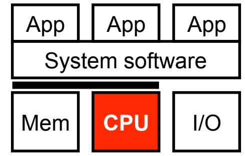
CIS 371
Computer Organization and Design

Unit 11: Static and Dynamic Scheduling

Slides originally developed by Drew Hilton, Amir Roth
and Milo Martin at University of Pennsylvania

This Unit: Static & Dynamic Scheduling



- 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 4.10 – 4.11

Code Scheduling & Limitations

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

```
ld r2,4(sp)
ld r3,8(sp)
add r3,r2,r1   //stall
st r1,0(sp)
ld r5,16(sp)
ld r6,20(sp)
sub r5,r6,r4   //stall
st r4,12(sp)
```

After

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

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

After

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

- One way to create larger scheduling scopes?

Scheduling Scope Limited by Branches

```
loop:
  jz r1, not_found
  ld [r1] -> r2
  sub r1, r2 -> r2
  jz r2, found
  ld [r1+4] -> r1
  jmp loop
```

Aside: what does this code do?

Legal to move load up past branch?

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

```
ld r2,4(sp)
ld r1,8(sp)
add r1,r2,r1   //stall
st r1,0(sp)
ld r2,16(sp)
ld r1,20(sp)
sub r2,r1,r1   //stall
st r1,12(sp)
```

Wrong!

```
ld r2,4(sp)
ld r1,8(sp)
ld r2,16(sp)
add r1,r2,r1   // wrong r2
ld r1,20(sp)
st r1,0(sp)    // wrong r1
sub r2,r1,r1
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 Scheduling Example

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
  - Uses floating point values in "F" registers
  - Uses floating point version of instructions (ldf, addf, mulf, stf, etc.)

```
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
```

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%

Static (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

```
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
```

→

```
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
```

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
  - Use different register names to fix problem

```
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

	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															
`ldf X+4(r1)➔f5`		F	D	X	M	W														
`mulf f0,f1➔f2`			F	D	E*	E*	E*	E*	E*	W										
`mulf f0,f5➔f6`				F	D	E*	E*	E*	E*	E*	W									
`ldf Y(r1)➔f3`					F	D	X	M	W											
`ldf Y+4(r1)➔f7`						F	D	X	M	s*	s*	W								
`addf f2,f3➔f4`							F	D	d*	E+	E+	s*	W							
`addf f6,f7➔f8`								F	p*	D	E+	p*	E+	W						
`stf f4➔Z(r1)`										F	D	X	M	W						
`stf f8➔Z+4(r1)`											F	D	X	M	W					
`addi r1➔8,r1`												F	D	X	M	W				
`blt r1,r2,0`													F	D	X	M	W			
`ldf X(r1)➔f1`														F	D	X	M	W		

+ 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 (ISA limits this)
– 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
```

→

```
ldf X-4(r1),f1
mulf f0,f1,f2
stf f2,X(r1)
mulf f0,f2,f3
stf f3,X+4(r1)
addi r1,4,r1
blt r1,r2,0
```

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

Recap: Static Scheduling Limitations

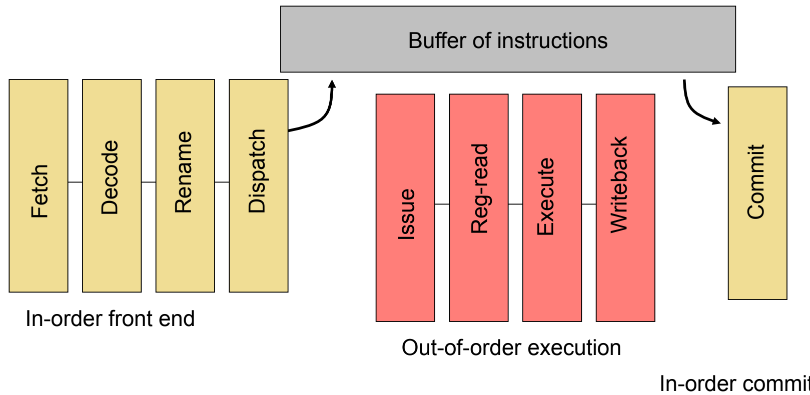
- Limited number of registers (set by ISA)
- Scheduling scope
  - Example: can't generally move memory operations past branches
- Inexact memory aliasing information
  - Often prevents reordering of loads above stores
- Caches misses (or any runtime event) confound scheduling
  - How can the compiler know which loads will miss vs hit?
  - Can impact the compiler's scheduling decisions

Dynamic Scheduling

Can Hardware Overcome These Limits?

- **Dynamically-scheduled processors**
  - Also called "out-of-order" processors
  - Hardware re-schedules insns…
  - …within a sliding window of VonNeumann insns
  - As with pipelining and superscalar, ISA unchanged
    - Same hardware/software interface, appearance of in-order
- Increases scheduling scope
  - Does loop unrolling transparently
  - Uses branch prediction to "unroll" branches
- Examples:
  - Pentium Pro/II/III (3-wide), Core 2 (4-wide), Alpha 21264 (4-wide), MIPS R10000 (4-wide), Power5 (5-wide)
- Basic overview of approach (more information in CIS501)

Out-of-order Pipeline



Limitations of In-Order Pipelines

	0	1	2	3	4	5	6	7	8	9	10	11	12
Ld [r1] -> r2	F	D	X	M_1	M_2	W							
add r2 + r3 -> r4	F	D	**d***	**d***	**d***	X	M_1	M_2	W				
xor r4 ^ r5 -> r6		F	D	**d***	**d***	**d***	X	M_1	M_2	W			
ld [r7] -> r4		F	D	**p***	**p***	**p***	X	M_1	M_2	W			

- In-order pipeline, two-cycle load-use penalty
  - 2-wide
- Why not?

	0	1	2	3	4	5	6	7	8	9	10	11	12
Ld [r1] -> r2	F	D	X	M_1	M_2	W							
add r2 + r3 -> **r4**	F	D	**d***	**d***	**d***	X	M_1	M_2	W				
xor **r4** ^ r5 -> r6		F	D	**d***	**d***	**d***	X	M_1	M_2	W			
ld [r7] -> **r4**		F	D	**X**	**M_1**	**M_2**	**W**						

Limitations of In-Order Pipelines

	0	1	2	3	4	5	6	7	8	9	10	11	12
Ld [p1] -> p2	F	D	X	M_1	M_2	W							
add p2 + p3 -> p4	F	D	**d***	**d***	**d***	X	M_1	M_2	W				
xor p4 ^ p5 -> p6		F	D	**d***	**d***	**d***	X	M_1	M_2	W			
ld [p7] -> p8		F	D	**p***	**p***	**p***	X	M_1	M_2	W			

- In-order pipeline, two-cycle load-use penalty
  - 2-wide
- Why not?

	0	1	2	3	4	5	6	7	8	9	10	11	12
Ld [p1] -> p2	F	D	X	M_1	M_2	W							
add p2 + p3 -> **p4**	F	D	**d***	**d***	**d***	X	M_1	M_2	W				
xor **p4** ^ p5 -> p6		F	D	**d***	**d***	**d***	X	M_1	M_2	W			
ld [p7] -> **p8**		F	D	**X**	**M_1**	**M_2**	**W**						

Out-of-Order to the Rescue

	0	1	2	3	4	5	6	7	8	9	10	11	12
Ld [p1] -> p2	F	Di	I	RR	X	M_1	M_2	W	C				
add p2 + p3 -> p4	F	Di				I	RR	X	W	C			
xor p4 ^ p5 -> p6		F	Di				I	RR	X	W	C		
ld [p7] -> p8		F	Di	I	RR	X	M_1	M_2	W		C		

- "Dynamic scheduling" done by the hardware
- Still 2-wide superscalar, but now out-of-order, too
  - Allows instructions to issues when dependences are ready
- Longer pipeline
  - Front end: Fetch, "Dispatch"
  - Execution core: "Issue", "Reg. Read", Execute, Memory, Writeback
  - Retirement: "Commit"

Code Example

- Code:

Raw insns

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

"Renamed" insns

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

- Difficult to reorder above code, names get in the way
- Divide insn independent of subtract and multiply insns
  - Should be able to 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 – level of indirection
  - 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

r1	r2	r3
p1	p2	p3
p4	p2	p3
p4	p2	p5
p4	p2	p6

FreeList

p4,p5,p6,p7
p5,p6,p7
p6,p7
p7

Original insns	Renamed insns
`add r2,r3,r1`	`add p2,p3,p4`
`sub r2,r1,r3`	`sub p2,p4,p5`
`mul r2,r3,r3`	`mul p2,p5,p6`
`div r1,4,r1`	`div p4,4,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 (implemented as a queue)
- 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()
maptable[arch_output] = new_reg
insn.phys_output = new_reg
```

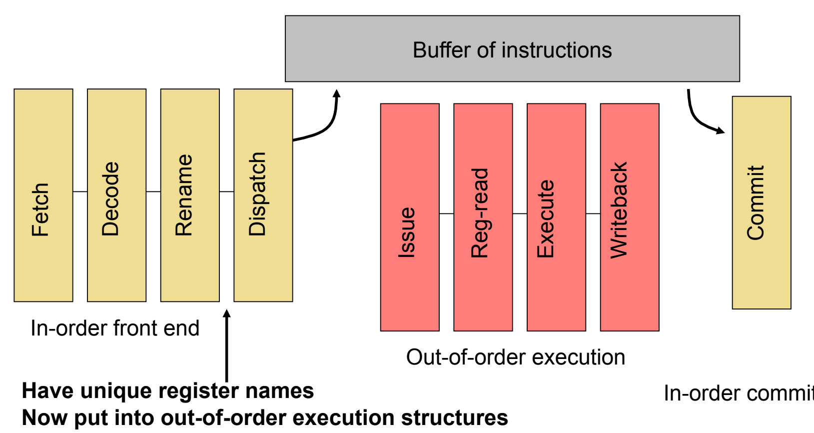
- At "commit"
  - Once all older instructions have committed, free register `put_free_phys_reg(insn.phys_to_free)`

Freeing over-written register

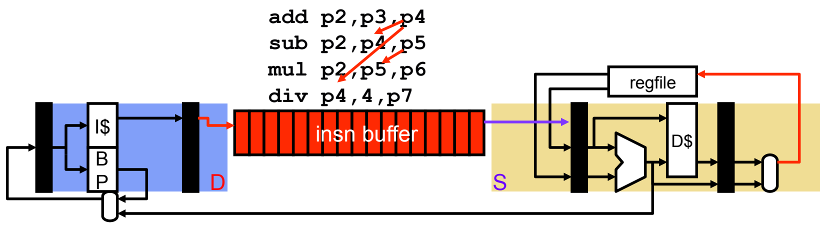
xor r1 ^ r2 -> r3	xor p1 ^ p2 -> p6	[p3]
add r3 + r4 -> r4	add p6 + p4 -> p7	[p4]
sub r5 - r2 -> r3	sub p5 - p2 -> p8	[p6]
addi r3 + 1 -> r1	addi p8 + 1 -> p9	[p1]

- P3 was r3 **before** xor
- P6 is r3 **after** xor
  - Anything older than xor should read p3
  - Anything younger than xor should p6 (until next r3 writing instruction
- At "commit" of xor, no older instructions exist

Out-of-order Pipeline



Step #2: Dynamic Scheduling



Ready Table

Time

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/Issue Algorithm

- Data structures:
  - Ready table[phys_reg] ➔ yes/no (part of "issue queue")
- Algorithm at "schedule" stage (prior to read registers):

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


Dynamic Scheduling Example

Dynamic Scheduling Example

- The following slides are a detailed but concrete example
- Yet, it contains enough detail to be overwhelming
  - Try not to worry about the details
- Focus on the big picture take-away:

Hardware can reorder instructions to extract instruction-level parallelism

Recall: Motivating Example

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [p1] -> p2	F	Di	I	RR	X	M_1	M_2	W	C				
add p2 + p3 -> p4	F	Di				I	RR	X	W	C			
xor p4 ^ p5 -> p6		F	Di				I	RR	X	W	C		
ld [p7] -> p8		F	Di	I	RR	X	M_1	M_2	W		C		

- How would this execution occur cycle-by-cycle?

Out-of-Order Pipeline – Cycle 0

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F												
add r2 + r3 -> r4	F												
xor r4 ^ r5 -> r6													
ld [r7] -> r4													

Map Table

r1	p8
r2	p7
r3	p6
r4	p5
r5	p4
r6	p3
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	---
p10	---
p11	---
p12	---

Reorder Buffer

Insn	To Free	Done?
ld		no
add		no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age

Out-of-Order Pipeline – Cycle 1a

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di											
add r2 + r3 -> r4	F												
xor r4 ^ r5 -> r6													
ld [r7] -> r4													

Map Table

r1	p8
r2	p9
r3	p6
r4	p5
r5	p4
r6	p3
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	no
p10	---
p11	---
p12	---

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add		no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
ld	p8	yes	---	yes	p9	0

Out-of-Order Pipeline – Cycle 1b

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di											
add r2 + r3 -> r4	F	Di											
xor r4 ^ r5 -> r6													
ld [r7] -> r4													

Map Table

r1	p8
r2	p9
r3	p6
r4	p10
r5	p4
r6	p3
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	no
p10	no
p11	---
p12	---

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
ld	p8	yes	---	yes	p9	0
add	p9	no	p6	yes	p10	1

Out-of-Order Pipeline – Cycle 1c

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di											
add r2 + r3 -> r4	F	Di											
xor r4 ^ r5 -> r6		F											
ld [r7] -> r4		F											

Map Table

r1	p8
r2	p9
r3	p6
r4	p10
r5	p4
r6	p3
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	no
p10	no
p11	---
p12	---

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor		no
ld		no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
ld	p8	yes	---	yes	p9	0
add	p9	no	p6	yes	p10	1

Out-of-Order Pipeline – Cycle 2a

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I										
add r2 + r3 -> r4	F	Di											
xor r4 ^ r5 -> r6		F											
ld [r7] -> r4		F											

Map Table

r1	p8
r2	p9
r3	p6
r4	p10
r5	p4
r6	p3
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	no
p10	no
p11	---
p12	---

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor		no
ld		no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
ld	p8	yes	---	yes	p9	0
add	p9	no	p6	yes	p10	1

Out-of-Order Pipeline – Cycle 2b

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I										
add r2 + r3 -> r4	F	Di											
xor r4 ^ r5 -> r6		F	Di										
ld [r7] -> r4		F											

Map Table

r1	p8
r2	p9
r3	p6
r4	p10
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	no
p10	no
p11	no
p12	---

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor	p3	no
ld		no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
add	p9	no	p6	yes	p10	1
xor	p10	no	p4	yes	p11	2

Out-of-Order Pipeline – Cycle 2c

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I										
add r2 + r3 -> r4	F	Di											
xor r4 ^ r5 -> r6		F	Di										
ld [r7] -> r4		F	Di										

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	no
p10	no
p11	no
p12	no

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
add	p9	no	p6	yes	p10	1
xor	p10	no	p4	yes	p11	2
ld	p2	yes	---	yes	p12	3

Out-of-Order Pipeline – Cycle 3

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I	RR									
add r2 + r3 -> r4	F	Di											
xor r4 ^ r5 -> r6		F	Di										
ld [r7] -> r4		F	Di	I									

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	no
p10	no
p11	no
p12	no

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
add	p9	no	p6	yes	p10	1
xor	p10	no	p4	yes	p11	2
ld	p2	yes	---	yes	p12	3

Out-of-Order Pipeline – Cycle 4

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I	RR	X								
add r2 + r3 -> r4	F	Di											
xor r4 ^ r5 -> r6		F	Di										
ld [r7] -> r4		F	Di	I	RR								

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	yes
p10	no
p11	no
p12	no

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
add	p9	yes	p6	yes	p10	1
xor	p10	no	p4	yes	p11	2
~~ld~~	~~p2~~	~~yes~~	~~---~~	~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 5a

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I	RR	X	M_1							
add r2 + r3 -> r4	F	Di				I							
xor r4 ^ r5 -> r6		F	Di										
ld [r7] -> r4		F	Di	I	RR	X							

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	yes
p10	yes
p11	no
p12	no

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
add	p9	yes	p6	yes	p10	1
xor	p10	yes	p4	yes	p11	2
~~ld~~	~~p2~~	~~yes~~	~~---~~	~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 5b

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I	RR	X	M_1							
add r2 + r3 -> r4	F	Di				I							
xor r4 ^ r5 -> r6		F	Di										
ld [r7] -> r4		F	Di	I	RR	X							

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	yes
p10	yes
p11	no
p12	yes

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
xor	p10	yes	p4	yes	p11	2
~~ld~~	~~p2~~	~~yes~~	~~---~~	~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 6

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I	RR	X	M_1	M_2						
add r2 + r3 -> r4	F	Di				I	RR						
xor r4 ^ r5 -> r6		F	Di				I						
ld [r7] -> r4		F	Di	I	RR	X	M_1						

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	yes
p10	yes
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
xor	p10	yes	p4	yes	p11	2
~~ld~~	~~p2~~	~~yes~~		~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 7

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I	RR	X	M_1	M_2	W					
add r2 + r3 -> r4	F	Di				I	RR	X					
xor r4 ^ r5 -> r6		F	Di				I	RR					
ld [r7] -> r4		F	Di	I	RR	X	M_1	M_2					

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	yes
p10	yes
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
ld	p7	yes
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
~~xor~~	~~p10~~	~~yes~~	~~p4~~	~~yes~~	~~p11~~	~~2~~
~~ld~~	~~p2~~	~~yes~~		~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 8a

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I	RR	X	M_1	M_2	W	C				
add r2 + r3 -> r4	F	Di				I	RR	X					
xor r4 ^ r5 -> r6		F	Di				I	RR					
ld [r7] -> r4		F	Di	I	RR	X	M_1	M_2					

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	---
p8	yes
p9	yes
p10	yes
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
~~ld~~	~~p7~~	~~yes~~
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
~~xor~~	~~p10~~	~~yes~~	~~p4~~	~~yes~~	~~p11~~	~~2~~
~~ld~~	~~p2~~	~~yes~~		~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 8b

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I	RR	X	M_1	M_2	W	C				
add r2 + r3 -> r4	F	Di				I	RR	X	W				
xor r4 ^ r5 -> r6		F	Di				I	RR	X				
ld [r7] -> r4		F	Di	I	RR	X	M_1	M_2	W				

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	---
p8	yes
p9	yes
p10	yes
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
~~ld~~	~~p7~~	~~yes~~
add	p5	yes
xor	p3	no
ld	p10	yes

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
~~xor~~	~~p10~~	~~yes~~	~~p4~~	~~yes~~	~~p11~~	~~2~~
~~ld~~	~~p2~~	~~yes~~		~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 9a

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I	RR	X	M_1	M_2	W	C				
add r2 + r3 -> r4	F	Di				I	RR	X	W	C			
xor r4 ^ r5 -> r6		F	Di				I	RR	X				
ld [r7] -> r4		F	Di	I	RR	X	M_1	M_2	W				

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	---
p6	yes
p7	---
p8	yes
p9	yes
p10	yes
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
~~ld~~	~~p7~~	~~yes~~
~~add~~	~~p5~~	~~yes~~
xor	p3	no
ld	p10	yes

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
~~xor~~	~~p10~~	~~yes~~	~~p4~~	~~yes~~	~~p11~~	~~2~~
~~ld~~	~~p2~~	~~yes~~		~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 9b

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I	RR	X	M_1	M_2	W	C				
add r2 + r3 -> r4	F	Di				I	RR	X	W	C			
xor r4 ^ r5 -> r6		F	Di				I	RR	X	W			
ld [r7] -> r4		F	Di	I	RR	X	M_1	M_2	W				

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	---
p6	yes
p7	---
p8	yes
p9	yes
p10	yes
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
~~ld~~	~~p7~~	~~yes~~
~~add~~	~~p5~~	~~yes~~
xor	p3	yes
ld	p10	yes

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
~~xor~~	~~p10~~	~~yes~~	~~p4~~	~~yes~~	~~p11~~	~~2~~
~~ld~~	~~p2~~	~~yes~~		~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 10

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I	RR	X	M_1	M_2	W	C				
add r2 + r3 -> r4	F	Di				I	RR	X	W	C			
xor r4 ^ r5 -> r6		F	Di				I	RR	X	W	C		
ld [r7] -> r4		F	Di	I	RR	X	M_1	M_2	W		C		

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	---
p4	yes
p5	---
p6	yes
p7	---
p8	yes
p9	yes
p10	---
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
~~ld~~	~~p7~~	~~yes~~
~~add~~	~~p5~~	~~yes~~
~~xor~~	~~p3~~	~~yes~~
~~ld~~	~~p10~~	~~yes~~

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
~~xor~~	~~p10~~	~~yes~~	~~p4~~	~~yes~~	~~p11~~	~~2~~
~~ld~~	~~p2~~	~~yes~~		~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Done!

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] -> r2	F	Di	I	RR	X	M_1	M_2	W	C				
add r2 + r3 -> r4	F	Di				I	RR	X	W	C			
xor r4 ^ r5 -> r6		F	Di				I	RR	X	W	C		
ld [r7] -> r4		F	Di	I	RR	X	M_1	M_2	W		C		

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	---
p4	yes
p5	---
p6	yes
p7	---
p8	yes
p9	yes
p10	---
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
~~ld~~	~~p7~~	~~yes~~
~~add~~	~~p5~~	~~yes~~
~~xor~~	~~p3~~	~~yes~~
~~ld~~	~~p10~~	~~yes~~

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Age
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
~~xor~~	~~p10~~	~~yes~~	~~p4~~	~~yes~~	~~p11~~	~~2~~
~~ld~~	~~p2~~	~~yes~~		~~yes~~	~~p12~~	~~3~~

But what about…

More Dynamic Scheduling Mechanisms

- How are physical registers reclaimed?
  - Need to recycle them eventually
- How are branch mispredictions handled?
  - Need to selectively flush instructions
- How are stores handled?
  - If they execute early, but then need to be flushed?
  - Avoid writing cache until "commit"
    - Forward to dependent loads with "load/store queue"
- What about out-of-order stores & loads?
  - What if a store executes "too early"
  - Solution: predict when to execute, speculate, detect violations
- How do we avoid hurting clock frequency?
  - And without using too much energy?

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)
```


Scheduling Redux

- Static scheduling
  - Performed by compiler, limited in several ways
- Dynamic scheduling
  - Performed by the hardware, overcomes limitations
- Static limitation -> Dynamic mitigation
  - Number of registers in the ISA -> register renaming
  - Scheduling scope -> branch prediction & speculation
  - Inexact memory aliasing information -> speculative memory ops
  - Unknown latencies of cache misses -> execute when ready
- Which to do? Compiler does what it can, hardware the rest
  - Why? dynamic scheduling needed to sustain more than 2-way issue
  - **Helps with hiding memory latency**(execute around misses)
  - Intel Core i7 is four-wide execute w/ 128-insn scheduling window
  - Even mobile phones will have dynamic scheduled cores (ARM A9)