Chapter 4

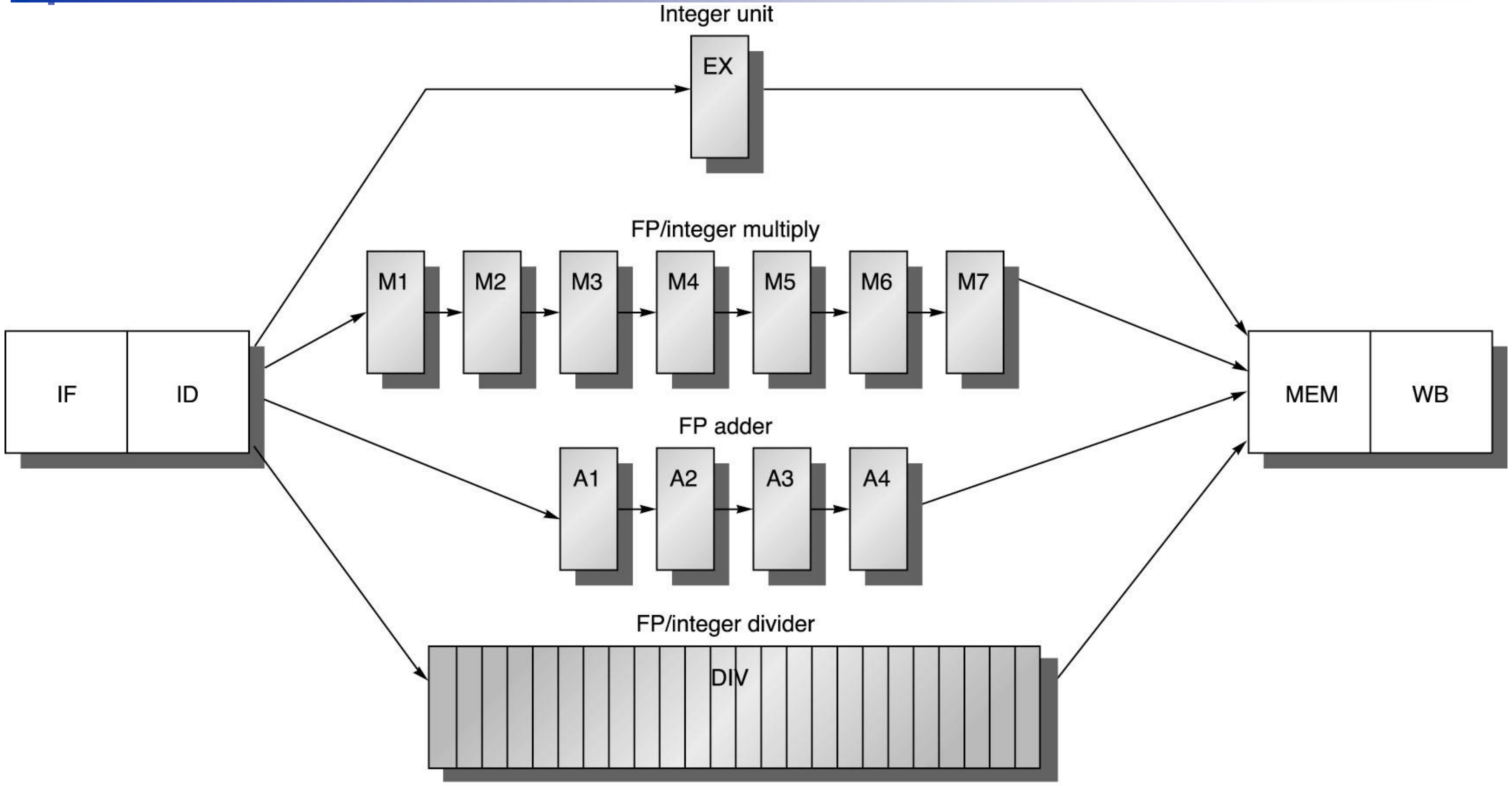
Implementing ISA
(Fetch, Decode, Execute)

Part 3: Quick Look at Advanced Pipelining

Key Concepts

- Floating-point pipeline
- Deep pipeline
- Multiple issue
  - Static (c.f., VLIW)
  - Dynamic (c.f., superscaler)
- Speculation
- Multi-threading
- Multicore processors

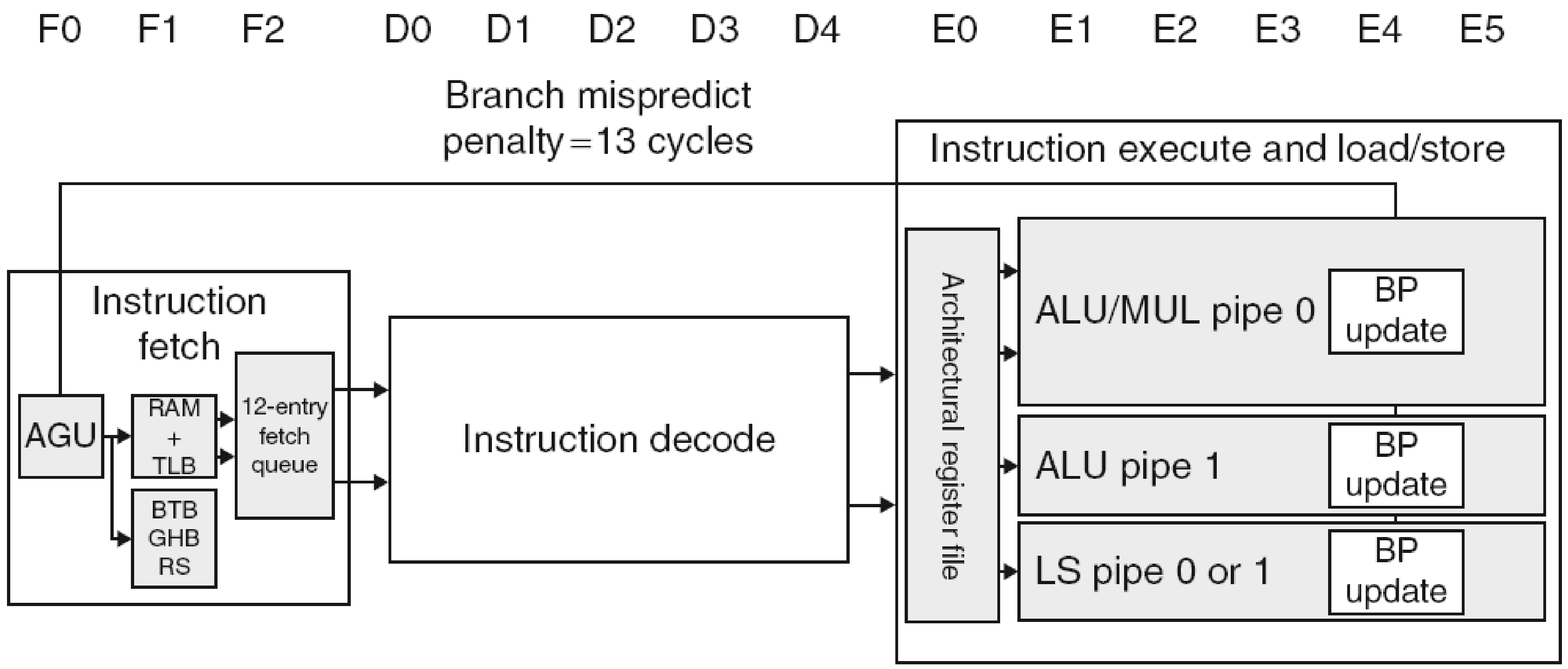
Floating-Point Pipeline



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

ARM Cortex-A8 Pipeline



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
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated
- Key technique to increase ILP (along with scheduling)

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 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

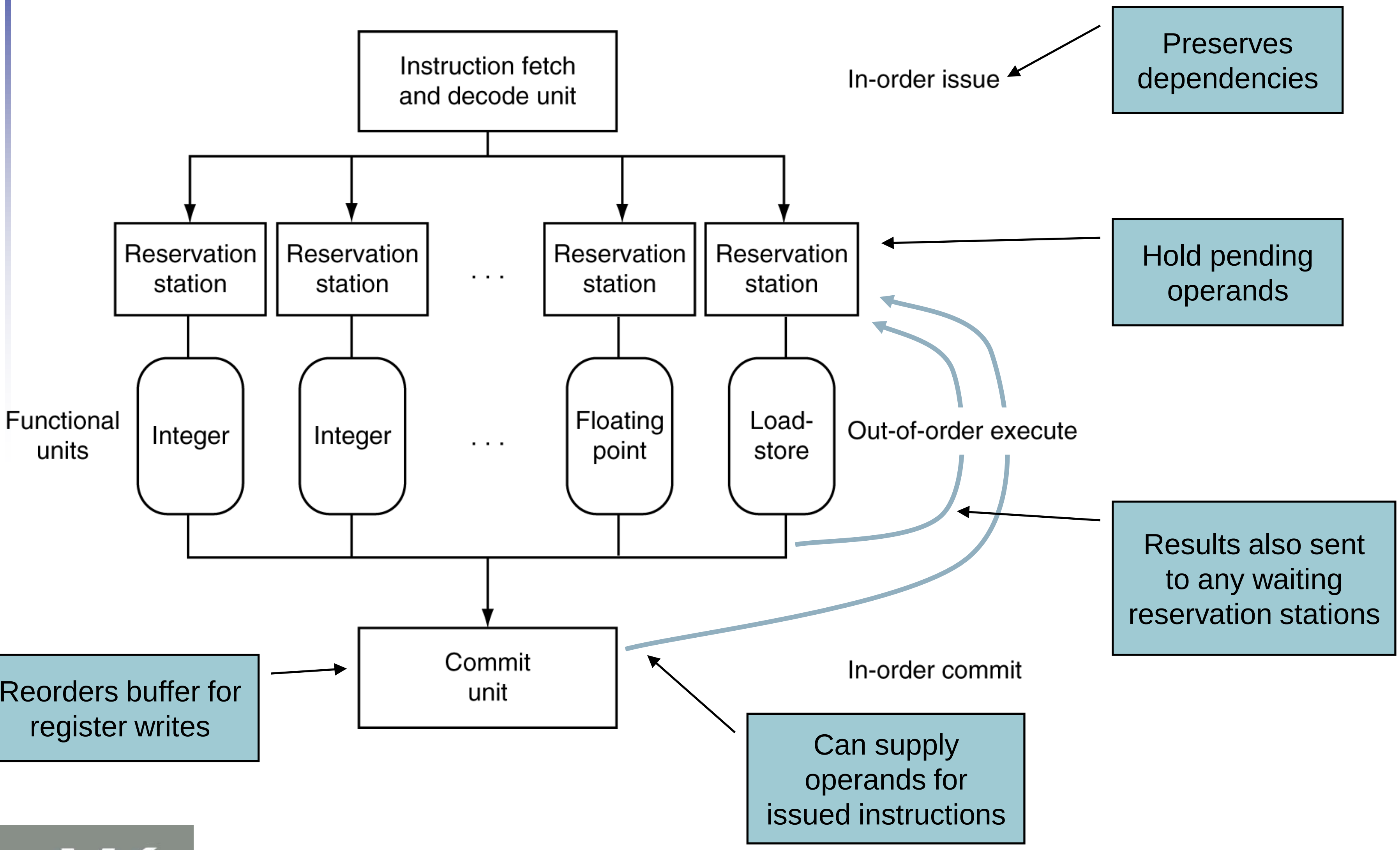
Dynamic Pipeline Scheduling

- 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



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

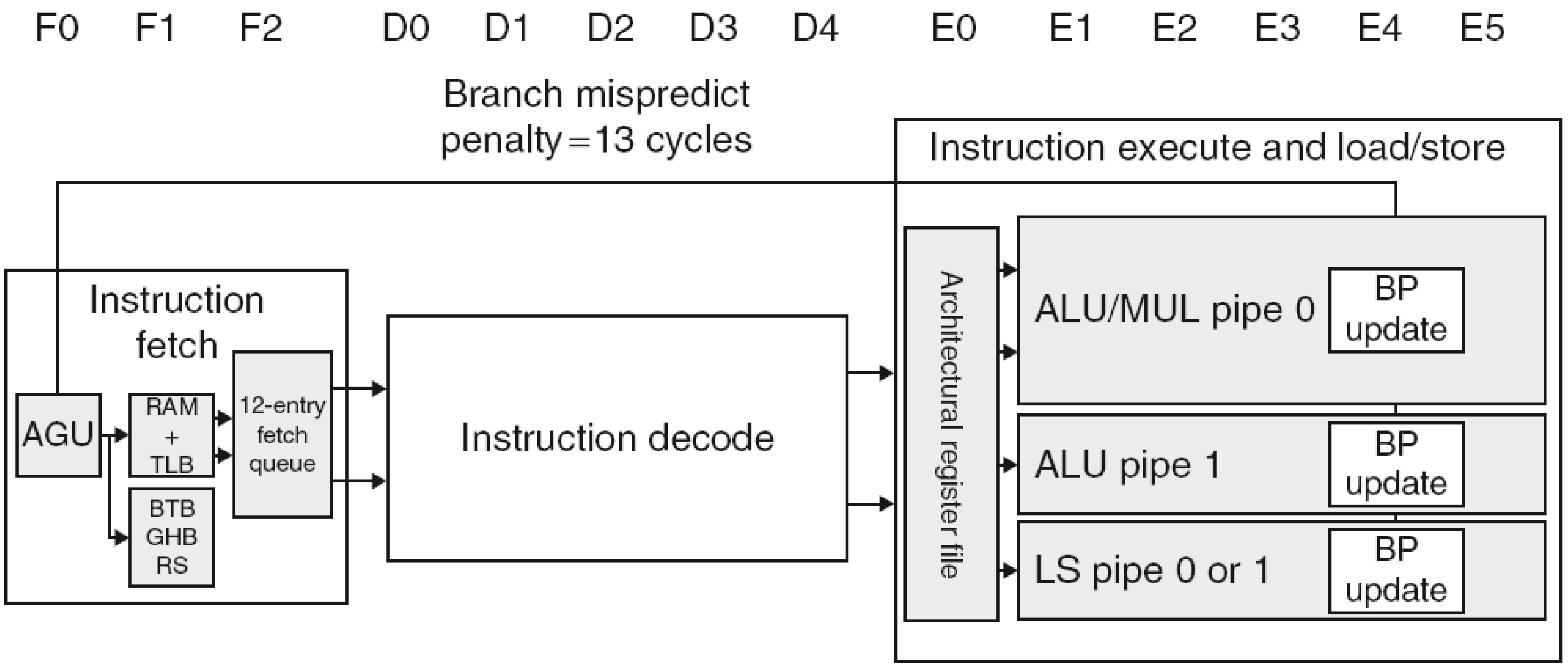
Section 4.11

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

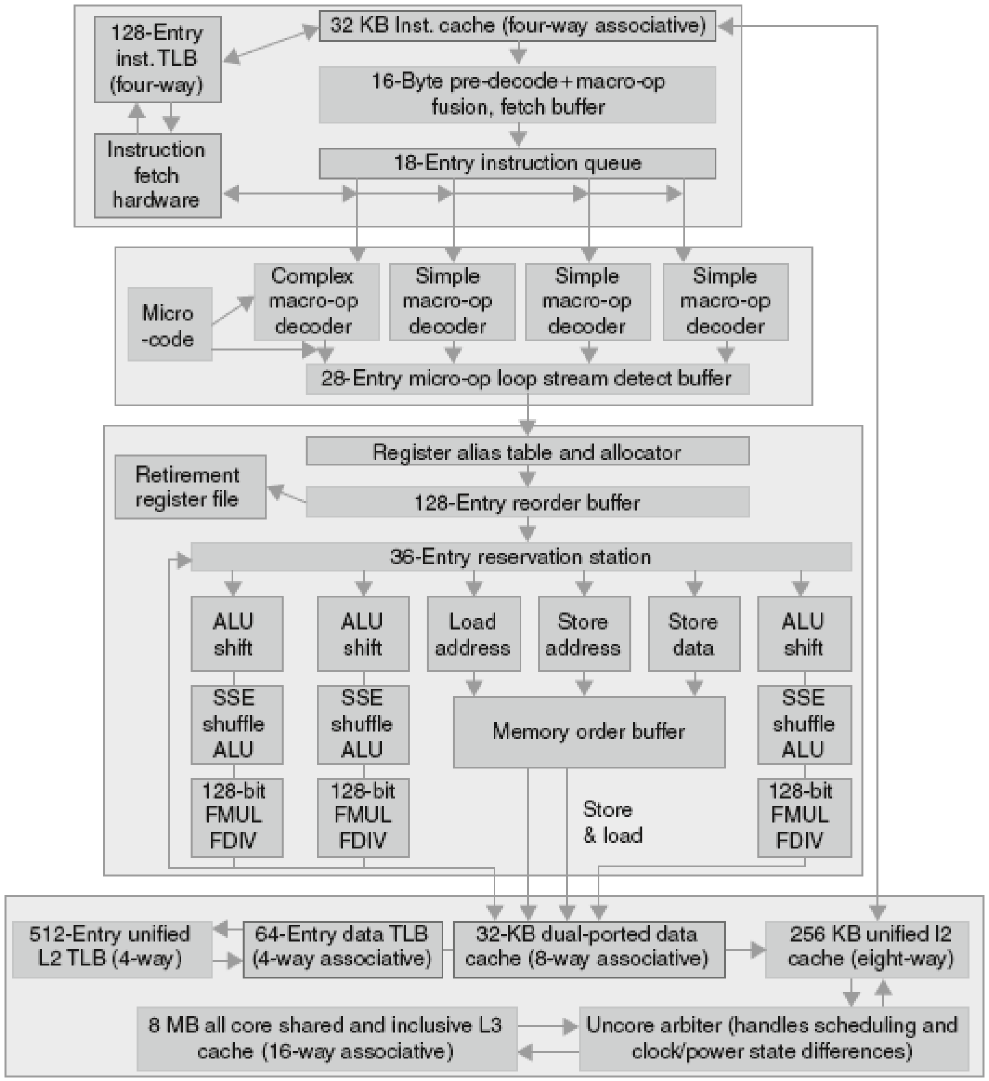
Cortex A8 and Intel i7

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



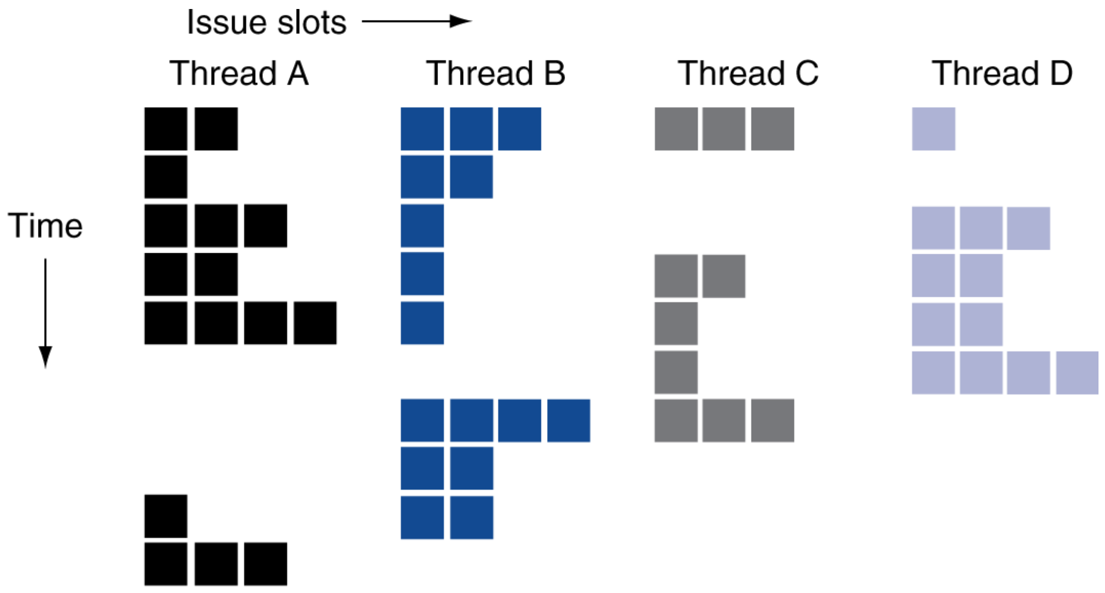
Core i7 Pipeline


128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode + macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
28-Entry micro-op loop stream detect buffer
Register alias table and allocator
Retirement register file
128-Entry reorder buffer
36-Entry reservation station
ALU shift
ALU shift
Load address
Store address
Store data
ALU shift
SSE shuffle ALU
SSE shuffle ALU
Memory order buffer
SSE shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & load
128-bit FMUL FDIV
512-Entry unified L2 TLB (4-way)
64-Entry data TLB (4-way associative)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (eight-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

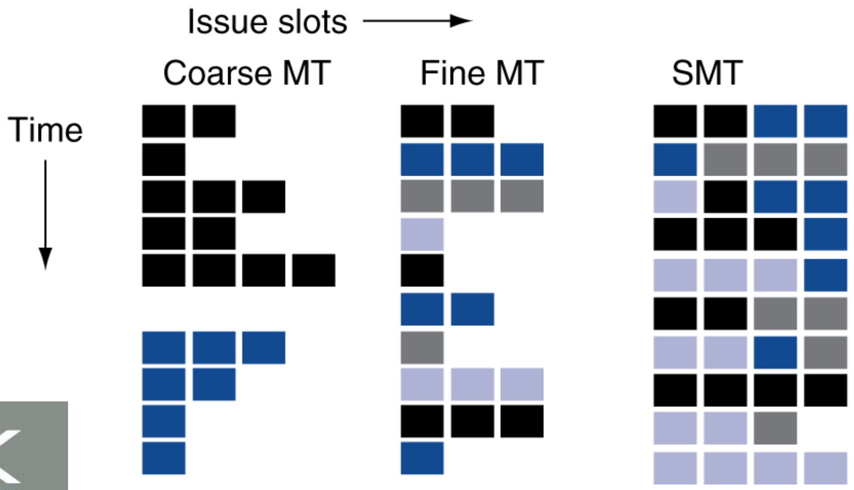
Section 6.4

Hardware multithreading

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)

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

Intel Desktop Products

- Multicore era

	Core i7	Core i5	Core i3	Pentium	Celeron
# cores/threads	4/8, 2/4	2/4	2/4	2/2	2/2
Hyper-Threading	Yes	Yes	Yes		
Turbo Boost	Yes	Yes			
AVX	Yes	Yes	Yes		
CPU overclocking	Yes				

Section 4.12

Instruction-Level Parallelism and Matrix Multiply

Matrix Multiply

- C code for loop unrolling and AVX

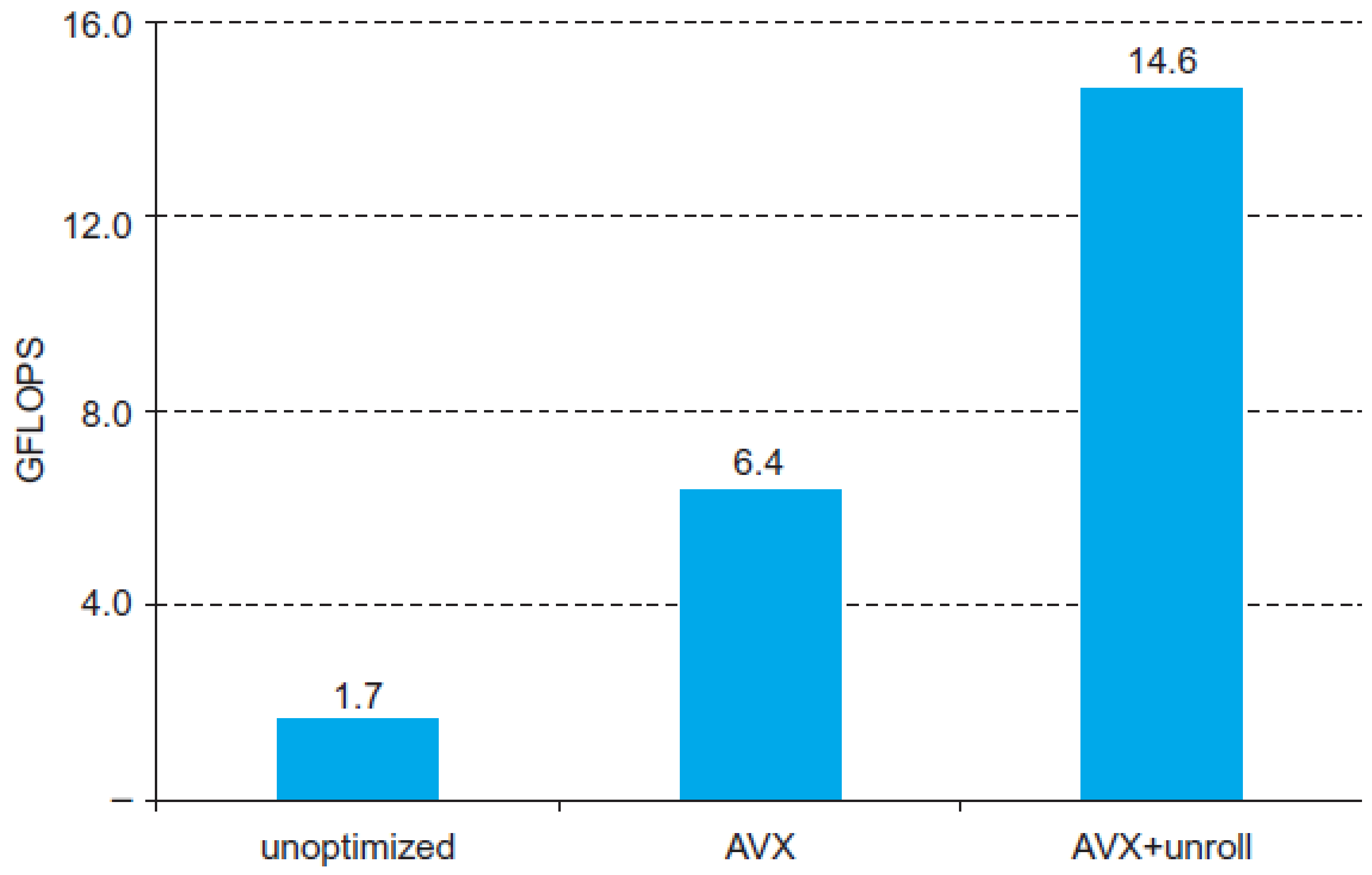
```
#include <x86intrin.h>
#define UNROLL (4)

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 4.9

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