Section 4.9

Exception

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180

An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - …: C000 0040
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …

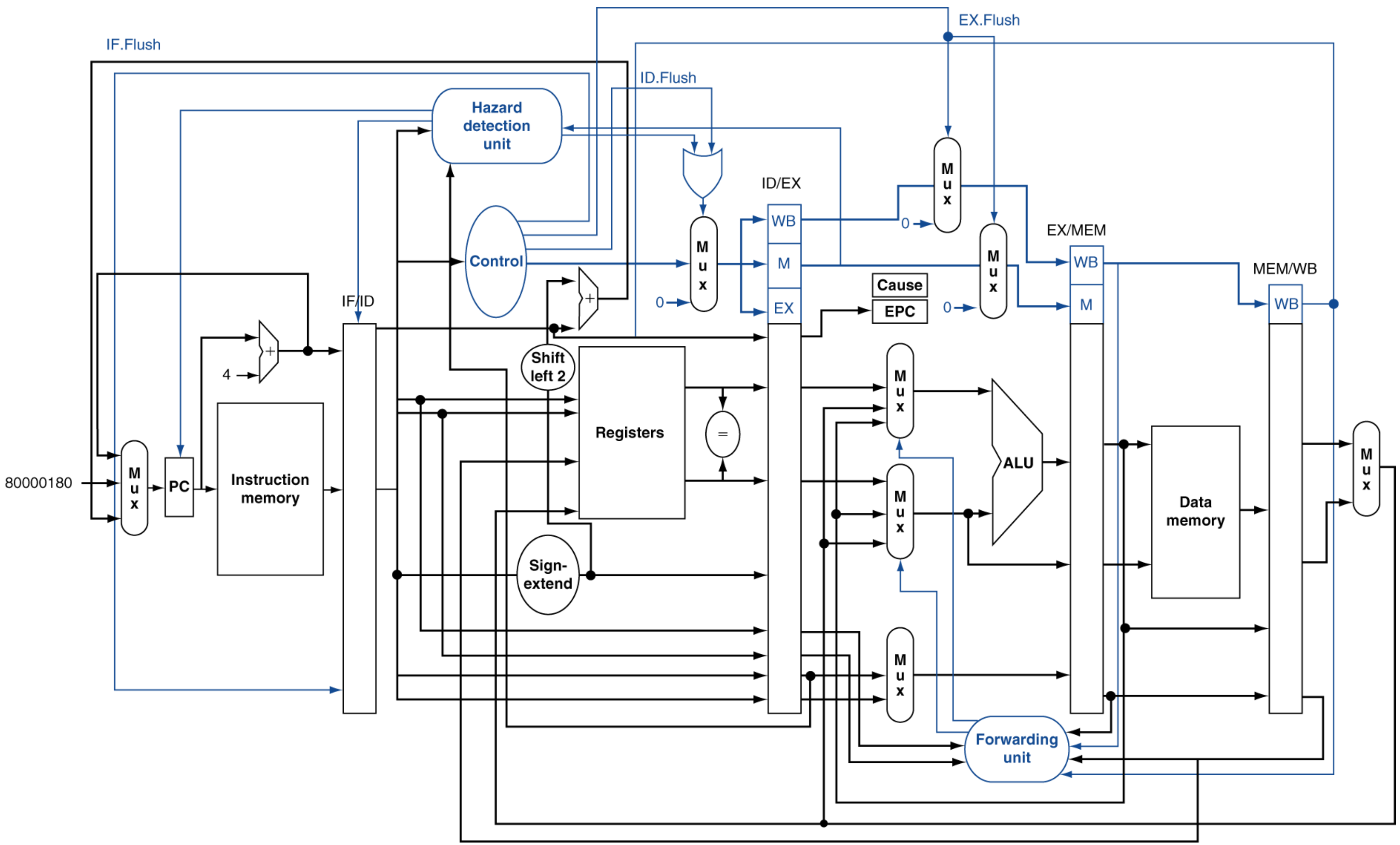
Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage

```
add $1, $2, $1
```

  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions



Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust

Exception Example

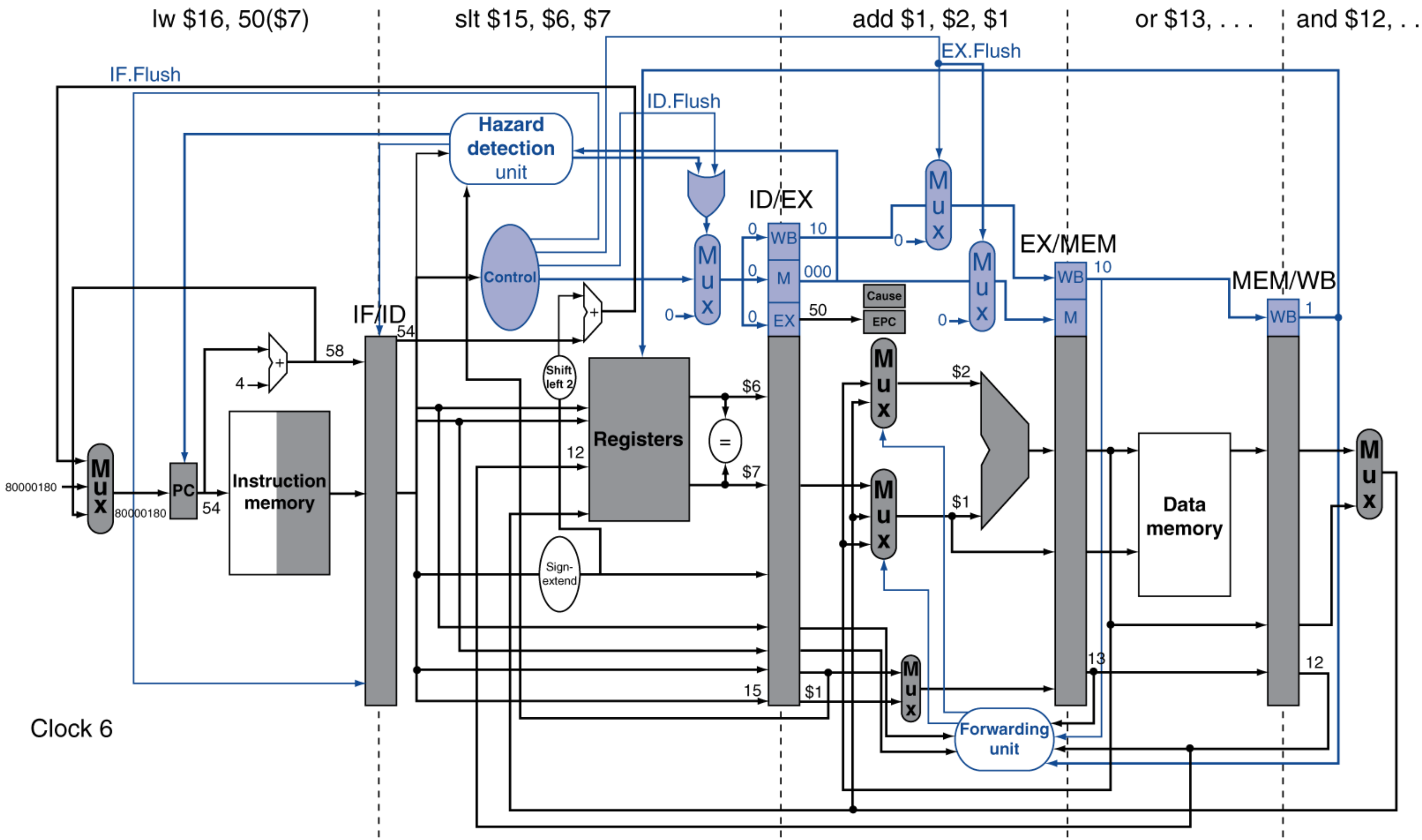
- Exception on add in

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

- Handler

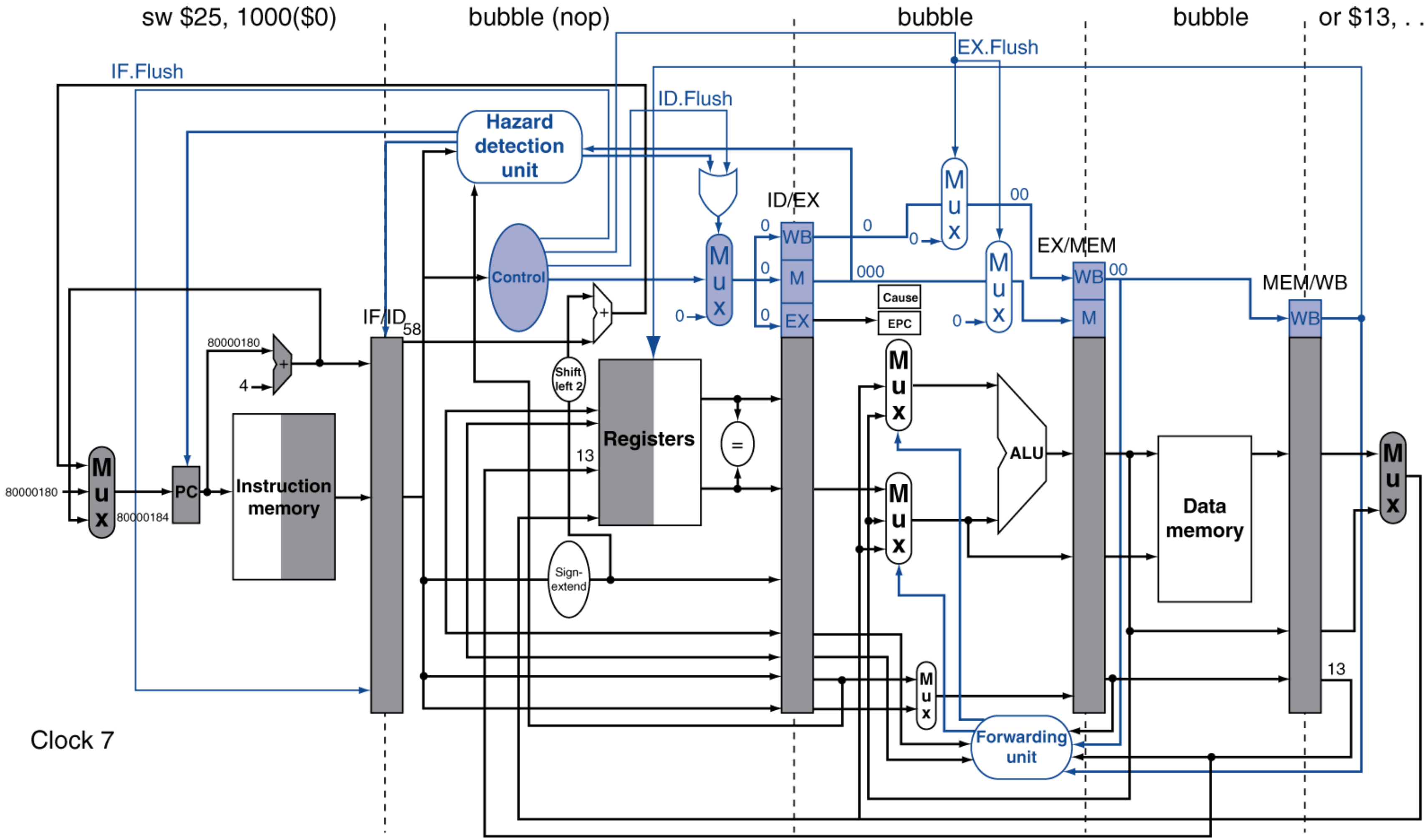
```
80000180   sw   $25, 1000($0)
80000184   sw   $26, 1004($0)
…
```

Exception Example



Clock 6

Exception Example



Clock 7

Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

Section 4.10

Parallelism via Instructions

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage $\Rightarrow$ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages $\Rightarrow$ 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

- 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

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

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

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)

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
  - $\Rightarrow$ Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

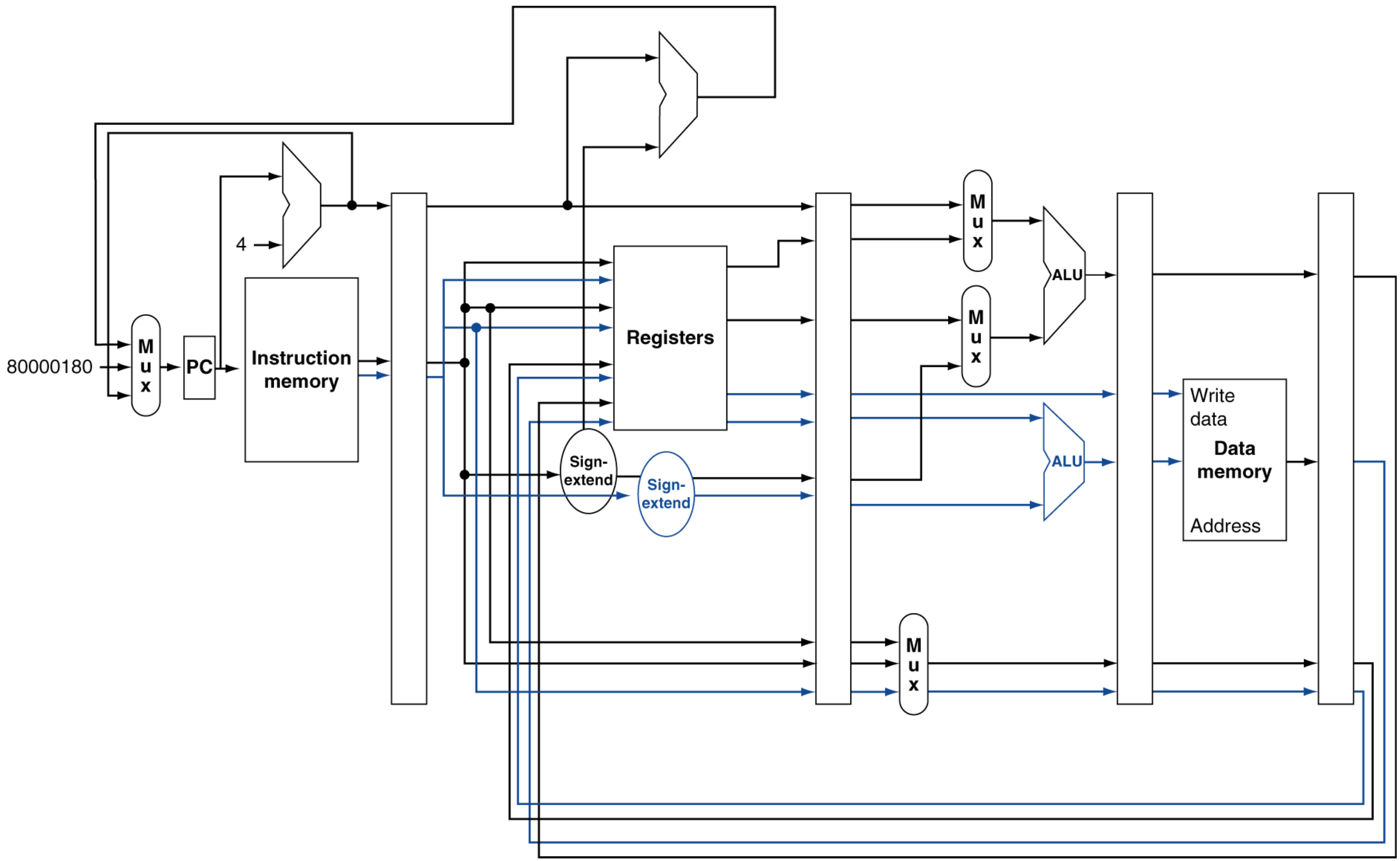
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

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

MIPS with Static Dual Issue



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 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
	`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

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

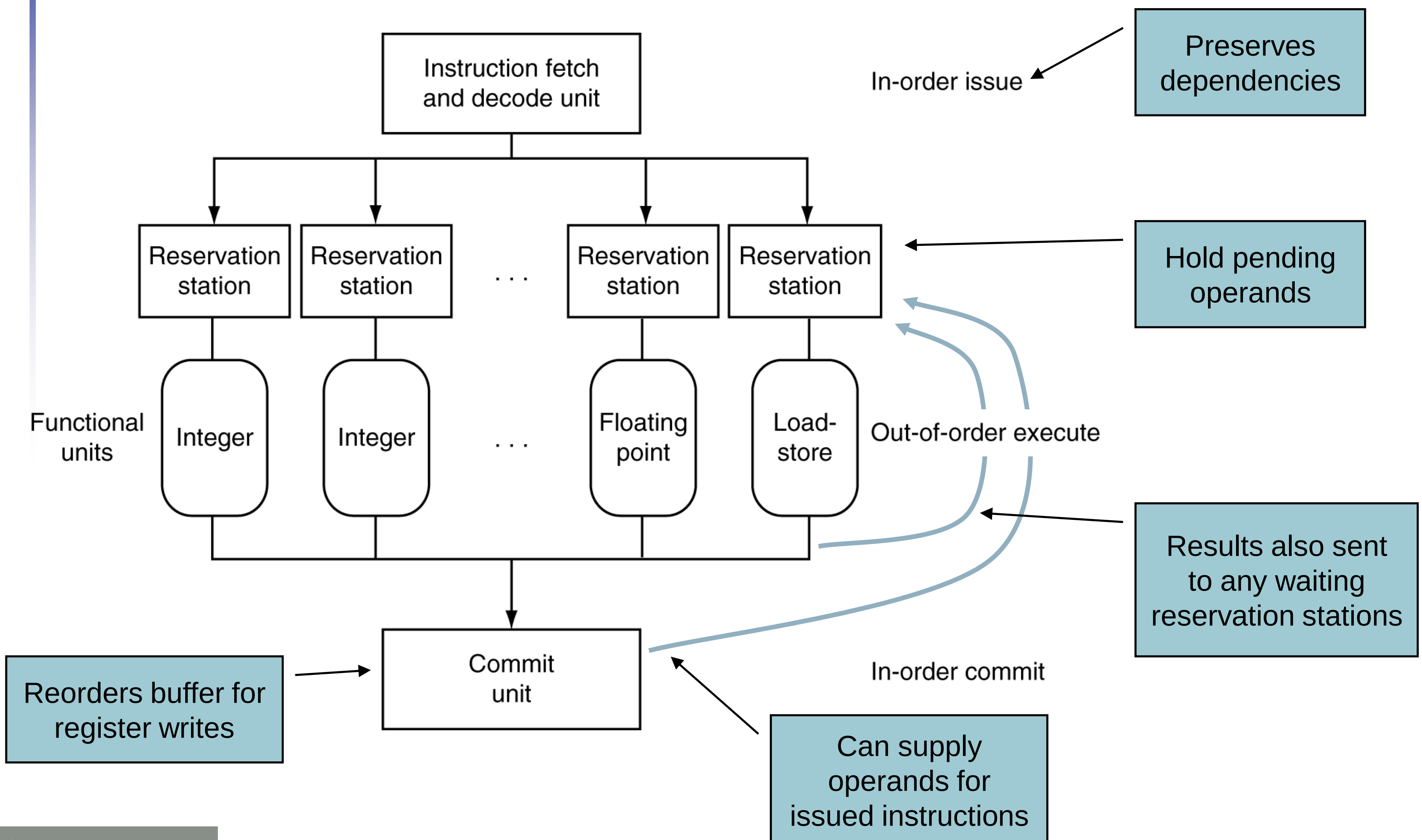
Dynamic Pipeline Scheduling

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

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 predicable
  - 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?

The BIG Picture

- 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

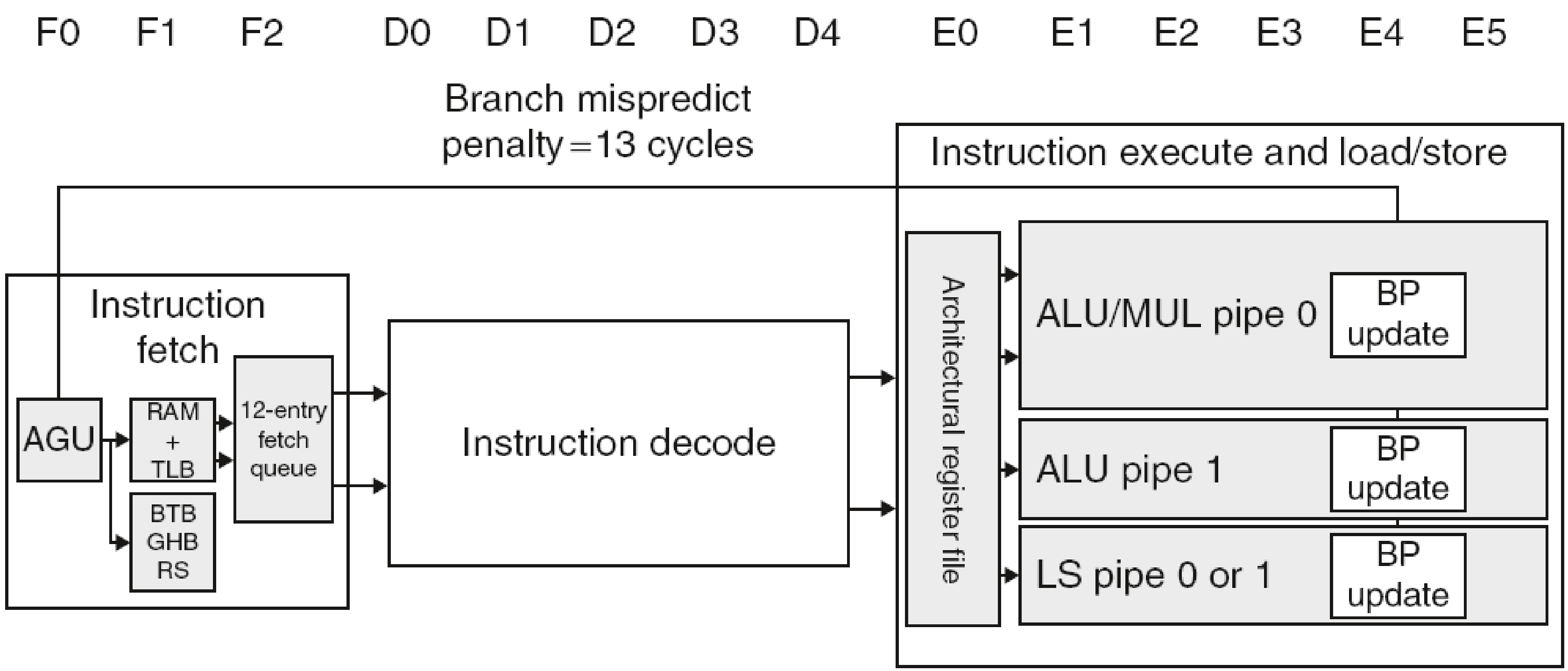
Section 4.11

Real Stuff: The ARM Cortex-A8 and Intel Core i7 Pipelines

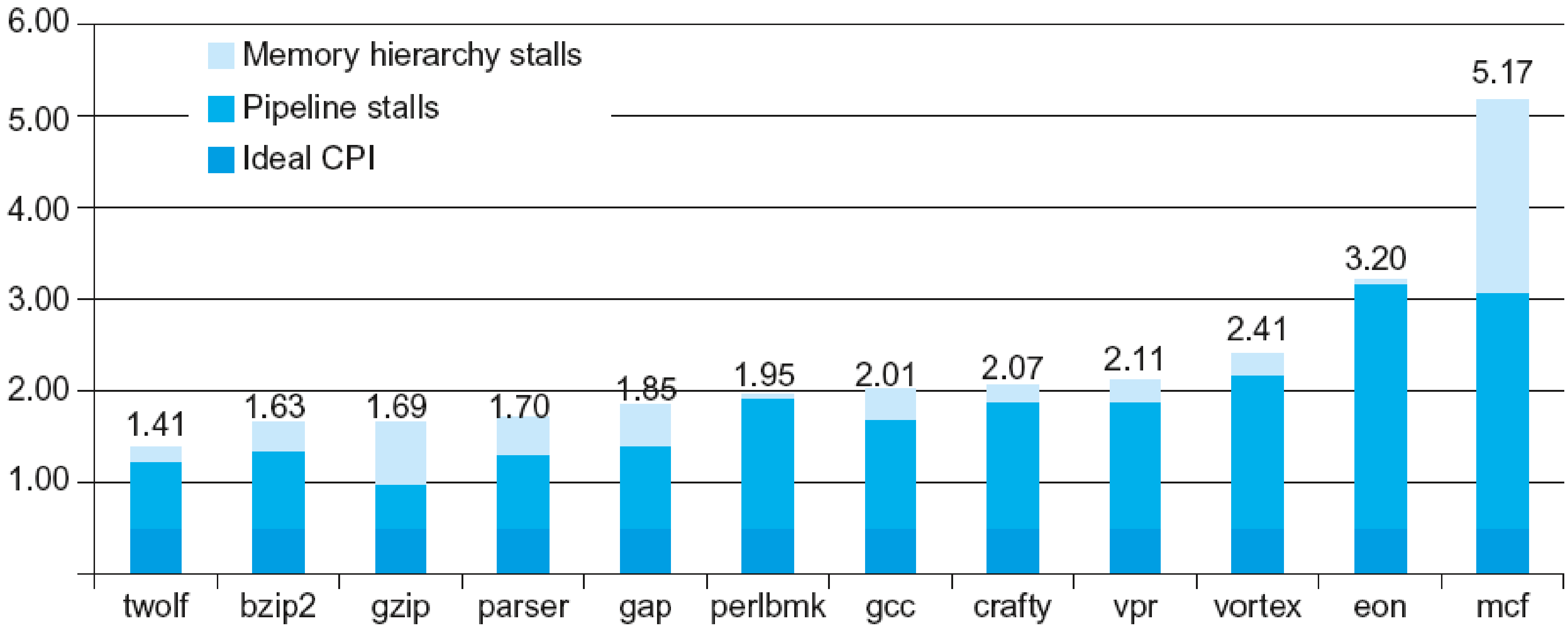
Cortex A8 and Intel i7

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

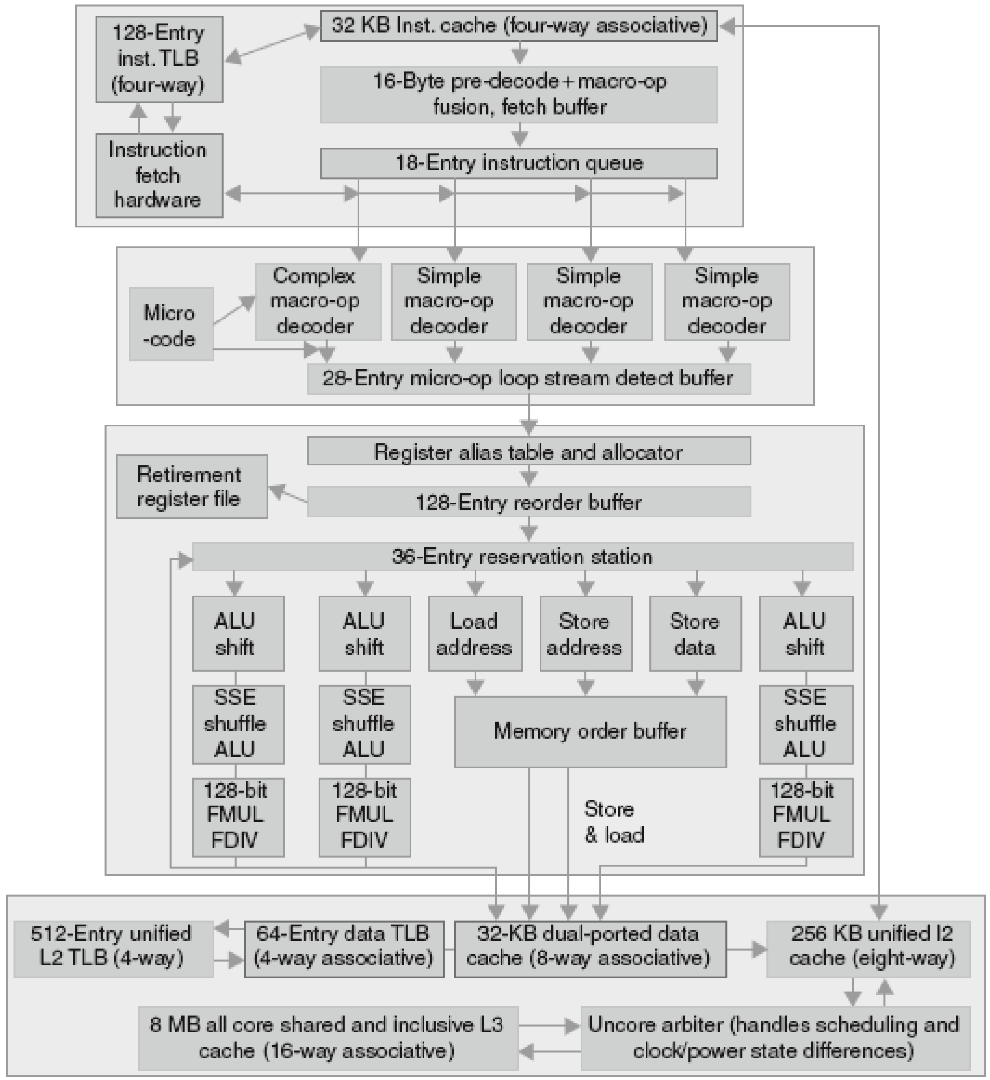
ARM Cortex-A8 Pipeline



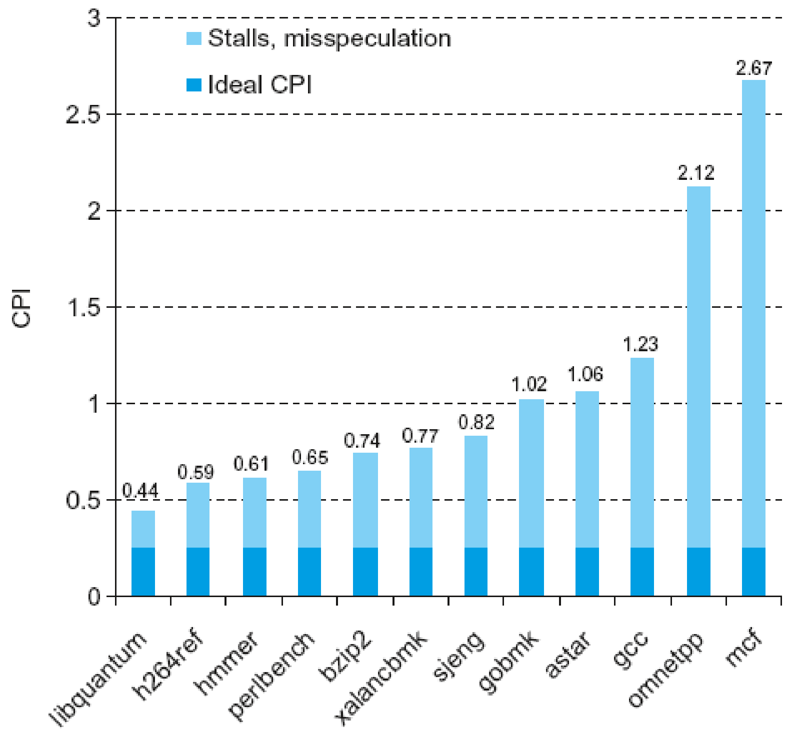
ARM Cortex-A8 Performance

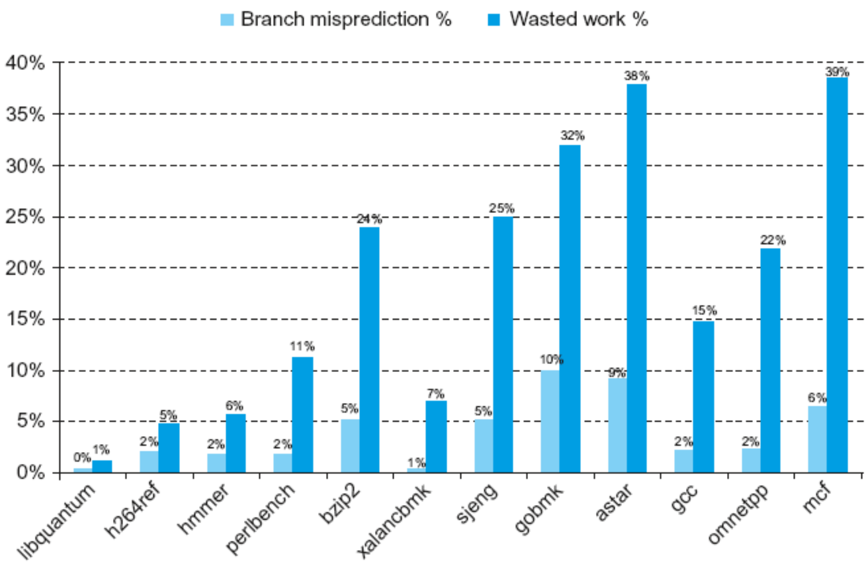


Core i7 Pipeline



Core i7 Performance





Section 4.12

Instruction-Level Parallelism and Matrix Multiply

Matrix Multiply

- Unrolled C code

```
#include <x86intrin.h>
#define UNROLL (4)

void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=UNROLL*4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c[4];
   for ( int x = 0; x < UNROLL; x++ )
    c[x] = _mm256_load_pd(C+i+x*4+j*n);

   for( int k = 0; k < n; k++ )
   {
    __m256d b = _mm256_broadcast_sd(B+k+j*n);
    for (int x = 0; x < UNROLL; x++)
    c[x] = _mm256_add_pd(c[x],
                        _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
   }

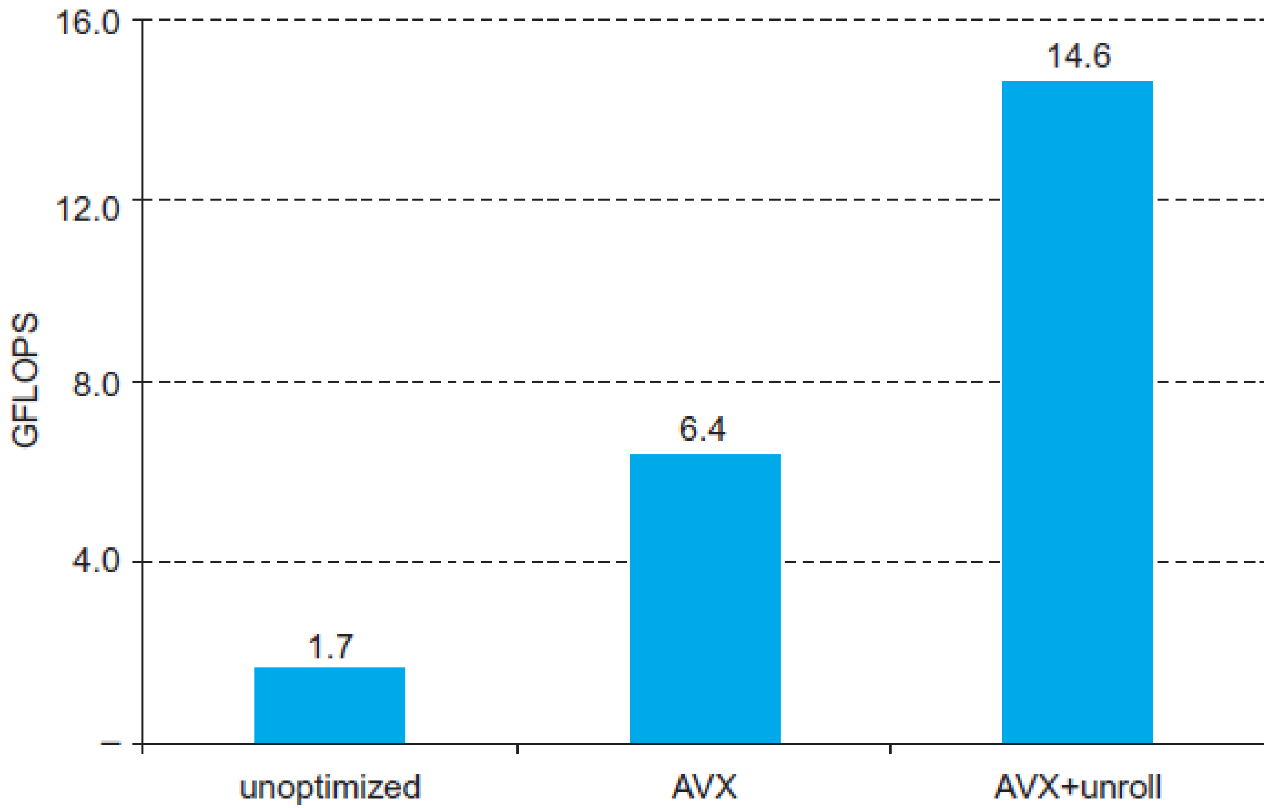
    for ( int x = 0; x < UNROLL; x++ )
     _mm256_store_pd(C+i+x*4+j*n, c[x]);
  }
}
```

Matrix Multiply

- Assembly code:

```
vmovapd (%r11),%ymm4                # Load 4 elements of C into %ymm4
mov %rbx,%rax                       # register %rax = %rbx
xor %ecx,%ecx                       # register %ecx = 0
vmovapd 0x20(%r11),%ymm3            # Load 4 elements of C into %ymm3
vmovapd 0x40(%r11),%ymm2            # Load 4 elements of C into %ymm2
vmovapd 0x60(%r11),%ymm1            # Load 4 elements of C into %ymm1
vbroadcastsd (%rcx,%r9,1),%ymm0     # Make 4 copies of B element
add $0x8,%rcx # register %rcx = %rcx + 8
vmulpd (%rax),%ymm0,%ymm5           # Parallel mul %ymm1,4 A elements
vaddpd %ymm5,%ymm4,%ymm4            # Parallel add %ymm5, %ymm4
vmulpd 0x20(%rax),%ymm0,%ymm5       # Parallel mul %ymm1,4 A elements
vaddpd %ymm5,%ymm3,%ymm3            # Parallel add %ymm5, %ymm3
vmulpd 0x40(%rax),%ymm0,%ymm5       # Parallel mul %ymm1,4 A elements
vmulpd 0x60(%rax),%ymm0,%ymm0       # Parallel mul %ymm1,4 A elements
add %r8,%rax                        # register %rax = %rax + %r8
cmp %r10,%rcx                       # compare %r8 to %rax
vaddpd %ymm5,%ymm2,%ymm2            # Parallel add %ymm5, %ymm2
vaddpd %ymm0,%ymm1,%ymm1            # Parallel add %ymm0, %ymm1
jne 68 <dgemm+0x68>                 # jump if not %r8 != %rax
add $0x1,%esi                       # register % esi = % esi + 1
vmovapd %ymm4,(%r11)                # Store %ymm4 into 4 C elements
vmovapd %ymm3,0x20(%r11)            # Store %ymm3 into 4 C elements
vmovapd %ymm2,0x40(%r11)            # Store %ymm2 into 4 C elements
vmovapd %ymm1,0x60(%r11)            # Store %ymm1 into 4 C elements
```

Performance Impact



Section 4.13

Advanced Topic: An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations

Section 4.14/15

Fallacies and Pitfalls, Concluding Remarks

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
    - e.g., predicated instructions

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

Section 6.4

Hardware multithreading

Multithreading

- How to exploit parallel streams of instructions to improve the performance of a single processor
  - Will ask same question with multiple processors (MIMD) later
- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Memory can be shared through virtual memory
  - Fast switching between threads
    - Process switch: 100s or 1000s of clock cycles
    - Thread switch must be instantaneous

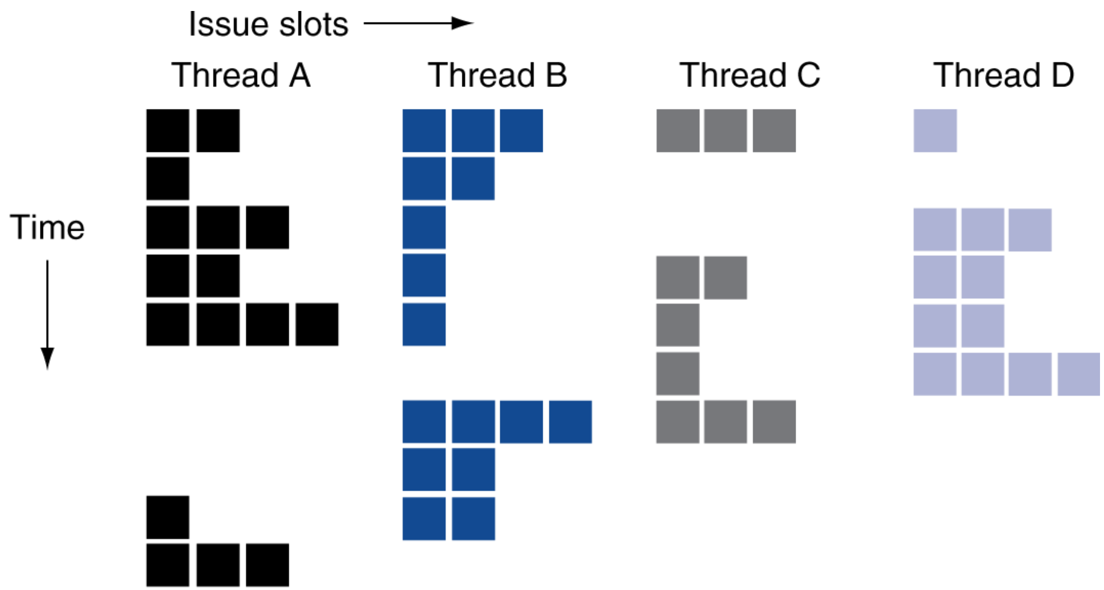
Fine-Grain Multithreading

- Interleave instruction execution (round-robin fashion)
- Switch threads after each cycle
  - Be able to switch threads on every clock cycle
- Can hide both short and long pipeline stalls
- Primary disadvantage:
  - Slow down the execution of individual threads
  - So invented coarse-grained multithreading as alternative

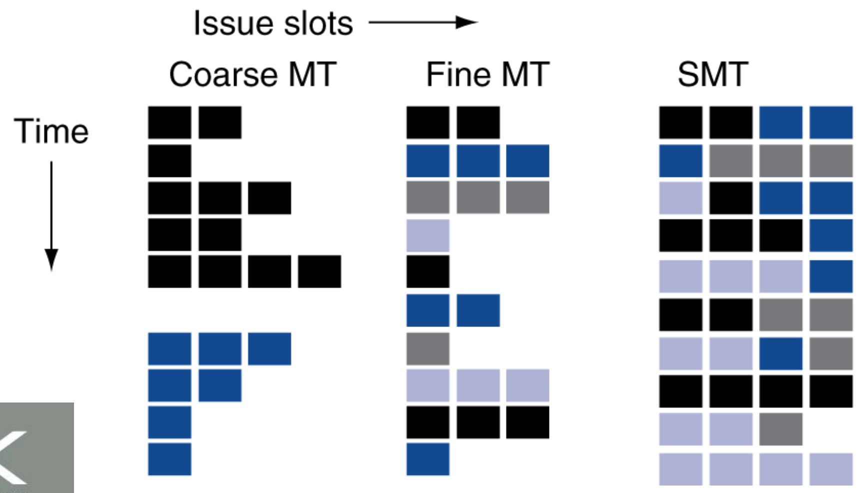
Coarse-Grain Multithreading

- Only switch on long stall (e.g., L2-cache miss)
  - Less likely to slow down individual threads
  - Thread switch not have to be extremely fast
- Execute instructions from a single thread
  - Simplify hardware
  - Doesn’t hide short stalls (eg, data hazards)
  - Pipeline start-up cost
- Useful for reducing the penalty of high-cost stalls
  - Pipeline refill is negligible compared to stall time

Multithreading Example



Superscalar,
No multithreading



Multithreading

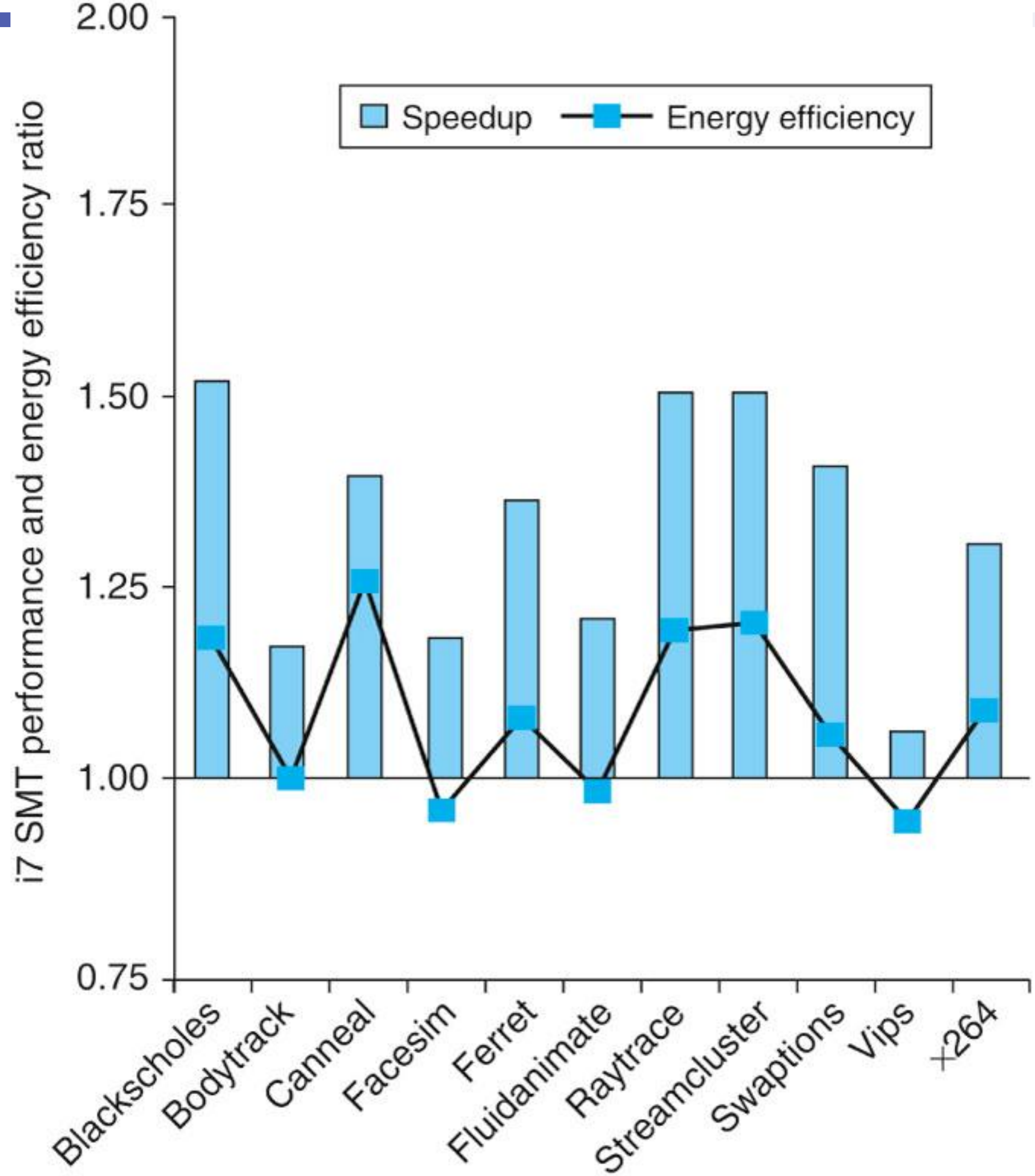
Simultaneous Multithreading

- Motivation: multiple-issue processors have more FUs than single thread's ILP
- Always execute instructions from multiple threads, in multiple-issue dynamically scheduled pipelined processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when FUs are available
  - Within a thread, dependencies handled by scheduling and register renaming
- Exploit TLP (thread-level parallelism) as well as ILP (instruction-level parallelism)

SMT

- Intel desktop products
  - Example: Intel Pentium-4 HT
    - Two threads: duplicated registers, shared function units and caches
  - Single processor of Intel Core i7 960 (next slide)
    - Two-way multithreading
    - Speed-up: 1.31, energy efficiency: 1.07

SMT



PARSEC Benchmarks

Future of Multithreading

- Will it survive? In what form?
- Power considerations $\Rightarrow$ simplified microarchitectures
  - Simpler forms of multithreading
- Tolerating cache-miss latency
  - Thread switch may be most effective
- Multiple simple cores might share resources more effectively