Lecture 10
Instruction-Level Parallelism

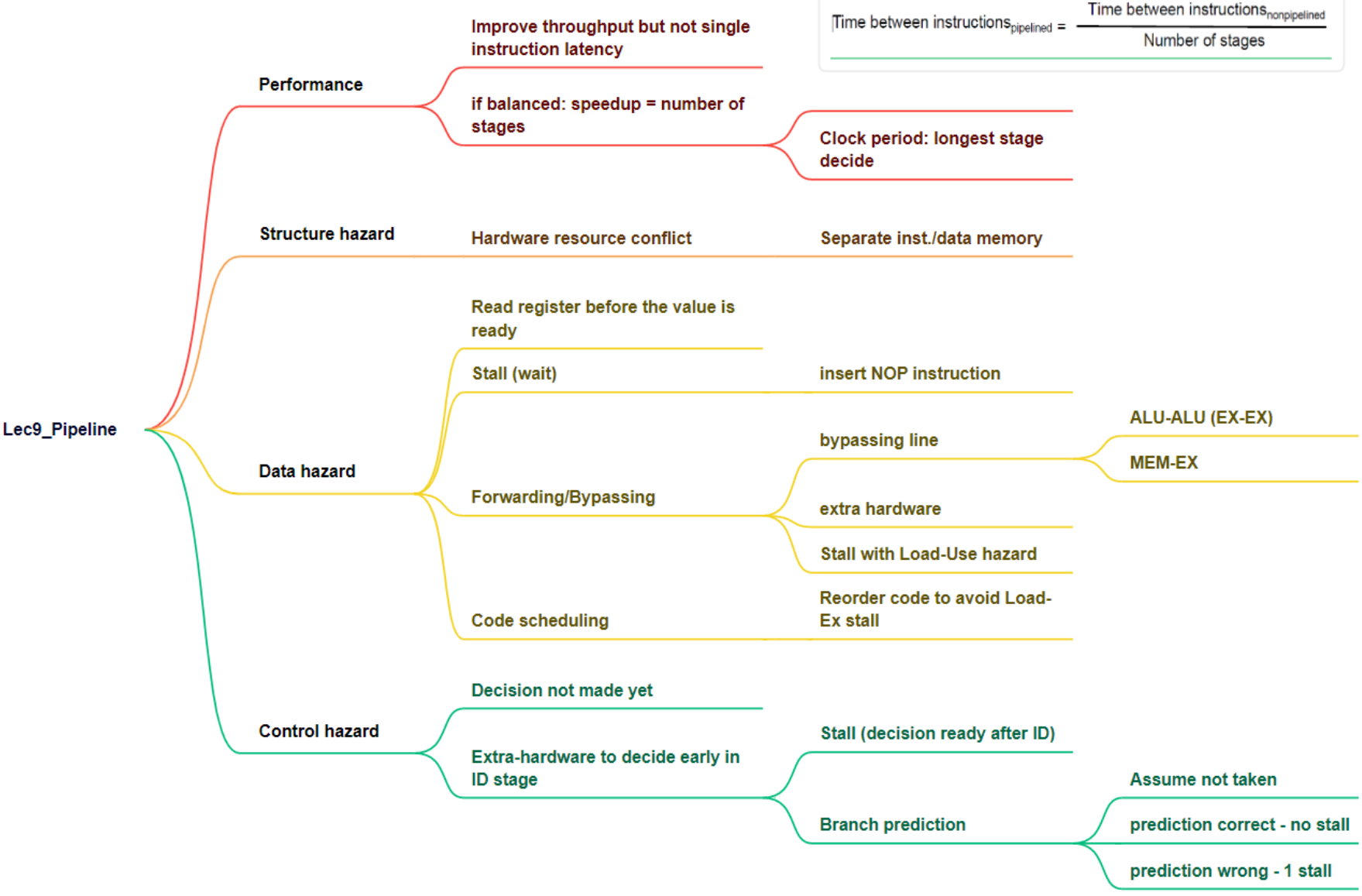
CS202 2023 Spring

Today's Agenda

- Recap
  - Pipeline Performance
  - Pipeline Hazard
- Context
  - Instruction-level parallelism
  - Multiple issues
  - Scheduling
- Reading: Textbook 4.10-4.11, 4.14
- Self-study: The rest sections of Chapter 3

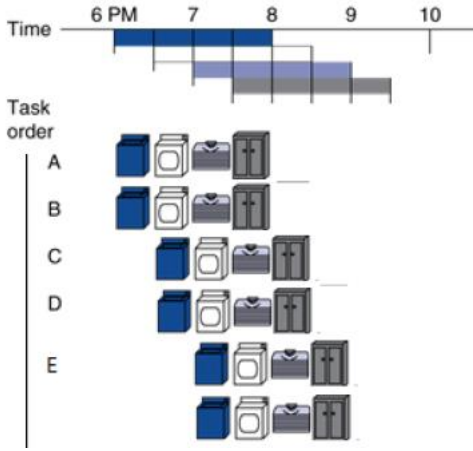
Recap

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$



Instruction-Level Parallelism (ILP)

- Instruction-level parallelism: parallelism among instructions
  - Pipelining is one type of ILP: because pipeline executes multiple
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - 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



Multiple Issue

编译器把可以同时取到的指令，知道他们是否可以一起打包放到issue slots,此过程是由
编译器决定的，可以避免hazards
在执行之前，就已经决定了

- **Static** multiple issue (Compilers decide)
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- **Dynamic** multiple issue (Hardware decide)
  - 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

cpu决定的，不按照顺序执行，能执行那个执行那个，乱序执行，但是执行完后不会立刻放进寄存器

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)

相当于在一个clock里面，放置一个64bit的指令，由两个32bit的指令组成

Scheduling Static Multiple Issue

- 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

packet之间不能有依赖关系
同时执行的指令不能有依赖关系

MIPS with Static Dual Issue

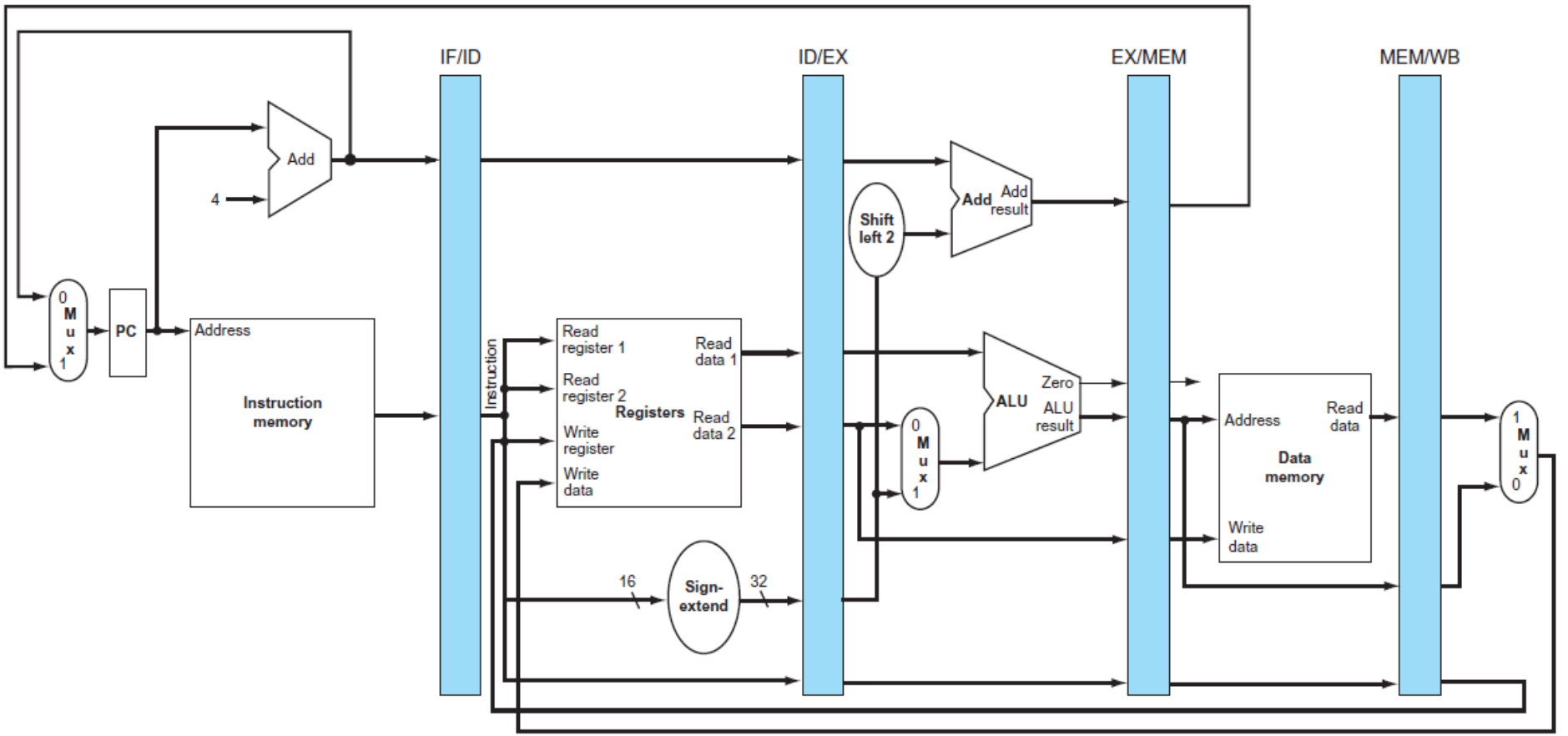
- Two-issue packets
  - Divide instructions into two types:
    - Type1: ALU/branch instruction
    - Type2: load/store instruction
  - During each cycle, execute a type1/type2 inst. simultaneously
  - 64-bit aligned instructions
    - Pad an unused instruction with nop

一个packet里面，只能是alu/branch+la/sw
也就是每类各一个

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

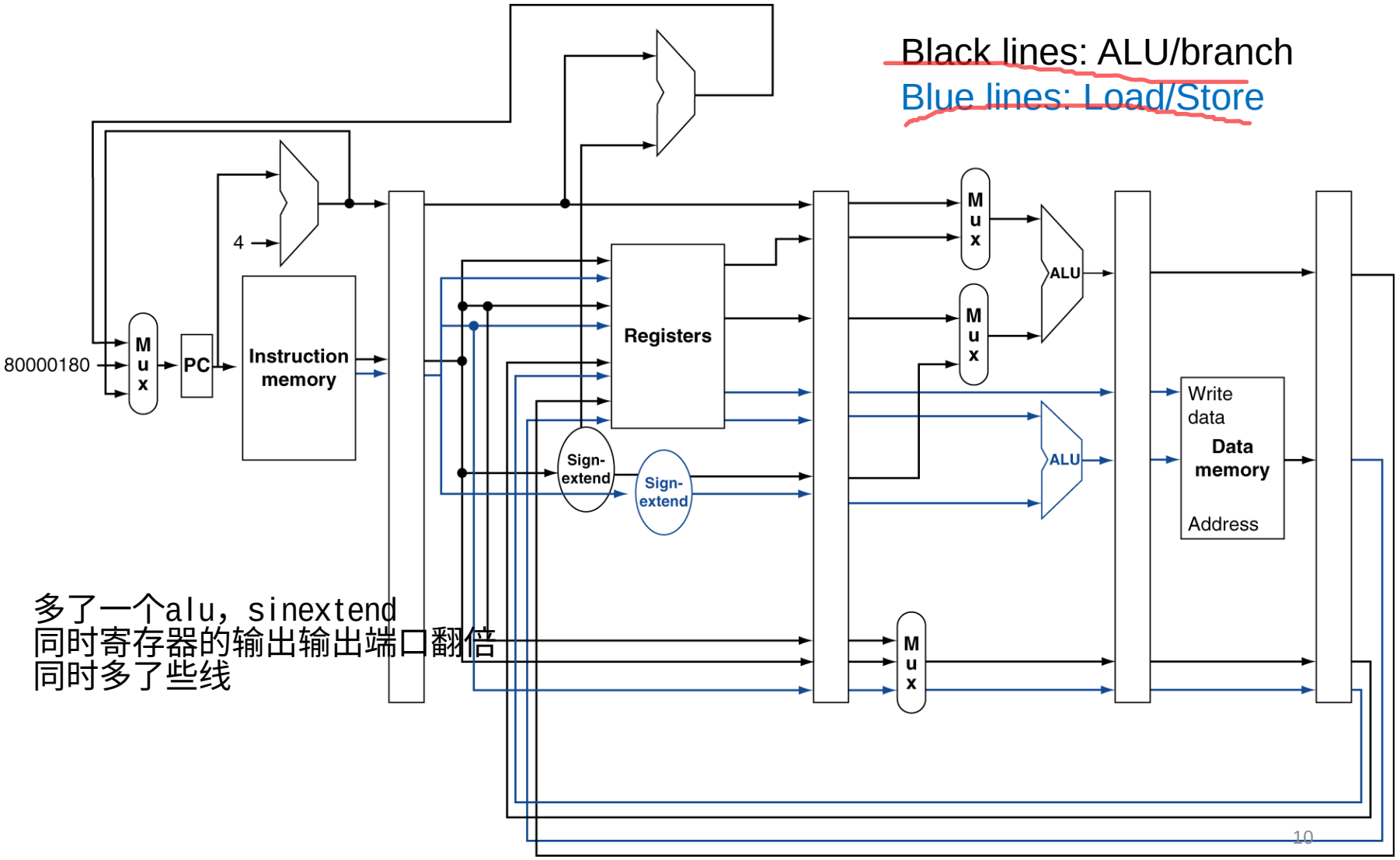
A Basic MIPS Pipeline Datapath

把要做的事先缓存进来



MIPS with Static Dual Issue

硬件上的改动



多了一个alu，sinextend
同时寄存器的输出输出端口翻倍
同时多了些线

Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- Removing hazards in static multiple issues
  - code scheduling and no-op insertion.
  - detect data hazard and generate all stalls between two issue packets, compiler avoids all dependencies within an instruction pair.
- 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(number of clock cycles between load and use), 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:		lw $t0, 0($s1)	1
			2
			3
			4

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:		lw $t0, 0($s1)	1
			2
	addu $t0, $t0, $s2		3
			4

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:		lw $t0, 0($s1)	1
			2
	addu $t0, $t0, $s2		3
		sw $t0, 0($s1)	4

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:		lw $t0, 0($s1)	1
	addi $s1, $s1,-4		2
	addu $t0, $t0, $s2		3
		sw $t0, 4($s1)	4

bring addi to cycle 2 to avoid stall, but need to adjust sw offset

稍微改动下指令，先-4，再给他偏移回去

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:		lw $t0, 0($s1)	1
	addi $s1, $s1,-4		2
	addu $t0, $t0, $s2		3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

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

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
	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)

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
    - Aka “name dependence” (Reuse of a register name)

Many anti-dependencies

```
for (i=4; i > 0;i--)
a[i]=a[i]+5;
```

```
a[4]=a[4]+5;
a[3]=a[3]+5;
a[2]=a[2]+5;
a[1]=a[1]+5;
```

```
lw   $t0, 0($s1)
addu $t0, $t0, $s2
sw   $t0, 0($s1)
addi $s1, $s1,-4

lw   $t0, 0($s1)
addu $t0, $t0, $s2
sw   $t0, 0($s1)
addi $s1, $s1,-4
```

a[4]=a[4]+5;

a[3]=a[3]+5;

Loop Unrolling Example

- Repeat the code in the 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)

	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

增添的

因为先做了s1的减法

Loop Unrolling Example

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

	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

Dynamic Multiple Issue

- "Superscalar" processors
  - An advanced pipelining techniques that enables the processor to execute more than one instruction per clock cycle **by selecting them during execution**.
- 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

不需要compiler做任何事，只需要硬件资源

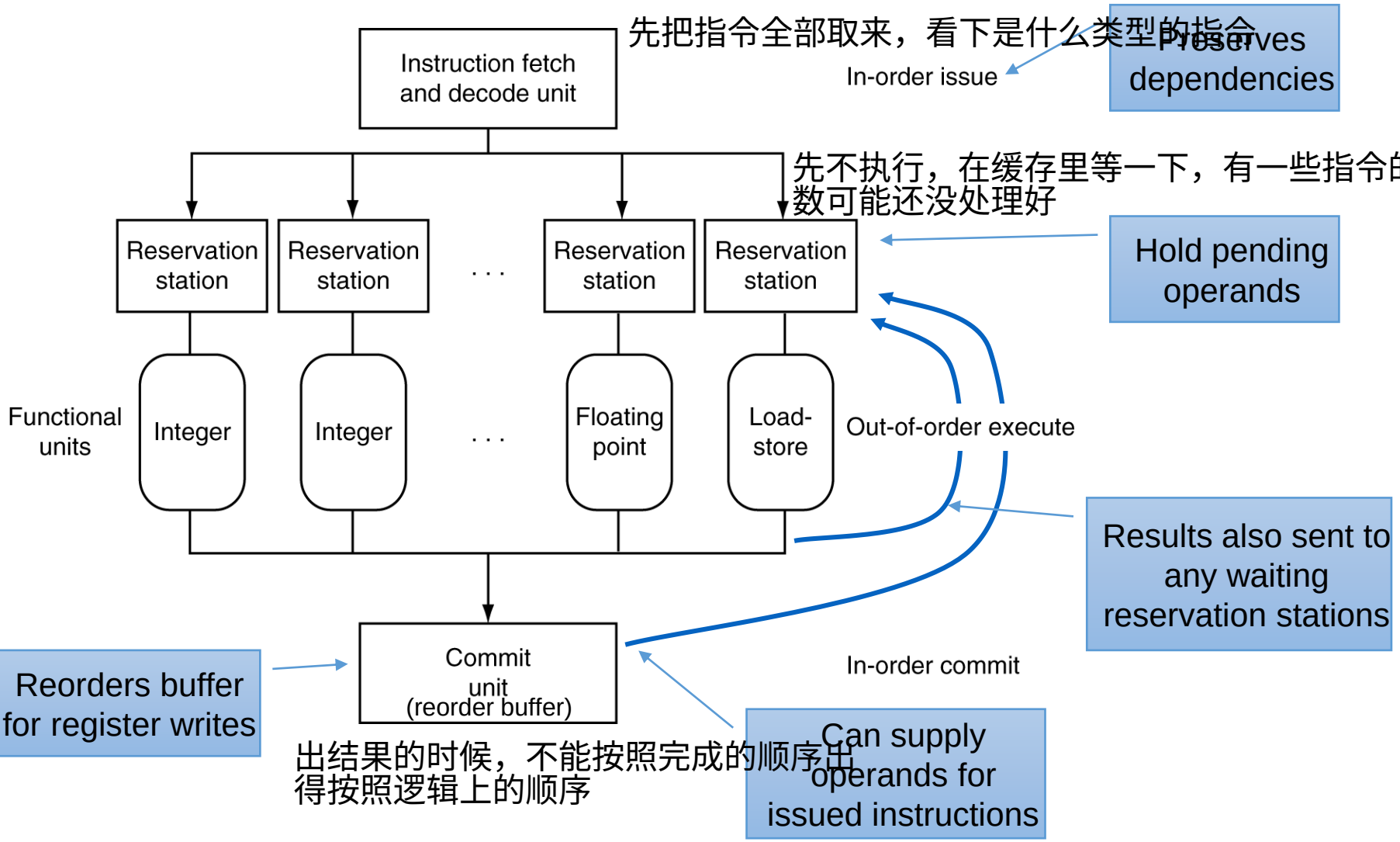
Dynamic Pipeline Scheduling

- Hardware support for reordering the order of instruction execution
- 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

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

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

Speculation for Static/Dynamic Multiple Issue

- "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
    - Avoid load and cache miss delay
    - Roll back if location is updated

Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move an instruction across a branch or a load across a store
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results and write to register or memory until it determines they are actually needed
  - Flush buffers on incorrect speculation

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)

Summary

- Techniques
  - Static Multiple issue
  - Dynamic Multiple issue
  - VLIW
  - Superscalar
  - Loop unrolling
  - Register renaming
  - Dynamic scheduling
  - Out-of-order execution
  - Speculation

	Static multiple issue	**Dynamic multiple issue**
Decision made by	Compiler (software)	Processor (hardware)
Also called	Very long instruction word (VLIW)	Superscalar
Ways to remove hazard	Loop unrolling Register renaming Speculation	Dynamic scheduling Out-of-order execution Register renaming Speculation

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

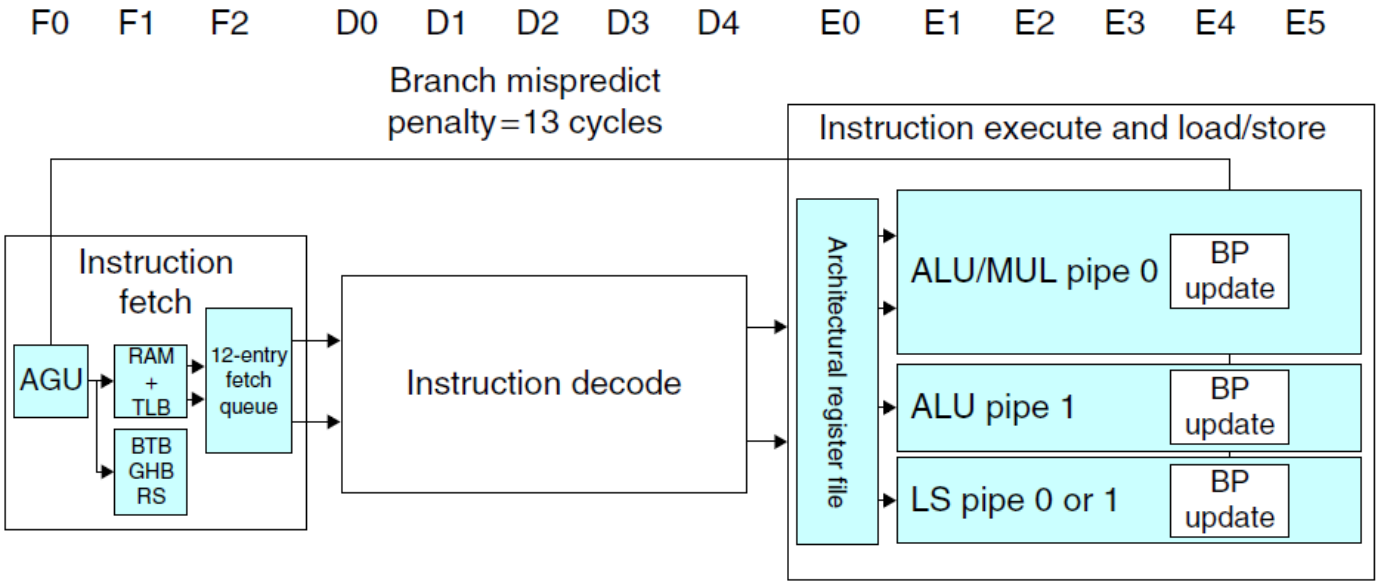
- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

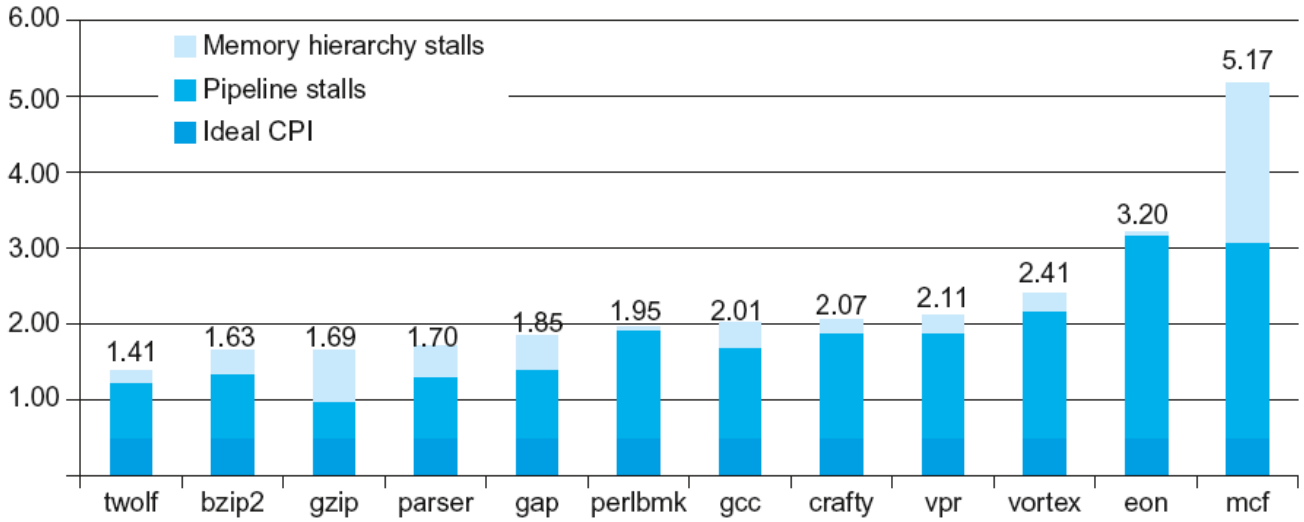
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

Cortex A8 and Intel i7

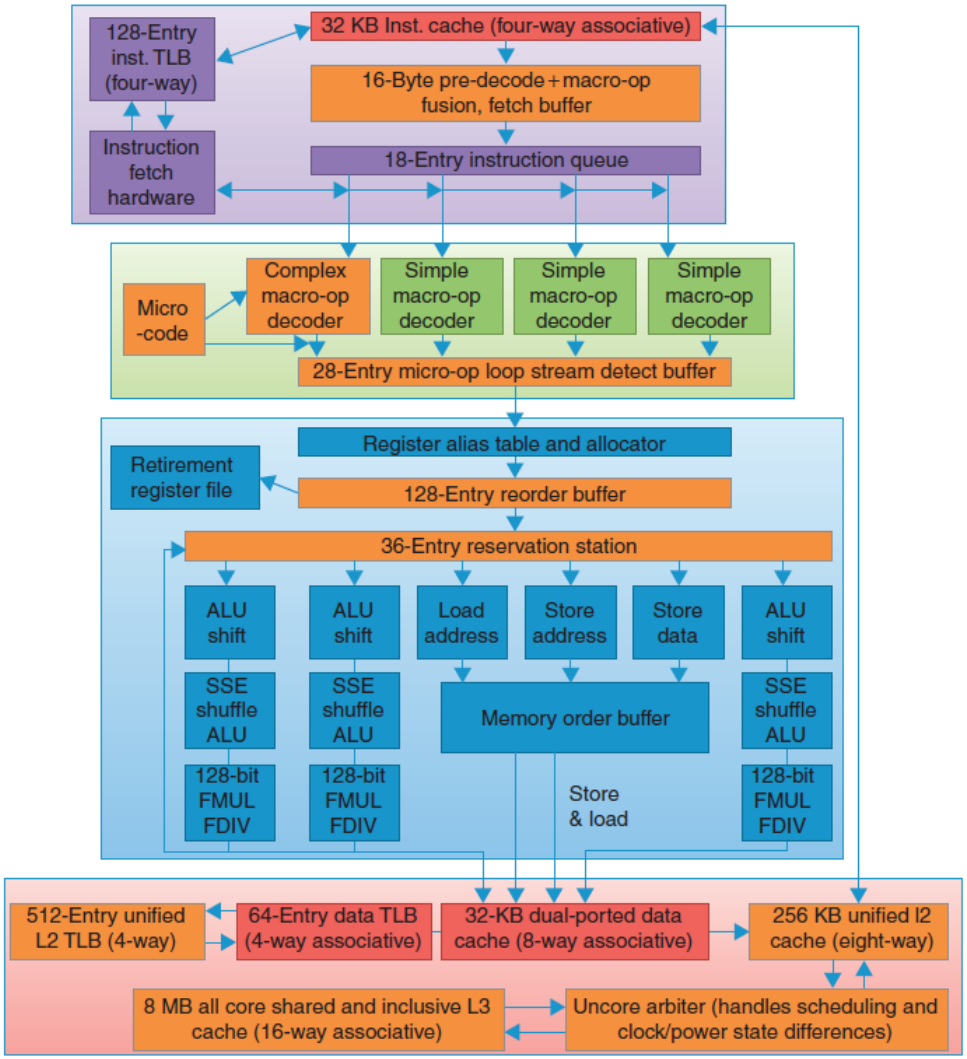
Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	14	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	2-level	2-level
1st level caches/core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-1024 KiB	256 KiB
3rd level caches (shared)	-	2- 8 MB

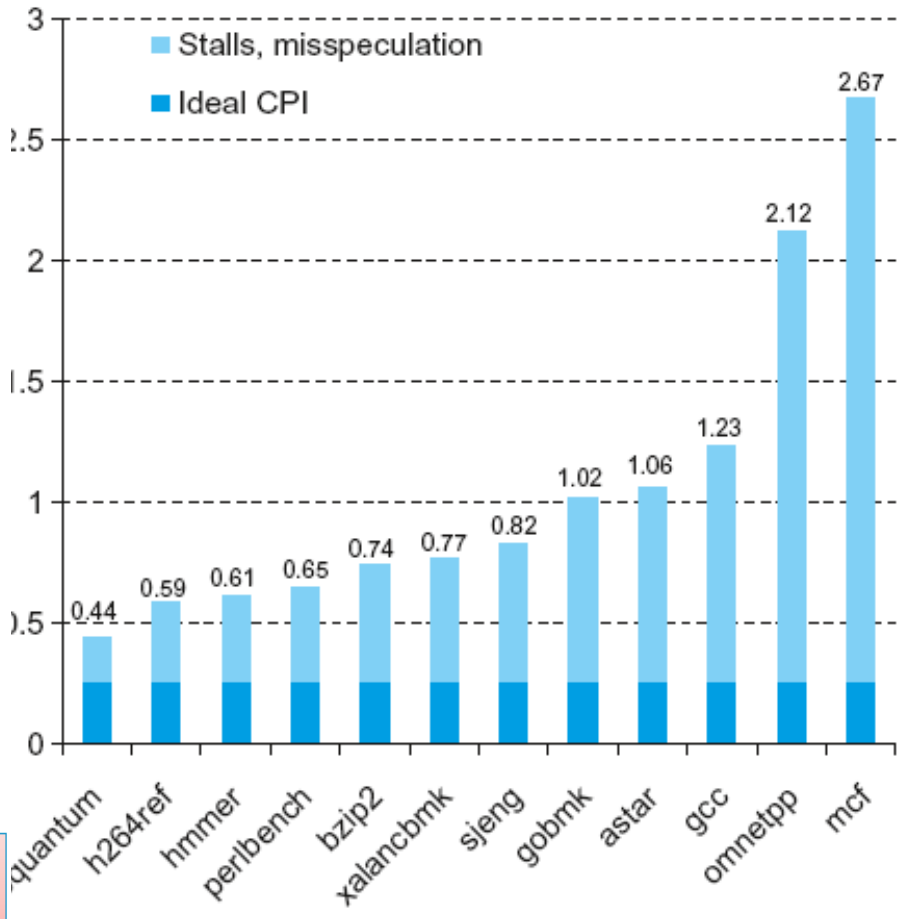
ARM Cortex-A8 Pipeline & Performance





Core i7 Pipeline & Performance





Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends

Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- 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