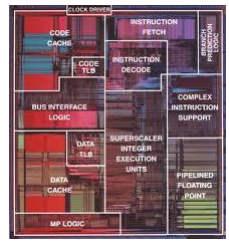
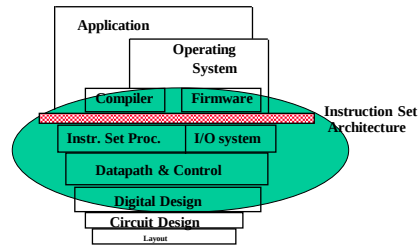
CS/SE 3340
Computer Architecture



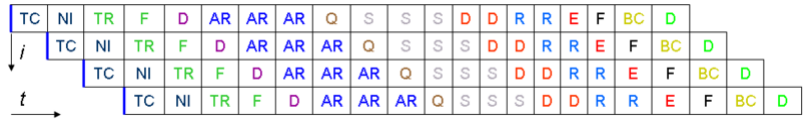


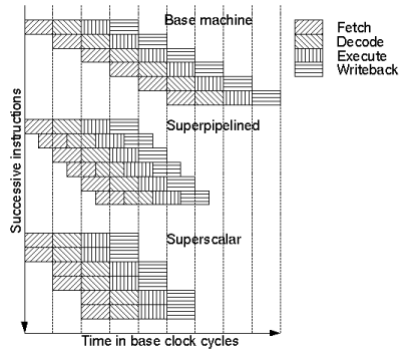
Advanced Instruction Level Parallelism

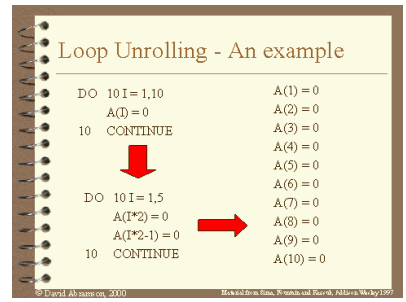
Adapted from slides by Profs. D. Patterson and J. Hennessey

Questions

1. How to further increase processor's performance
2. What is a super-pipelined/superscalar processor
3. What is execution speculation and how it is done?
4. How does MIPS exploit spatial parallelism?
5. What is loop unrolling
6. What/Why/How of dynamic pipeline scheduling?







3

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel – exploiting *temporal parallelism*
- To increase ILP
  - Deeper pipeline (e.g. *Intel* processors) – *super-pipeline*
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue – exploiting *spatial parallelism*
    - Replicate pipeline stages ⇒ *multiple pipelines*
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC) as performance metric
    - e.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But in practice dependencies reduce performance

4

Multiple Issue

- *Static multiple issue*
  - Compiler groups instructions to be issued together
  - Packages them into "issue slots"
  - Compiler detects and avoids hazards
- *Dynamic multiple issue*
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

5

Speculation

- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll-back if path taken is different
  - Speculate on load
    - Roll-back if location is updated

6

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

7

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
- Static speculation
  - Can add ISA support for deferring exceptions
- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)

8

Static Multiple Issue

- Compiler groups instructions into *"issue packets"*
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an *issue packet* as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ *Very Long Instruction Word (VLIW)*

9

Scheduling Static Multiple Issue

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

10

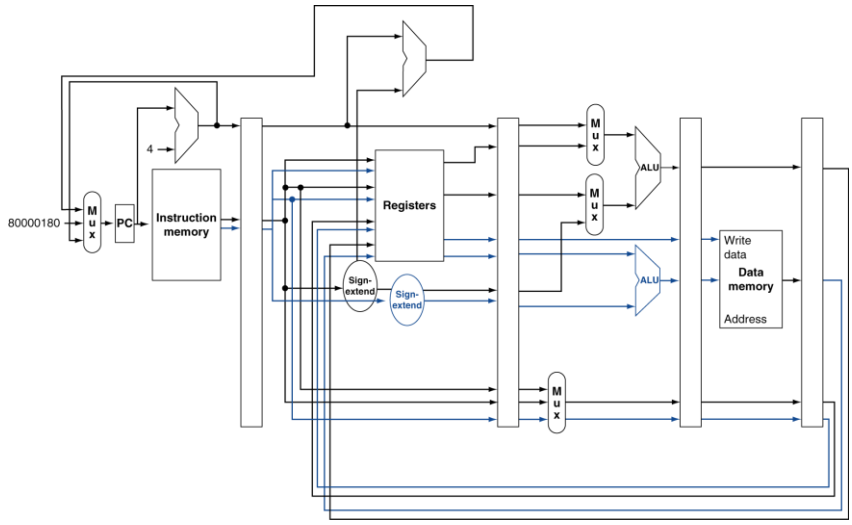
MIPS with Static Dual Issue

- *Two-issue packets*
  - 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

11

MIPS with Static Dual Issue



12

Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
  - Possibility of hazards increases!
- EX stage 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 required

Scheduling Example

- Schedule this 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
	addu $t0, $t0, $s2	nop	2
	addi $s1, $s1,-4	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called "*register renaming*"
  - Avoid loop-carried "anti-dependencies"
    - Store followed by a load of the same register
    - a.k.a. "name dependence"
      - Reuse of a register name

15

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, $t3, $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

16

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

17

Dynamic Pipeline Scheduling

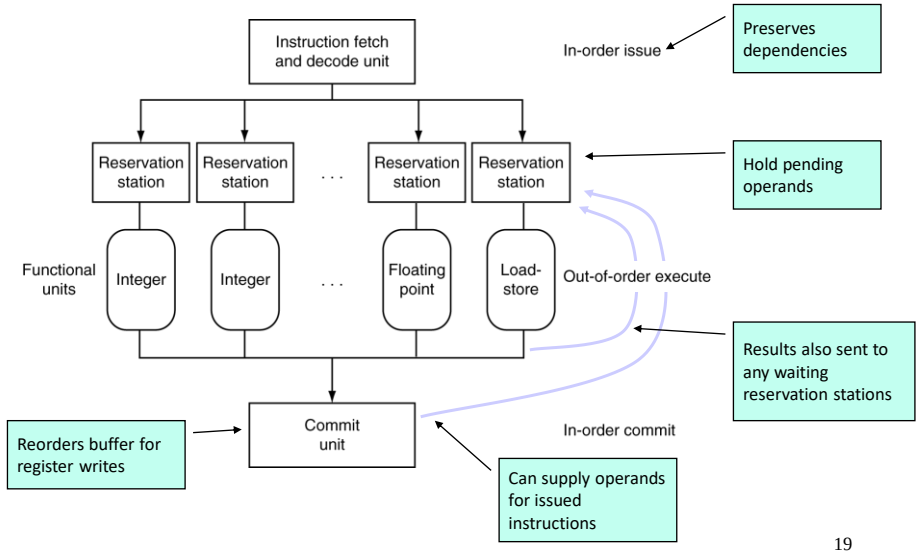
- Allow the CPU to execute instructions *out of order* to avoid stalls
  - But commit result to registers in order
- Example

```
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`

18

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

- 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

21

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

22

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

23

Summary

- To improve processor performance further with pipelining: increase Instruction-Level Parallelism (ILP)
  - Deeper pipeline –> *super pipeline*
  - Multiple pipeline –> *multiple issue*
- Multiple issue
  - Static
  - Dynamic (superscalar)
- Rely heavily on *compiler technologies* to deal with pipeline hazards

24