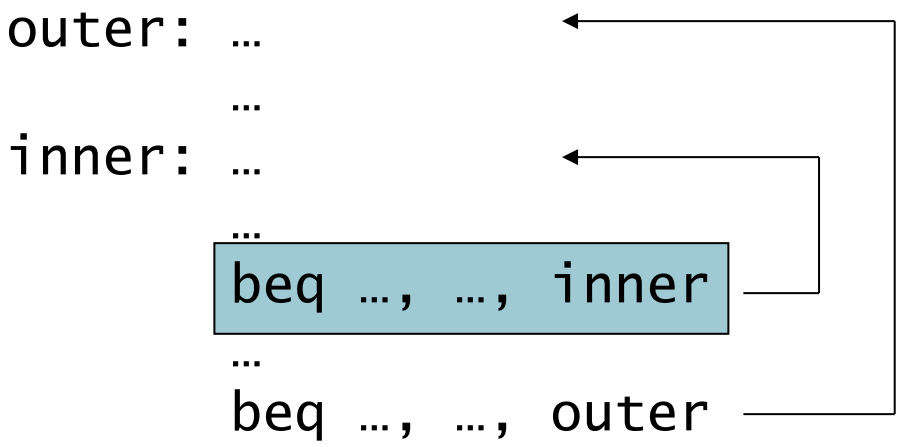
Branch Prediction Buffer

- **A small memory that is indexed by the lower portion of the address of the branch instruction**
  - Contains one or more bits indicating whether the branch was recently taken or not



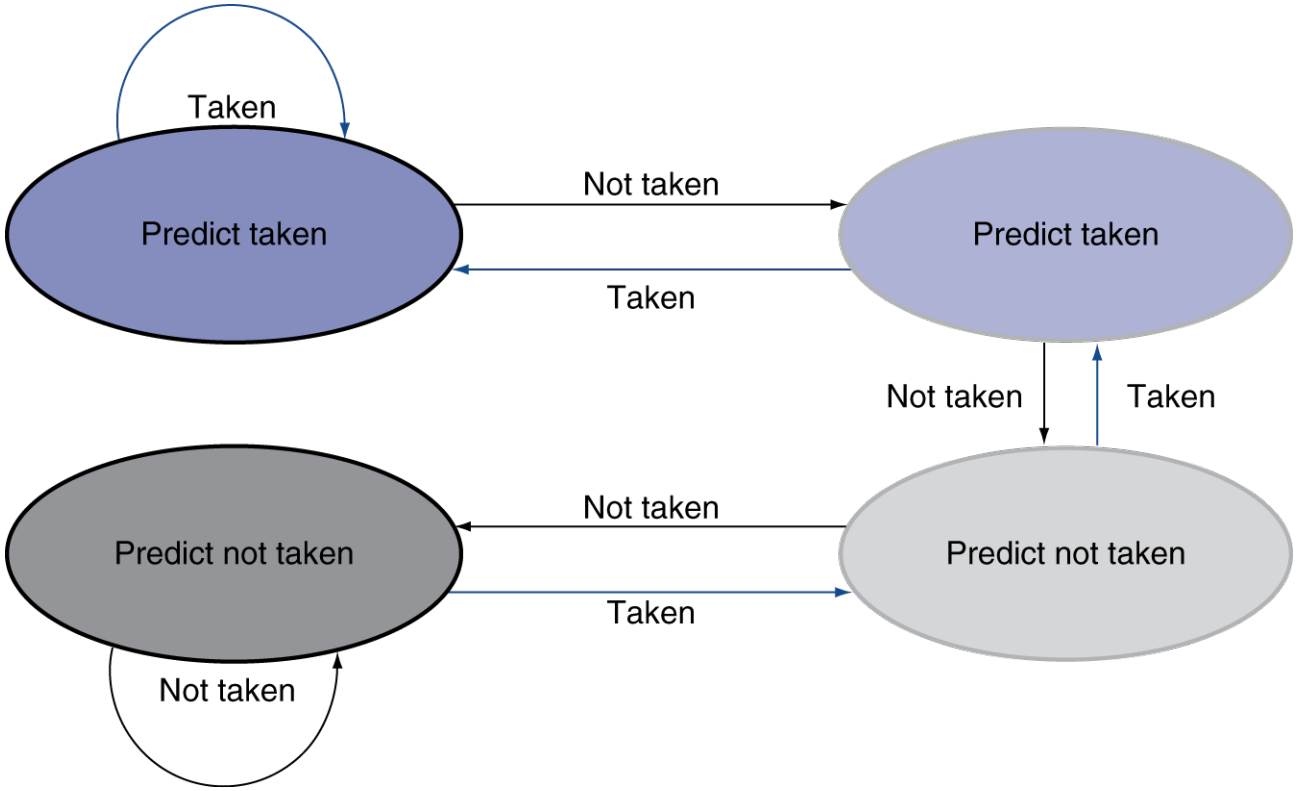
1-Bit Predictor: Shortcoming

- **Inner loop branches mispredicted twice!**
  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around



2-Bit Predictor

- **Only change prediction on two successive mispredictions**



Tournament Predictor

- **A more recent innovation in branch prediction is the use of tournament predictors**
  - Uses multiple predictors, tracking, for each branch, which predictor yields the best results
- **A typical tournament predictor might contain two predictions for each branch index**
  - One based on local information and one based on global branch behavior
- **A selector chooses which predictor to use for any given prediction**

Calculating the Branch Target

- **Still need to calculate the target address**
- **Branch prediction buffers do not supply the target PC value**
  - A **Branch Target Buffer** does this

- **Cache of target addresses**
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

Static Multiple Issue

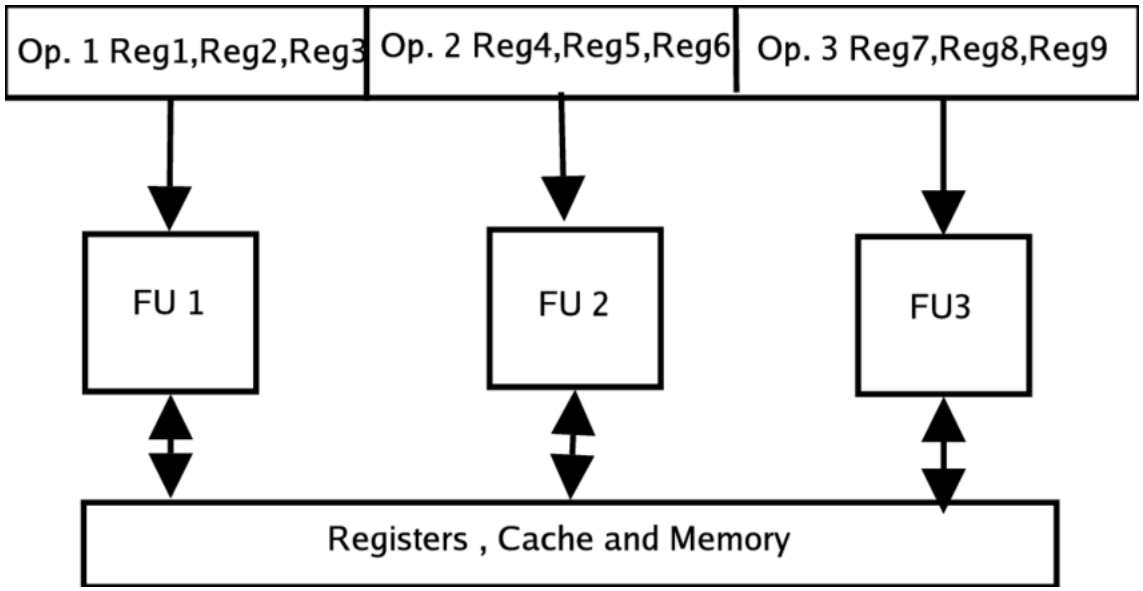
- **Compiler groups instructions into issue packets**
  - Group of instructions that can be issued on a single cycle
  - Determined by which pipeline resources are required

- **Compiler must remove some/all hazards**
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with `nop` if necessary

- **Complexity is on the compiler**
  - Follows the RISC philosophy

Very Long Instruction Word

- **Think of an issue packet as a very long instruction**
  - Specifies multiple concurrent operations ⇒ **Very Long Instruction Word** (VLIW)
    - Typically, with many separate opcode fields
    - The layout of simultaneously issuing instructions is restricted to simplify the decoding and instruction issue



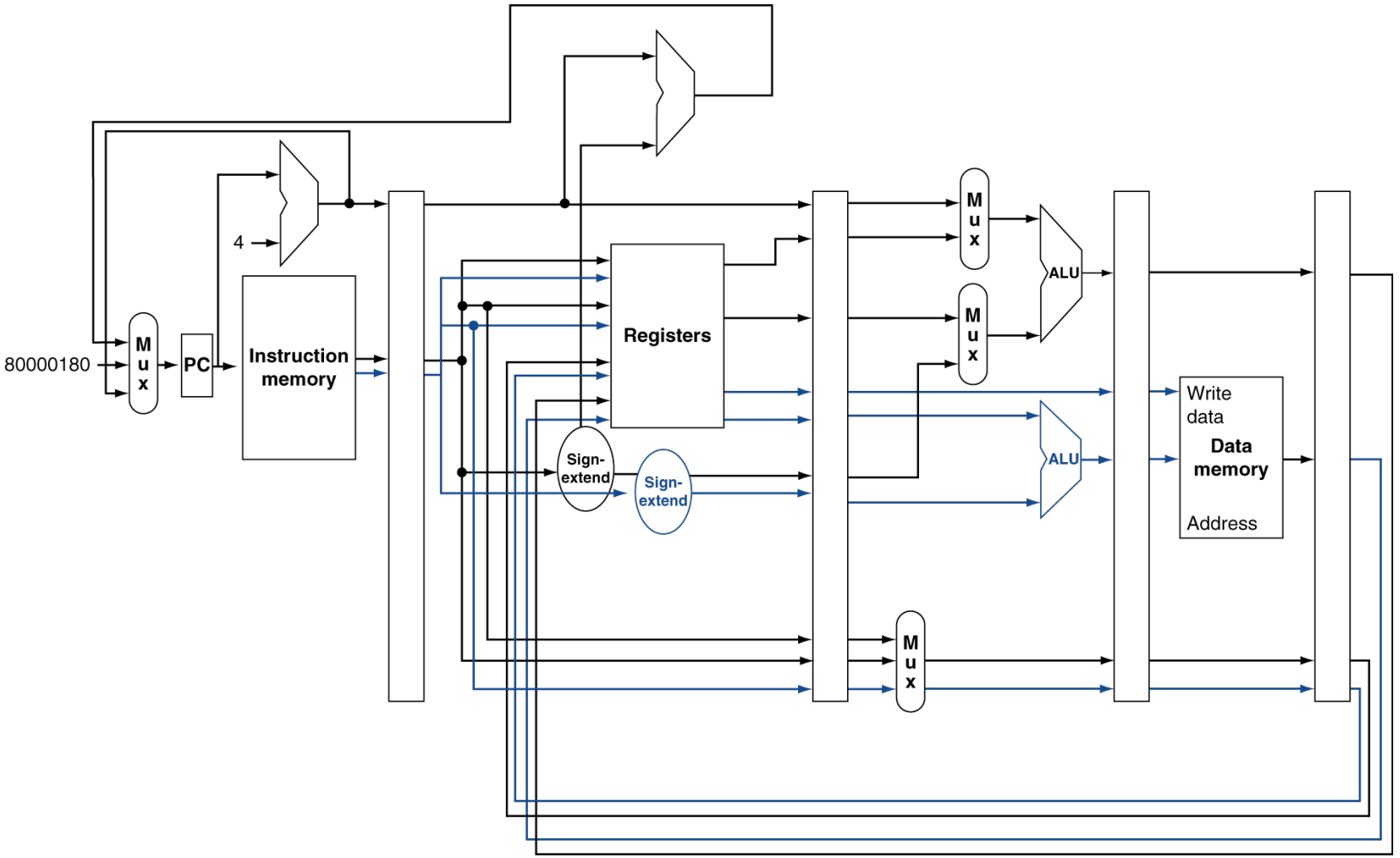
MIPS with Static Dual Issue

- **Consider a simple two-issue packet**
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with `nop`

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

MIPS with Static Dual Issue

- **Without these extra resources, the two-issue pipeline would be hindered by structural hazards**



Hazards in the Dual-Issue MIPS

- **More instructions executing in parallel**
- **EX data hazard**
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet
    - ```
      add  $t0, $s0, $s1
      load $s2, 0($t0)
      ```
    - Split into two packets, effectively a stall
- **Load-use hazard**
  - Still one cycle use latency, but now two instructions
- **More aggressive scheduling is required**
      ```

Scheduling Example

- **How can this loop be scheduled on a static dual-issue MIPS?**

```
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
```

- **Reorder the instructions to avoid as many pipeline stalls as possible**
  - Assume branches are predicted, so that control hazards are handled by the hardware

Scheduling Example

- **Schedule for dual-issue MIPS:**

```
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

- **IPC = 5/4 = 1.25 (c.f. peak IPC = 2)**
  - We do not count any `nops` executed as useful instructions

Loop Unrolling

- **Replicate loop body to expose more parallelism**
  - Reduces loop-control overhead
  - Applied particularly in loops that access arrays

- **Use different registers per replication**
  - Called "**register renaming**"
  - Avoid loop-carried "anti-dependencies"
    - Store followed by a load of the same register
    - Aka "name dependence"
      - Reuse of a register name

Loop Unrolling Example

	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

- **IPC = 14/8 = 1.75**
  - Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

- **“Superscalar” processors**
- **CPU decides whether to issue 0, 1, 2, … each cycle**
  - Avoiding structural and data hazards
- **Avoids the need for compiler scheduling**
  - Though it may still help
  - Code semantics ensured by the CPU

Dynamic Pipeline Scheduling

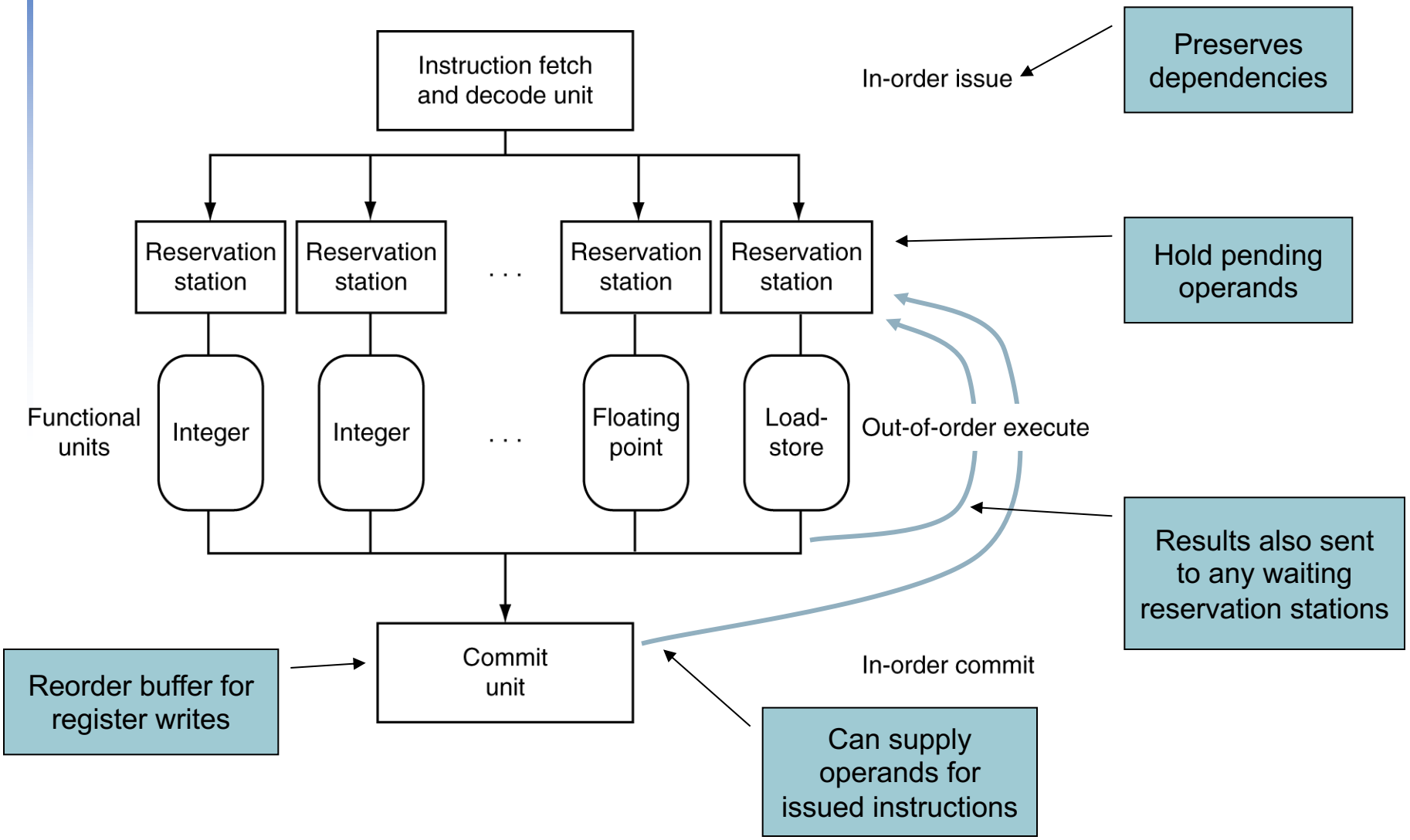
- **Allow the CPU to execute instructions out of order to avoid stalls**
  - But commit result to registers in order

- **A simple example of avoiding a data hazard**

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

  - Can start sub while addu is waiting for lw

Dynamically Scheduled CPU



Register Renaming

- **Reservation stations and reorder buffer effectively provide register renaming**

- **On instruction issue to reservation station**
  - If operand is available in register file or reorder buffer
    - Copied to reservation station
    - No longer required in the register; can be overwritten
  - If operand is not yet available
    - It will be provided to the reservation station by a function unit
    - Register update may not be required

Speculation

- **Dynamic scheduling is often extended by including hardware-based speculation**
- **Predict branch and continue issuing**
  - Don’t commit until branch outcome determined
- **Load speculation**
  - Avoid load and cache miss delay
    - Predict the effective address
    - Predict loaded value
    - Load before completing outstanding stores
    - Bypass stored values to load unit
  - Don’t commit load until speculation cleared

Why Do Dynamic Scheduling?

- **Why not just let the compiler schedule code?**
- **Not all stalls are predictable**
  - e.g., cache misses
- **Can't always schedule around branches**
  - Branch outcome is dynamically determined
- **Different implementations of an ISA have different latencies and hazards**

Does Multiple Issue Work?

- **Yes, but not as much as we'd like**
- **Programs have real dependencies that limit ILP**
  - Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- **Some parallelism is hard to expose**
  - Limited window size during instruction issue
- **Memory delays and limited bandwidth**
  - Hard to keep pipelines full
- **Speculation can help if done well**

Power Efficiency

- **Complexity of dynamic scheduling and speculations requires power**
  - Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103 W
Intel Core	2006	3000 MHz	14	4	Yes	2	75 W
Intel Core i7 Nehalem	2008	3600 MHz	14	4	Yes	2-4	87 W
Intel Core Westmere	2010	3730 MHz	14	4	Yes	6	130 W
Intel Core i7 Ivy Bridge	2012	3400 MHz	14	4	Yes	6	130 W
Intel Core Broadwell	2014	3700 MHz	14	4	Yes	10	140 W
Intel Core i9 Skylake	2016	3100 MHz	14	4	Yes	14	165 W
Intel Ice Lake	2018	4200 MHz	14	4	Yes	16	185 W

Cortex A53 and Intel i7

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-2048 KiB	256 KiB (per core)
3rd level caches (shared)	(platform dependent)	2-8 MB

Concluding Remarks

- **Pipelining improves instruction throughput using parallelism**
  - More instructions completed per second
  - Latency for each instruction not reduced
- **Hazards: structural, data, control**
- **Multiple issue and dynamic scheduling (ILP)**
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall