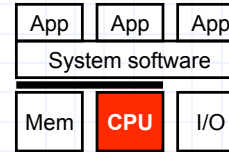


CIS 371

Computer Organization and Design

Unit 8: Static and Dynamic Scheduling

This Unit: Code Scheduling



- Pipelining and superscalar review
- Code scheduling
 - To reduce pipeline stalls
 - To increase ILP (insn level parallelism)
- Two approaches
 - Static scheduling by the compiler
 - Dynamic scheduling by the hardware

Readings

- P+H
 - Chapter 6.9 (again)

Pipelining Review

- Increases clock frequency by staging instruction execution
- “Scalar” pipelines have a best-case CPI of 1
- Challenges:
 - Data and control dependencies further worsen CPI
 - Data: With full bypassing, load-to-use stalls
 - Control: use branch prediction to mitigate penalty
- Big win, done by all processors today
- How many stages (depth)?
 - Five stages is pretty good minimum
 - Intel Pentium II/III: 12 stages
 - Intel Pentium 4: 22+ stages
 - Intel Core 2: 14 stages

Pipeline Diagram

	1	2	3	4	5	6	7	8	9
add \$3,\$2,\$1	F	D	X	M	W				
lw \$4,4(\$3)		F	D	X	M	W			
addi \$6,\$4,1			F	D	d*	X	M	W	
sub \$8,\$3,\$1				F	d*	D	X	M	W

- Use compiler scheduling to reduce load-use stall frequency
 - Like software interlocks, but for performance not correctness

	1	2	3	4	5	6	7	8	9
add \$3,\$2,\$1	F	D	X	M	W				
lw \$4,4(\$3)		F	D	X	M	W			
sub \$8,\$3,\$1			F	D	X	M	W		
addi \$6,\$4,1				F	D	X	M	W	

Superscalar Pipeline Review

- Execute two or more instruction per cycle
- Challenges:
 - wide fetch (branch prediction harder, misprediction more costly)
 - wide decode (stall logic)
 - wide execute (more ALUs)
 - wide bypassing (more possibly bypassing paths)
 - Finding enough independent instructions (and fill delay slots)
- How many instructions per cycle max (width)?
 - Really simple, low-power cores are still scalar (single issue)
 - Even low-power cores a dual-issue (Intel Atom, aka Silverthorne)
 - Most desktop/laptop chips three-issue or four-issue
 - A few 5 or 6-issue chips have been built (IBM Power4, Itanium II)

Superscalar Pipeline Diagrams - Ideal

scalar

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(r1) → r2	F	D	X	M	W							
lw 4(r1) → r3		F	D	X	M	W						
lw 8(r1) → r4			F	D	X	M	W					
add r14,r15 → r6				F	D	X	M	W				
add r12,r13 → r7					F	D	X	M	W			
add r17,r16 → r8						F	D	X	M	W		
lw 0(r18) → r9							F	D	X	M	W	

2-way superscalar

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(r1) → r2	F	D	X	M	W							
lw 4(r1) → r3		F	D	X	M	W						
lw 8(r1) → r4			F	D	X	M	W					
add r14,r15 → r6				F	D	X	M	W				
add r12,r13 → r7					F	D	X	M	W			
add r17,r16 → r8						F	D	X	M	W		
lw 0(r18) → r9							F	D	X	M	W	

Superscalar Pipeline Diagrams - Realistic

scalar

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(r1) → r2	F	D	X	M	W							
lw 4(r1) → r3		F	D	X	M	W						
lw 8(r1) → r4			F	D	X	M	W					
add r4,r5 → r6				F	d*	D	X	M	W			
add r2,r3 → r7					F	D	X	M	W			
add r7,r6 → r8						F	D	X	M	W		
lw 0(r8) → r9							F	D	X	M	W	

2-way superscalar

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(r1) → r2	F	D	X	M	W							
lw 4(r1) → r3		F	D	X	M	W						
lw 8(r1) → r4			F	D	X	M	W					
add r4,r5 → r6				F	d*	d*	D	X	M	W		
add r2,r3 → r7					F	d*	D	X	M	W		
add r7,r6 → r8						F	D	X	M	W		
lw 0(r8) → r9							F	d*	D	X	M	W

Code Scheduling

- Scheduling: act of finding independent instructions
 - "Static" done at compile time by the compiler (software)
 - "Dynamic" done at runtime by the processor (hardware)
- Why schedule code?
 - Scalar pipelines: fill in load-to-use delay slots to improve CPI
 - Superscalar: place independent instructions together
 - As above, load-to-use delay slots
 - Allow multiple-issue decode logic to let them execute at the same time

Compiler Scheduling

- Compiler can schedule (move) instructions to reduce stalls
 - **Basic pipeline scheduling**: eliminate back-to-back load-use pairs
 - Example code sequence: $a = b + c$; $d = f - e$;
 - sp stack pointer, $sp+0$ is "a", $sp+4$ is "b", etc...

Before	After
ld r2,4(sp)	ld r2,4(sp)
ld r3,8(sp)	ld r3,8(sp)
add r3,r2,r1 //stall	ld r5,16(sp)
st r1,0(sp)	add r3,r2,r1 //no stall
ld r5,16(sp)	ld r6,20(sp)
ld r6,20(sp)	st r1,0(sp)
sub r5,r6,r4 //stall	sub r5,r6,r4 //no stall
st r4,12(sp)	st r4,12(sp)

Compiler Scheduling Requires

- **Large scheduling scope**
 - Independent instruction to put between load-use pairs
 - + Original example: large scope, two independent computations
 - This example: small scope, one computation

Before	After
ld r2,4(sp)	ld r2,4(sp)
ld r3,8(sp)	ld r3,8(sp)
add r3,r2,r1 //stall	add r3,r2,r1 //stall
st r1,0(sp)	st r1,0(sp)

- One way to create larger scheduling scopes?



Compiler Scheduling Requires

- **Enough registers**
 - To hold additional "live" values
 - Example code contains 7 different values (including sp)
 - Before: max 3 values live at any time → 3 registers enough
 - After: max 4 values live → 3 registers not enough

Original	Wrong!
ld r2,4(sp)	ld r2,4(sp)
ld r1,8(sp)	ld r1,8(sp)
add r1,r2,r1 //stall	ld r2,16(sp)
st r1,0(sp)	add r1,r2,r1 // wrong r1
ld r2,16(sp)	ld r1,20(sp)
ld r1,20(sp)	st r1,0(sp) // wrong r1
sub r2,r1,r1 //stall	sub r2,r1,r1
st r1,12(sp)	st r1,12(sp)

Compiler Scheduling Requires

- **Alias analysis**
 - Ability to tell whether load/store reference same memory locations
 - Effectively, whether load/store can be rearranged
 - Example code: easy, all loads/stores use same base register (*sp*)
 - New example: can compiler tell that *r8 != sp*?
 - Must be **conservative**

Before

```
ld r2,4(sp)
ld r3,8(sp)
add r3,r2,r1 //stall
st r1,0(sp)
ld r5,0(r8)
ld r6,4(r8)
sub r5,r6,r4 //stall
st r4,8(r8)
```

Wrong(?)

```
ld r2,4(sp)
ld r3,8(sp)
ld r5,0(r8) //does r8==sp?
add r3,r2,r1
ld r6,4(r8) //does r8+4==sp?
st r1,0(sp)
sub r5,r6,r4
st r4,8(r8)
```

Code Example: SAXPY

- **SAXPY** (Single-precision A X Plus Y)
 - Linear algebra routine (used in solving systems of equations)
 - Part of early "Livermore Loops" benchmark suite

```
for (i=0;i<N;i++)
    Z[i]=A*X[i]+Y[i];
```

```
0: ldf X(r1)→f1      // loop
1: mulf f0,f1→f2    // A in f0
2: ldf Y(r1)→f3    // X,Y,Z are constant addresses
3: addf f2,f3→f4
4: stf f4→Z(r1)
5: addi r1,4→r1    // i in r1
6: blt r1,r2,0     // N*4 in r2
```

New Metric: Utilization

- **Utilization**: actual performance / peak performance
 - Important metric for performance/cost
 - No point to paying for hardware you will rarely use
- Adding hardware usually improves performance & reduces utilization
 - Additional hardware can only be exploited some of the time
 - Diminishing marginal returns
- Compiler can help make better use of existing hardware
 - Important for superscalar

SAXPY Performance and Utilization

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
ldf X(r1)→f1	F	D	X	M	W															
mulf f0,f1→f2		F	D	d*	E*	E*	E*	E*	E*	W										
ldf Y(r1)→f3			F	p*	D	X	M	W												
addf f2,f3→f4				F	D	d*	d*	d*	E+	E+	W									
stf f4→Z(r1)					F	p*	p*	p*	D	X	M	W								
addi r1,4→r1									F	D	X	M	W							
blt r1,r2,0										F	D	X	M	W						
ldf X(r1)→f1											F	D	X	M	W					

- **Scalar pipeline**
 - Full bypassing, 5-cycle E*, 2-cycle E+, branches predicted taken
 - Single iteration (7 insns) latency: 16-5 = 11 cycles
 - **Performance**: 7 insns / 11 cycles = 0.64 IPC
 - **Utilization**: 0.64 actual IPC / 1 peak IPC = 64%

SAXPY Performance and Utilization

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
ldf X(r1) → f1	F	D	X	M	W															
mulf f0, f1 → f2	F	D	d*	d*	E*	E*	E*	E*	E*	W										
ldf Y(r1) → f3	F	D	p*	X	M	W														
addf f2, f3 → f4	F	p*	p*	D	d*	d*	d*	d*	E+	E+	W									
stf f4 → Z(r1)		F	p*	D	p*	p*	p*	p*	d*	X	M	W								
addi r1, 4 → r1			F	p*	p*	p*	p*	p*	D	X	M	W								
blt r1, r2, 0			F	p*	p*	p*	p*	p*	D	d*	X	M	W							
ldf X(r1) → f1										F	D	X	M	W						

- 2-way superscalar pipeline
 - Any two insns per cycle + split integer and floating point pipelines
 - + **Performance**: 7 insns / 10 cycles = 0.70 IPC
 - **Utilization**: 0.70 actual IPC / 2 peak IPC = 35%
 - More hazards → more stalls
 - Each stall is more expensive

(Compiler) Instruction Scheduling

- Idea: place independent insns between slow ops and uses
 - Otherwise, pipeline stalls while waiting for RAW hazards to resolve
 - Have already seen pipeline scheduling
- To schedule well you need ... **independent insns**
- **Scheduling scope**: code region we are scheduling
 - The bigger the better (more independent insns to choose from)
 - Once scope is defined, schedule is pretty obvious
 - Trick is creating a large scope (must schedule across branches)
- Compiler scheduling (really scope enlarging) techniques
 - Loop unrolling (for loops)

Loop Unrolling SAXPY

- Goal: separate dependent insns from one another
- SAXPY problem: not enough flexibility within one iteration
 - Longest chain of insns is 9 cycles
 - Load (1)
 - Forward to multiply (5)
 - Forward to add (2)
 - Forward to store (1)
 - Can't hide a 9-cycle chain using only 7 insns
 - But how about two 9-cycle chains using 14 insns?
- **Loop unrolling**: schedule two or more iterations together
 - Fuse iterations
 - Schedule to reduce stalls
 - Schedule introduces ordering problems, rename registers to fix

Unrolling SAXPY I: Fuse Iterations

- Combine two (in general K) iterations of loop
 - Fuse loop control: induction variable (i) increment + branch
 - Adjust (implicit) induction uses: constants → constants + 4

<pre>ldf X(r1), f1 mulf f0, f1, f2 ldf Y(r1), f3 addf f2, f3, f4 stf f4, Z(r1) addi r1, 4, r1 blt r1, r2, 0 ldf X(r1), f1 mulf f0, f1, f2 ldf Y(r1), f3 addf f2, f3, f4 stf f4, Z(r1) addi r1, 4, r1 blt r1, r2, 0</pre>		<pre>ldf X(r1), f1 mulf f0, f1, f2 ldf Y(r1), f3 addf f2, f3, f4 stf f4, Z(r1) ldf X+4(r1), f1 mulf f0, f1, f2 ldf Y+4(r1), f3 addf f2, f3, f4 stf f4, Z+4(r1) addi r1, 8, r1 blt r1, r2, 0</pre>
--	--	---

Unrolling SAXPY II: Pipeline Schedule

- Pipeline schedule to reduce stalls
 - Have already seen this: pipeline scheduling

```

ldf X(r1), f1
mulf f0, f1, f2
ldf Y(r1), f3
addf f2, f3, f4
stf f4, Z(r1)
ldf X+4(r1), f1
mulf f0, f1, f2
ldf Y+4(r1), f3
addf f2, f3, f4
stf f4, Z+4(r1)
addi r1, 8, r1
blt r1, r2, 0
    
```

```

ldf X(r1), f1
ldf X+4(r1), f1
mulf f0, f1, f2
mulf f0, f1, f2
ldf Y(r1), f3
ldf Y+4(r1), f3
addf f2, f3, f4
addf f2, f3, f4
stf f4, Z(r1)
stf f4, Z+4(r1)
addi r1, 8, r1
blt r1, r2, 0
    
```

Unrolling SAXPY III: Rename Registers

- Pipeline scheduling causes reordering violations
 - Rename registers to correct

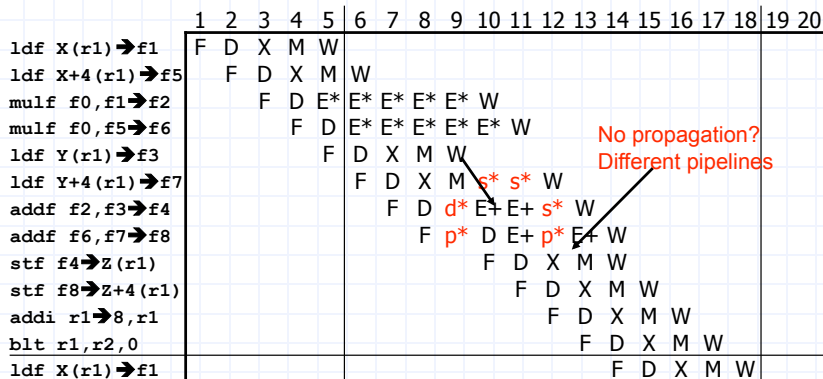
```

ldf X(r1), f1
ldf X+4(r1), f1
mulf f0, f1, f2
mulf f0, f1, f2
ldf Y(r1), f3
ldf Y+4(r1), f3
addf f2, f3, f4
addf f2, f3, f4
stf f4, Z(r1)
stf f4, Z+4(r1)
addi r1, 8, r1
blt r1, r2, 0
    
```

```

ldf X(r1), f1
ldf X+4(r1), f5
mulf f0, f1, f2
mulf f0, f5, f6
ldf Y(r1), f3
ldf Y+4(r1), f7
addf f2, f3, f4
addf f6, f7, f8
stf f4, Z(r1)
stf f8, Z+4(r1)
addi r1, 8, r1
blt r1, r2, 0
    
```

Unrolled SAXPY Performance/Utilization



- + Performance: 12 insn / 13 cycles = 0.92 IPC
- + Utilization: 0.92 actual IPC / 1 peak IPC = 92%
- + **Speedup**: (2 * 11 cycles) / 13 cycles = 1.69

Loop Unrolling Shortcomings

- Static code growth → more I\$ misses (limits degree of unrolling)
- Needs more registers to hold values
- Doesn't handle non-loops...
- **Doesn't handle recurrences** (inter-iteration dependences)

```

for (i=0; i<N; i++)
    X[i]=A*X[i-1];
    
```

```

ldf X-4(r1), f1
mulf f0, f1, f2
stf f2, X(r1)
addi r1, 4, r1
blt r1, r2, 0
ldf X-4(r1), f1
mulf f0, f1, f2
stf f2, X(r1)
addi r1, 4, r1
blt r1, r2, 0
    
```

- Two mulf's are not parallel
- Other (more advanced) techniques help

Anything The Compiler Can Do...

- **Dynamically-scheduled processors**
 - Hardware re-schedules insns...
 - ...within a sliding window of VonNeumann insns
- Does loop unrolling transparently
- Does equivalent of loop unrolling on non-loop code
 - Uses branch prediction to "unroll" branches
- Pentium Pro/II/III (3-wide), Core/2 (4-wide), Alpha 21264 (4-wide), MIPS R10000 (4-wide), Power5 (5-wide)
- Quick overview of approach
 - Lots more information in CIS501 (graduate level architecture)

The Problem With In-Order Pipelines

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
addf f0, f1 → f2	F	D	E	E	E	W										
mulf f2, f3 → f2		F	D	d*	d*	E*	E*	E*	E*	E*	W					
subf f0, f1 → f4			F	p*	p*	D	E	E	E	W						

- What's happening in cycle 4?
 - **mulf** stalls due to **data dependence**
 - OK, this is a fundamental problem
 - **subf** stalls due to **pipeline hazard**
 - Why? **subf** can't proceed into D because **addf** is there
 - That is the only reason, and it isn't a fundamental one
 - Maintaining in-order writes to register file
- Why can't **subf** go into D in cycle 4 and E+ in cycle 5?

Code Example

- Code:

Raw insns

```
add r2, r3 → r1
sub r2, r1 → r3
mul r2, r3 → r3
div r1, 4 → r1
```

- Divide insn independent of subtract and multiply insns
 - Can execute in parallel with subtract
- Many registers re-used
 - Just as in static scheduling, the register names get in the way
 - How does the hardware get around this?
- Approach: (step #1) rename registers, (step #2) schedule

Step #1: Register Renaming

- To eliminate register conflicts/hazards
- "Architected" vs "Physical" registers
 - Names: r1, r2, r3
 - Locations: p1, p2, p3, p4, p5, p6, p7
 - Original mapping: r1 → p1, r2 → p2, r3 → p3, p4-p7 are "available"

MapTable	FreeList	Raw insns	Renamed insns
r1 r2 r3		add r2, r3, r1	add p2, p3, p4
p1 p2 p3	p4, p5, p6, p7	sub r2, r1, r3	sub p2, p4, p5
p4 p2 p3	p5, p6, p7	mul r2, r3, r3	mul p2, p5, p6
p4 p2 p5	p6, p7	div r1, 4, r1	div p4, 4, p7
p4 p2 p6	p7		

- Renaming – conceptually write each register once
 - + Removes **false** dependences
 - + Leaves **true** dependences intact!
- When to reuse a physical register? After overwriting insn done

Register Renaming Algorithm

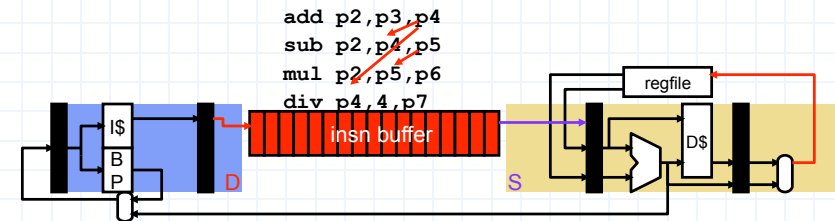
- Data structures:
 - maptable[architectural_reg] → physical_reg
 - Free list: get/put free register
- Algorithm: at decode for each instruction:


```

insn.phys_input1 = maptable[insn.arch_input1]
insn.phys_input2 = maptable[insn.arch_input2]
insn.phys_to_free = maptable[arch_output]
new_reg = get_free_phys_reg()
insn.phys_output = new_reg
maptable[arch_output] = new_reg
            
```
- At "commit"
 - Once all older instructions have committed, free register


```
put_free_phys_reg(insn.phys_to_free)
```

Step #2: Dynamic Scheduling



Ready Table

P2	P3	P4	P5	P6	P7
Yes	Yes				
Yes	Yes	Yes			
Yes	Yes	Yes	Yes		Yes
Yes	Yes	Yes	Yes	Yes	Yes

add p2, p3, p4
sub p2, p4, p5 and div p4, 4, p7
mul p2, p5, p6

- Instructions fetch/decoded/renamed into *Instruction Buffer*
 - Also called "instruction window" or "instruction scheduler"
- Instructions (conceptually) check ready bits every cycle
 - Execute when ready

Dynamic Scheduling Algorithm

- Data structures:
 - Ready table[phys_reg] → yes/no
- Algorithm at "schedule" stage (prior to read registers):


```

foreach instruction:
    if table[insn.phys_input1] == ready &&
        table[insn.phys_input2] == ready then
        insn as "ready"
select the oldest "ready" instruction
table[insn.phys_output] = ready
            
```

Dynamic Scheduling - OoO Execution

- Dynamic scheduling
 - Totally in the hardware
 - Also called "out-of-order execution" (OoO)
- Fetch many instructions into instruction window
 - Use branch prediction to speculate past (multiple) branches
 - Flush pipeline on branch misprediction
- Rename to avoid false dependencies
- Execute instructions as soon as possible
 - Register dependencies are known
 - Handling memory dependencies more tricky
- "Commit" instructions in order
 - Anything strange happens before commit, just flush the pipeline
- Current machines: 100+ instruction scheduling window

Dynamically Scheduling Memory Ops

- Compilers must schedule memory ops conservatively
- Options for hardware:
 - Don't execute any load until all prior stores execute (conservative)
 - Execute loads as soon as possible, detect violations (aggressive)
 - When a store executes, it checks if any later loads executed too early (to same address). If so, flush pipeline
 - Learn violations over time, selectively reorder (predictive)

Before	Wrong(?)
ld r2,4(sp)	ld r2,4(sp)
ld r3,8(sp)	ld r3,8(sp)
add r3,r2,r1 //stall	ld r5,0(r8) //does r8==sp?
st r1,0(sp)	add r3,r2,r1
ld r5,0(r8)	ld r6,4(r8) //does r8+4==sp?
ld r6,4(r8)	st r1,0(sp)
sub r5,r6,r4 //stall	sub r5,r6,r4
st r4,8(r8)	st r4,8(r8)

CIS371 (Roth/Martin): Pipelining

33

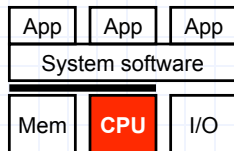
Static vs Dynamic Scheduling

- If we can do this in software...
- ...why build complex (slow-clock, high-power) hardware?
 - + Performance portability
 - Don't want to recompile for new machines
 - + More information available
 - Memory addresses, branch directions
 - + More registers available
 - Compiler may not have enough to schedule well
 - + Speculative memory operation re-ordering
 - Compiler must be conservative, hardware can speculate
 - But compiler has a larger scope
 - Compiler does as much as it can (not much)
 - Hardware does the rest

CIS 501 (Martin/Roth): Dynamic Scheduling I

34

This Unit: Code Scheduling



- Pipelining and superscalar review
- Code scheduling
 - To reduce pipeline stalls
 - To increase ILP (insn level parallelism)
- Two approaches
 - Static scheduling by the compiler
 - Dynamic scheduling by the hardware
- Up next: memory system & caches

CIS 371 (Roth/Martin): Scheduling

35