

Unit 11 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

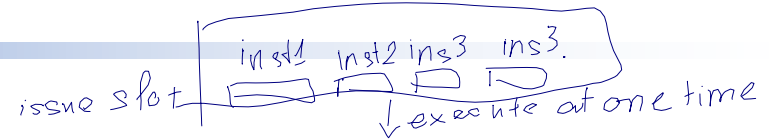
← break down instruction into smaller stages
thus less work per stage, & shorter clock cycle

Execute at the same time

4 instr : 1 cycle.
1 instr : $\frac{1}{4}$ cycle

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
- CPU resolves hazards using advanced techniques at runtime

Speculation

= "guess" in pipe line

- “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

static

dynamic

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

execute in advance
but go back to bubble if incorrect

bubble/no op

this is the hardware

execute in advance
but store in buffer & flush buffer if incorrect

only on branch cond.

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?

- e.g., speculative load before null-pointer check

unconfirmed path.

- Static speculation

- Can add ISA support for deferring exceptions

software use ISA support to defer exception on unconfirm path

- Dynamic speculation

- Can buffer exceptions until instruction completion (which may not occur)

hardware

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)
only for software / static multiple issue

Scheduling Static Multiple Issue

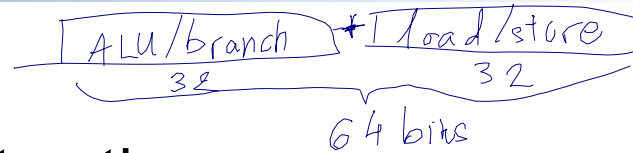
- Compiler must remove some/all hazards
 - Reorder instructions into issue packets
 - No dependencies with a packet
 - Possibly some dependencies between packets
 - Varies between ISAs; compiler must know!
 - Pad with nop if necessary

instruction scheduling for data hazard is handled by compiler

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



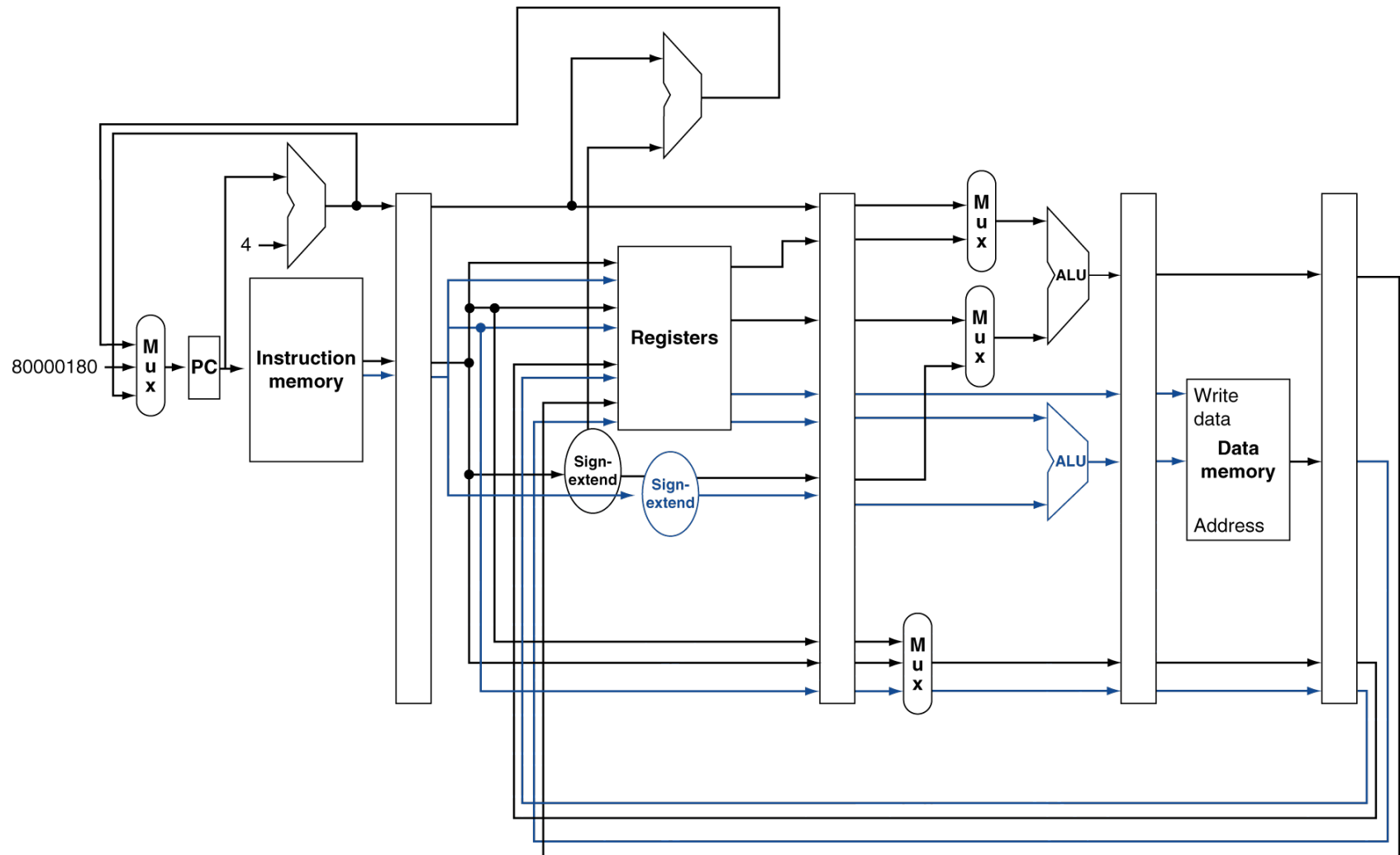
combine multiple issue 4. pipeline.

Address	Instruction type	Pipeline Stages						
		IF	ID	EX	MEM	WB		
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

but need to watch out for data dependency (hazard.)

multiple issue. load at the same time

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)` ← data hazard. input == output.
 - Split into two packets, effectively a stall
- More aggressive scheduling required

↑
same as reorder (in pipeline)
(speculation vs prediction (no related))

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 $\$s1 \neq 0$

the important instr to jump or not →

	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

this depends on (2) thus scheduled to (4)

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

compiler decides how the scheduling is organized.

Loop Unrolling

traditional loop.
- exec every statement
in the loop

Unrolling: (static way) (VLIW also static way)
the OS has an algorithm:
- how many times the loop runs. (for example 3 times)
- expand the unimportant instructions 3 times (previous sample)
& run the important instr 1 time in last iteration.
⇒ save instruction counts
(⇒ less instruction count)
(⇒ better CPI) 2 benefits

Qn: which one is static unroll?
dynamic way?

- Replicate loop body to expose more parallelism

check the condition in final stage only

- Reduces loop-control overhead

- Use different registers per replication

- Called “register renaming”

to avoid dependency

- 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

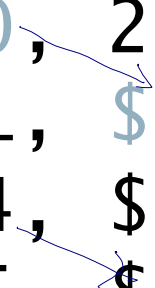
- “Superscalar” processors *using HW to solve multiple issues*
- CPU decides whether to issue 0, 1, 2, ... each cycle
- Avoids the need for compiler scheduling
 - Though it may still help
 - Code semantics ensured by the CPU

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
slli    $t5, $s4, 20
```

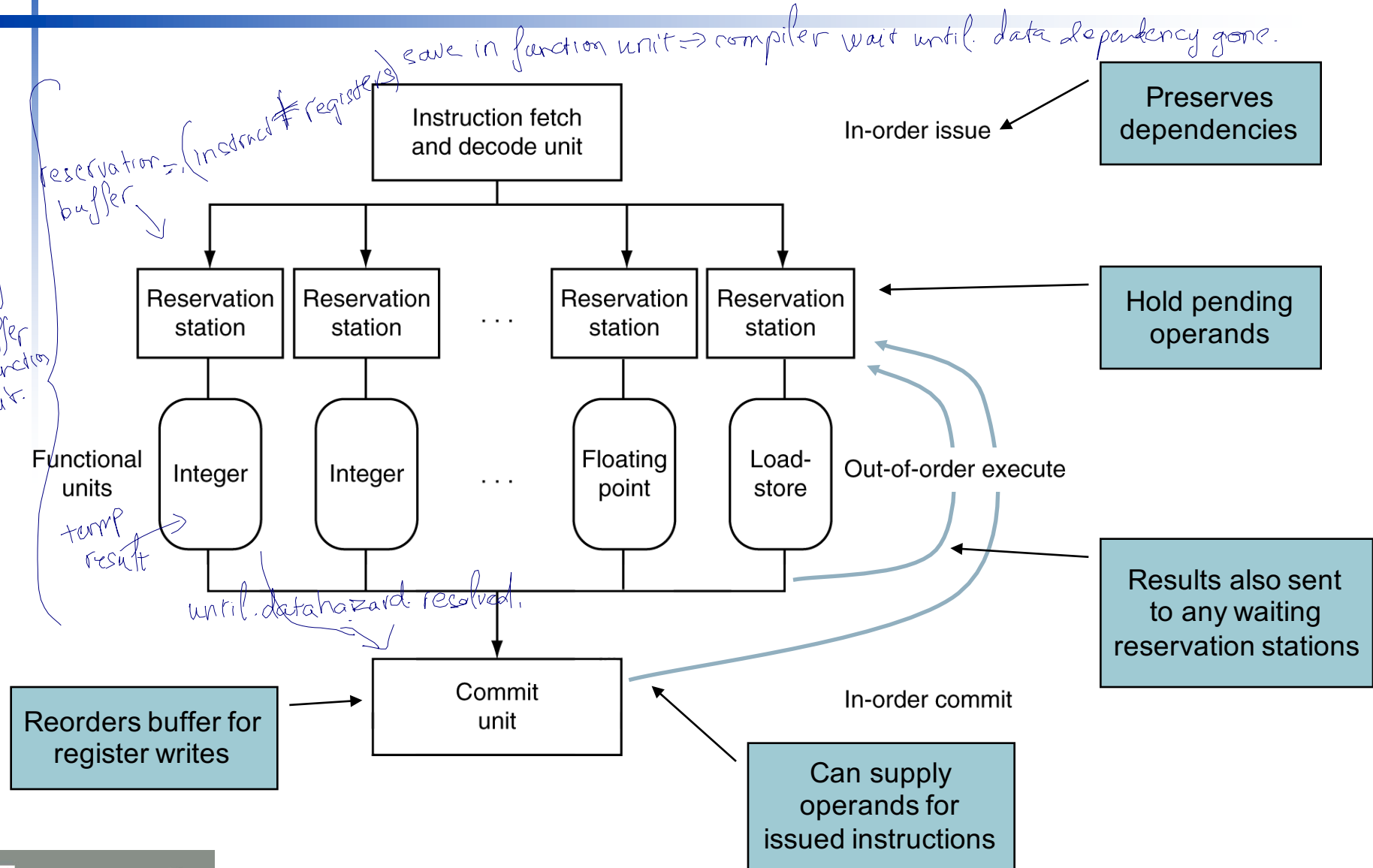


leading output register == following input register.

- Can start sub while addu is waiting for lw

HW to fix data hazard.

Dynamically Scheduled CPU



Register Renaming

← put all registers that have data hazard prob
& create ⁱⁿ new temp register name.
before
to resolve data dependency prob

- 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

no dependency

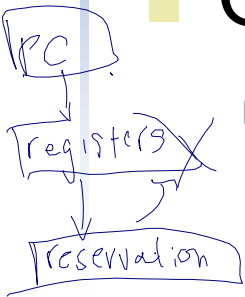
- Copied to reservation station
- No longer required in the register; can be overwritten

← register freed up for reuse

- If operand is not yet available

← depends on some results, wait for reorder buffer to provision

- It will be provided to the reservation station by a function unit



Speculation

multiple issue. to handle branch. cmd. → we speculate.
why speculate → not waste time

- Predict branch and continue issuing
 - Don't commit until branch outcome determined
- Load speculation
 - Avoid load data hazard
 - Load before completing outstanding store
 - Don't commit load until speculation cleared

sometimes we can't use software way, has to use HW way.

Why Do Dynamic Scheduling?

Q/N

- Why not just let the compiler schedule code?
- Not all stalls are predicable
- 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
- Instructions have dependencies will limit ILP
- Some dependencies are hard to eliminate
- Speculation can help if done well

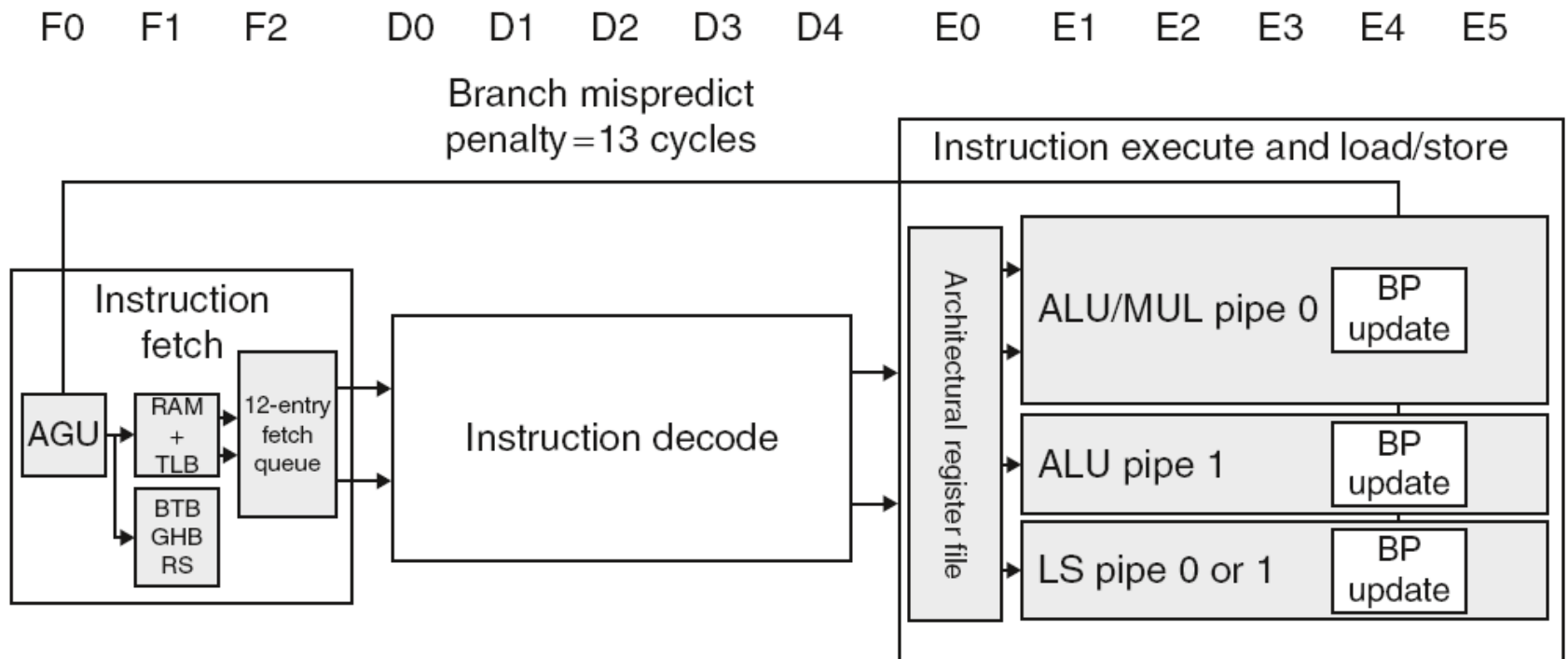
Cortex A8 and Intel i7

→ g 2k from this page.

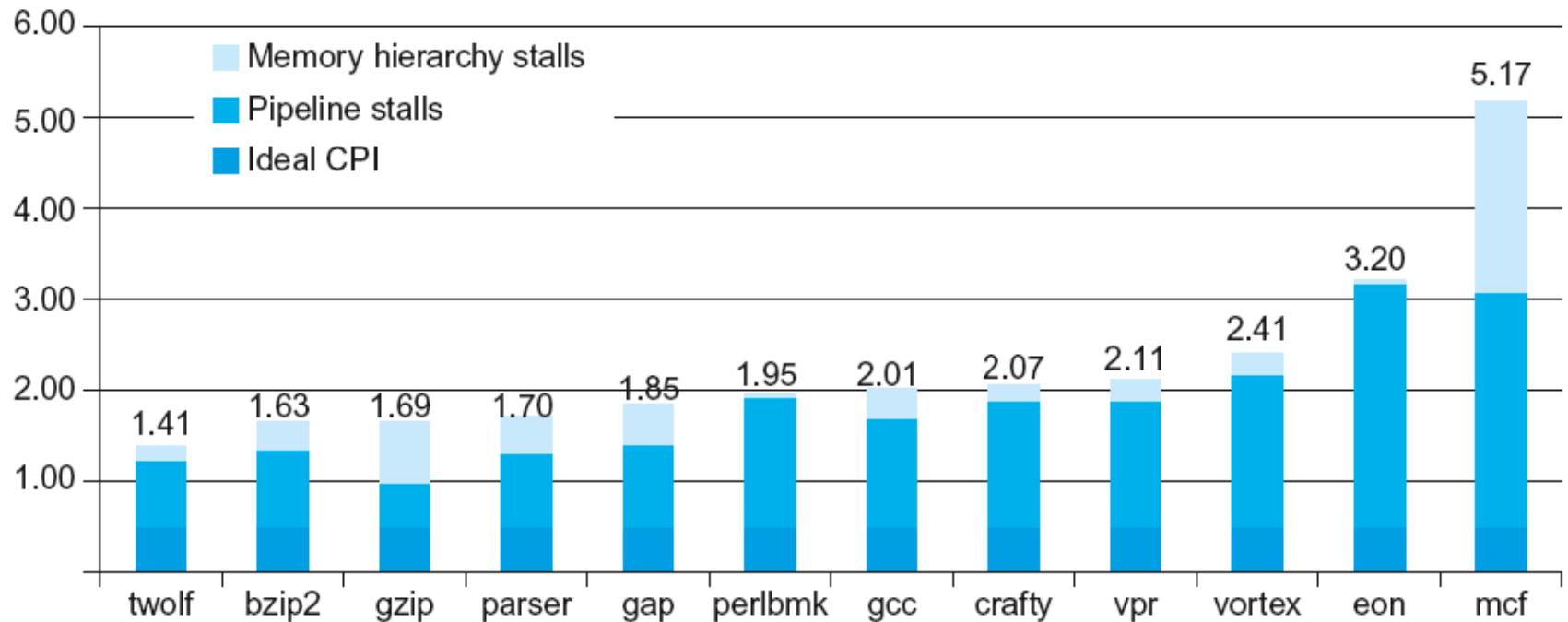
Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	14	14
Pipeline schedule	Static in-order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

ARM Cortex-A8 Pipeline

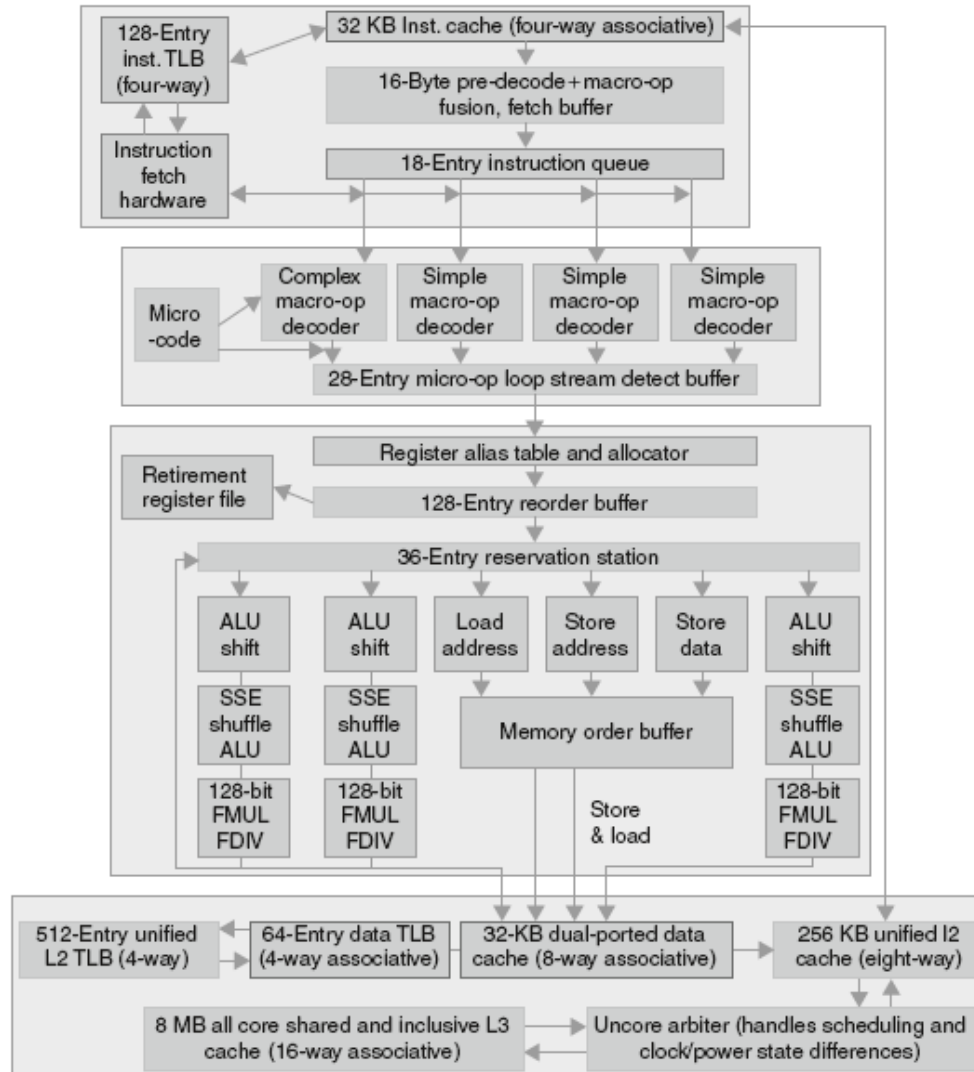
g2k



ARM Cortex-A8 Performance ^{92k}

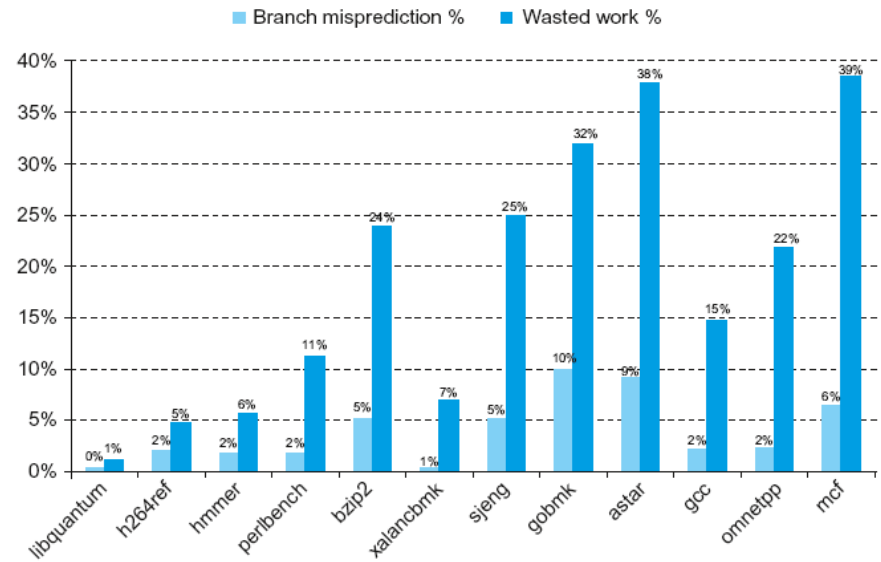
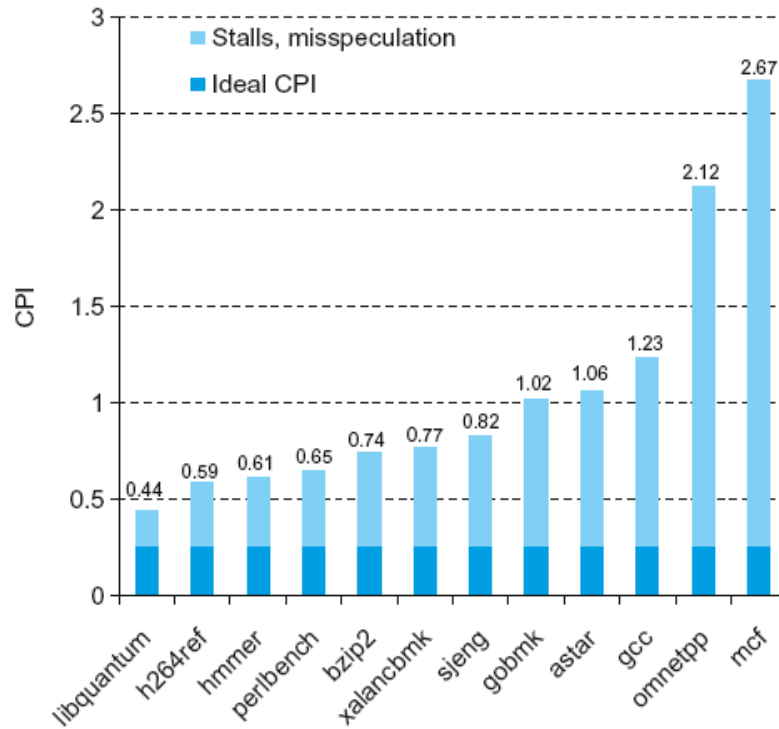


Core i7 Pipeline



Core i7 Performance

g2k



Matrix Multiply

g2k

■ Unrolled C code

```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3
4 void dgemm (int n, double* A, double* B, double* C)
5 {
6     for ( int i = 0; i < n; i+=UNROLL*4 )
7         for ( int j = 0; j < n; j++ ) {
8             __m256d c[4];
9             for ( int x = 0; x < UNROLL; x++ )
10                 c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12             for( int k = 0; k < n; k++ )
13             {
14                 __m256d b = _mm256_broadcast_sd(B+k*j*n);
15                 for (int x = 0; x < UNROLL; x++)
16                     c[x] = _mm256_add_pd(c[x],
17                                           _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18             }
19
20             for ( int x = 0; x < UNROLL; x++ )
21                 _mm256_store_pd(C+i+x*4+j*n, c[x]);
22         }
23 }
```

Matrix Multiply

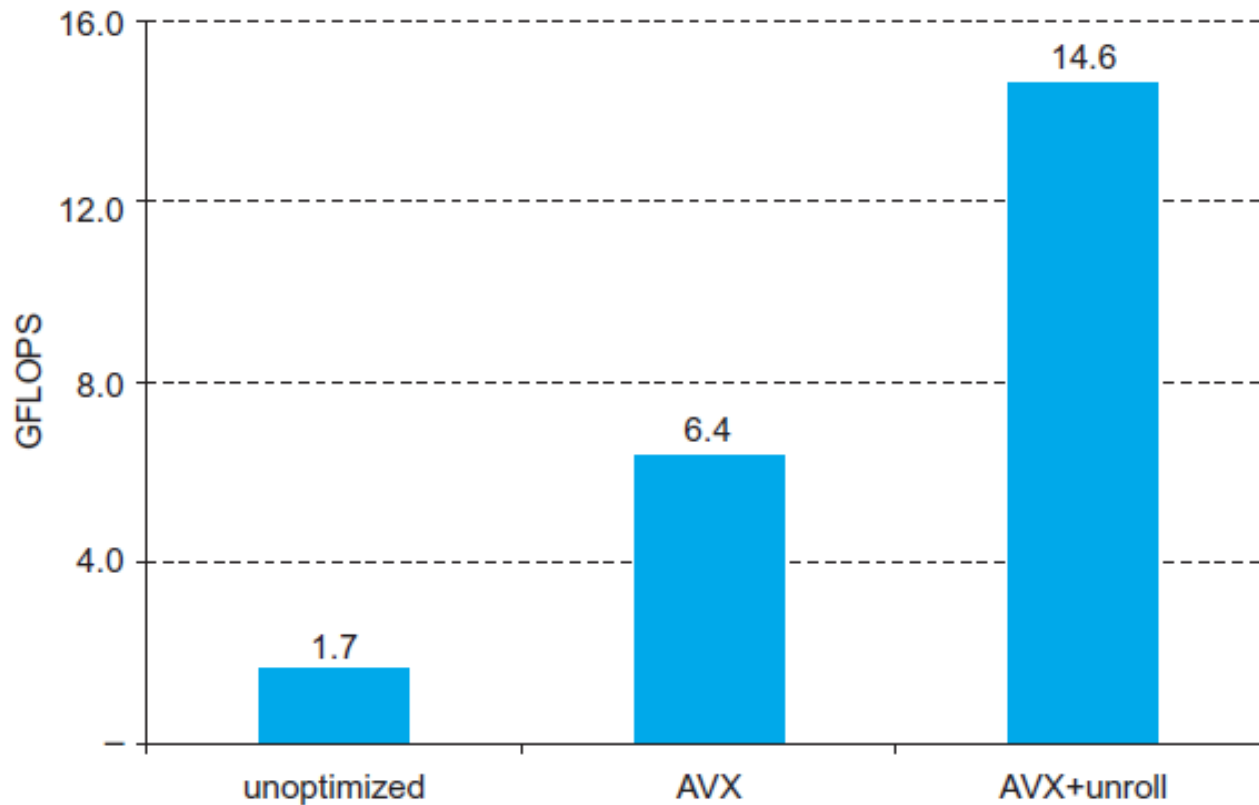
g2k

■ Assembly code:

```
1 vmovapd (%r11),%ymm4          # Load 4 elements of C into %ymm4
2 mov %rbx,%rax                 # register %rax = %rbx
3 xor %ecx,%ecx                 # register %ecx = 0
4 vmovapd 0x20(%r11),%ymm3      # Load 4 elements of C into %ymm3
5 vmovapd 0x40(%r11),%ymm2      # Load 4 elements of C into %ymm2
6 vmovapd 0x60(%r11),%ymm1      # Load 4 elements of C into %ymm1
7 vbroadcastsd (%rcx,%r9,1),%ymm0 # Make 4 copies of B element
8 add $0x8,%rcx # register %rcx = %rcx + 8
9 vmulpd (%rax),%ymm0,%ymm5      # Parallel mul %ymm1,4 A elements
10 vaddpd %ymm5,%ymm4,%ymm4      # Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5 # Parallel mul %ymm1,4 A elements
12 vaddpd %ymm5,%ymm3,%ymm3      # Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5 # Parallel mul %ymm1,4 A elements
14 vmulpd 0x60(%rax),%ymm0,%ymm0 # Parallel mul %ymm1,4 A elements
15 add %r8,%rax                 # register %rax = %rax + %r8
16 cmp %r10,%rcx               # compare %r8 to %rax
17 vaddpd %ymm5,%ymm2,%ymm2      # Parallel add %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1      # Parallel add %ymm0, %ymm1
19 jne 68 <dgemm+0x68>          # jump if not %r8 != %rax
20 add $0x1,%esi               # register %esi = %esi + 1
21 vmovapd %ymm4, (%r11)        # Store %ymm4 into 4 C elements
22 vmovapd %ymm3, 0x20(%r11)    # Store %ymm3 into 4 C elements
23 vmovapd %ymm2, 0x40(%r11)    # Store %ymm2 into 4 C elements
24 vmovapd %ymm1, 0x60(%r11)    # Store %ymm1 into 4 C elements
```

Performance Impact

g2k



Fallacies (Misconception)

- Pipelining is ^{NOT} easy (Wrong)
 - The basic idea is easy
 - The devil is in the details *← implementation is tough*
 - e.g., detecting data hazards
- Pipelining is ~~independent~~ of technology (Wrong)
 - More transistors make more advanced techniques feasible
 - Pipeline-related ISA design needs to take account of technology trends

Pitfalls (Traps, Confusions)

- Poor ISA design can make pipelining harder (Yes)
 - 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)
 - Instruction dependencies limit achievable parallelism
 - Instruction complexity leads to the power wall

test

take advantage
of HW.