CS252
Graduate Computer Architecture
Lecture 6

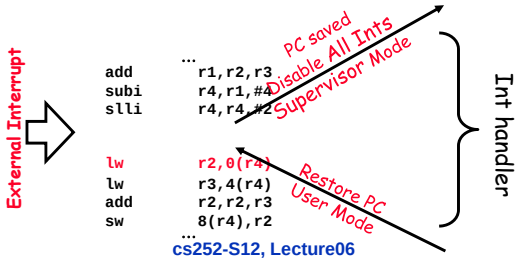
Static Scheduling, Scoreboard
February 6th, 2012

John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs252

Review: Precise Interrupts/Exceptions

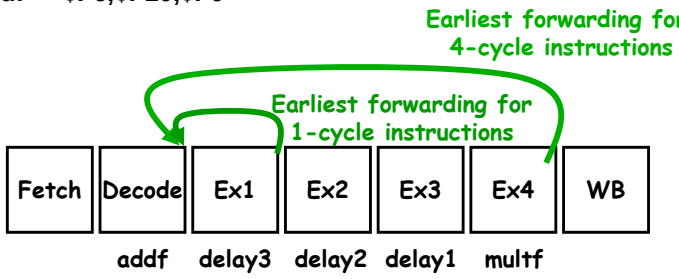
- An interrupt or exception is considered *precise* if there is a single instruction (or interrupt point) for which:
  - All instructions before that have committed their state
  - No following instructions (including the interrupting instruction) have modified any state.
- This means, that you can restart execution at the interrupt point and "get the right answer"
  - Implicit in our previous example of a device interrupt:
    - » Interrupt point is at first lw instruction



Can we make CPI closer to 1?

- Let's assume full pipelining:
  - If we have a 4-cycle *latency*, then we need 3 instructions between a producing instruction and its use:

```
multf   $F0,$F2,$F4
delay-1
delay-2
delay-3
addf    $F6,$F10,$F0
```



FP Loop: Where are the Hazards?

```
Loop:  LD    F0,0(R1)   ;F0=vector element
       ADDD  F4,F0,F2   ;add scalar from F2
       SD    0(R1),F4   ;store result
       SUBI  R1,R1,8    ;decrement pointer 8B (DW)
       BNEZ  R1,Loop    ;branch R1!=zero
       NOP              ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Execution Latency in clock cycles*	*Use Latency in clock cycles*
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	4	2
Load double	FP ALU op	2	1
Load double	Store double	2	0
Integer op	Integer op	1	0

- Where are the stalls?

FP Loop Showing Stalls

```
1 Loop: LD    F0,0(R1)  ;F0=vector element
2       stall
3       ADDD  F4,F0,F2  ;add scalar in F2
4       stall
5       stall
6       SD    0(R1),F4  ;store result
7       SUBI  R1,R1,8   ;decrement pointer 8B (DW)
8       BNEZ  R1,Loop   ;branch R1!=zero
9       stall           ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Use Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- **9 clocks: Rewrite code to minimize stalls?**

Revised FP Loop Minimizing Stalls

```
1 Loop: LD    F0,0(R1)
2       stall
3       ADDD  F4,F0,F2
4       SUBI  R1,R1,8
5       BNEZ  R1,Loop   ;delayed branch
6       SD    8(R1),F4  ;altered when move past SUBI
```

Swap BNEZ and SD by changing address of SD

Instruction producing result	*Instruction using result*	*Use Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

6 clocks: Unroll loop 4 times code to make faster?

Unroll Loop Four Times (straightforward way)

```
1 Loop:LD     F0,0(R1)
2      ADDD   F4,F0,F2
3      SD     0(R1),F4     ;drop SUBI & BNEZ
4      LD     F6,-8(R1)
5      ADDD   F8,F6,F2
6      SD     -8(R1),F8    ;drop SUBI & BNEZ
7      LD     F10,-16(R1)
8      ADDD   F12,F10,F2
9      SD     -16(R1),F12  ;drop SUBI & BNEZ
10     LD     F14,-24(R1)
11     ADDD   F16,F14,F2
12     SD     -24(R1),F16
13     SUBI   R1,R1,#32    ;alter to 4*8
14     BNEZ   R1,LOOP
15     NOP
```

1 cycle stall

2 cycles stall

Rewrite loop to minimize stalls?

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration

Assumes R1 is multiple of 4

Unrolled Loop That Minimizes Stalls

```
1 Loop:LD     F0,0(R1)
2      LD     F6,-8(R1)
3      LD     F10,-16(R1)
4      LD     F14,-24(R1)
5      ADDD   F4,F0,F2
6      ADDD   F8,F6,F2
7      ADDD   F12,F10,F2
8      ADDD   F16,F14,F2
9      SD     0(R1),F4
10     SD     -8(R1),F8
11     SD     -16(R1),F12
12     SUBI   R1,R1,#32
13     BNEZ   R1,LOOP
14     SD     8(R1),F16    ; 8-32 = -24
```

- **What assumptions made when moved code?**
  - OK to move store past SUBI even though changes register
  - OK to move loads before stores: get right data?
  - When is it safe for compiler to do such changes?

14 clock cycles, or 3.5 per iteration

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Superscalar DLX: 2 instructions, 1 FP & 1 anything else
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - More ports for FP registers to do FP load & FP op in a pair

Type	*Pipe Stages*					
Int. instruction	IF	ID	EX	MEM	WB	
FP instruction		IF	ID	EX	MEM	WB
Int. instruction		IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB
Int. instruction			IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	MEM	WB

- 1 cycle load delay expands to 3 instructions in SS
  - instruction in right half can't use it, nor instructions in next slot

Loop Unrolling in Superscalar

	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	LD F0,0(R1)		1
	LD F6,-8(R1)		2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12		8
	SD -24(R1),F16		9
	SUBI R1,R1,#40		10
	BNEZ R1,LOOP		11
	SD -32(R1),F20		12

- Unrolled 5 times to avoid delays (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration (1.5X)

VLIW: Very Large Instruction Word

- Each "instruction" has explicit coding for multiple operations
  - In EPIC, grouping called a "packet"
  - In Transmeta, grouping called a "molecule" (with "atoms" as ops)
- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - » 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16				7
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#48	8
SD -0(R1),F28				BNEZ R1,LOOP	9

Unrolled 7 times to avoid delays

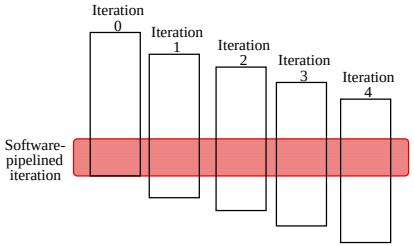
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW (15 vs. 6 in SS)

Another possibility: Software Pipelining

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (Tomasulo in SW)



Software Pipelining Example

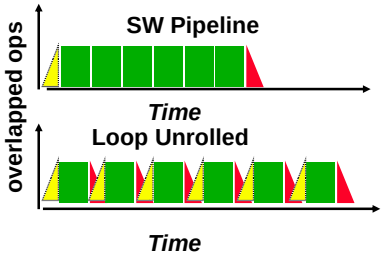
Before: Unrolled 3 times

```
1  LD    F0,0(R1)
2  ADDD  F4,F0,F2
3  SD    0(R1),F4
4  LD    F6,-8(R1)
5  ADDD  F8,F6,F2
6  SD    -8(R1),F8
7  LD    F10,-16(R1)
8  ADDD  F12,F10,F2
9  SD    -16(R1),F12
10 SUBI  R1,R1,#24
11 BNEZ  R1,LOOP
```

After: Software Pipelined

```
1  SD    0(R1),F4 ; Stores M[i]
2  ADDD  F4,F0,F2 ; Adds to M[i-1]
3  LD    F0,-16(R1); Loads M[i-2]
4  SUBI  R1,R1,#8
5  BNEZ  R1,LOOP
```

- Symbolic Loop Unrolling
  - Maximize result-use distance
  - Less code space than unrolling
  - Fill & drain pipe only once per loop vs. once per each unrolled iteration in loop unrolling



5 cycles per iteration

Software Pipelining with Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
LD F0,-48(R1)	ST 0(R1),F4	ADDD F4,F0,F2			1
LD F6,-56(R1)	ST -8(R1),F8	ADDD F8,F6,F2		SUBI R1,R1,#24	2
LD F10,-40(R1)	ST 8(R1),F12	ADDD F12,F10,F2		BNEZ R1,LOOP	3

- Software pipelined across 9 iterations of original loop
  - In each iteration of above loop, we:
    - » Store to m,m-8,m-16 (iterations I-3,I-2,I-1)
    - » Compute for m-24,m-32,m-40 (iterations I,I+1,I+2)
    - » Load from m-48,m-56,m-64 (iterations I+3,I+4,I+5)
- 9 results in 9 cycles, or 1 clock per iteration
- Average: 3.3 ops per clock, 66% efficiency

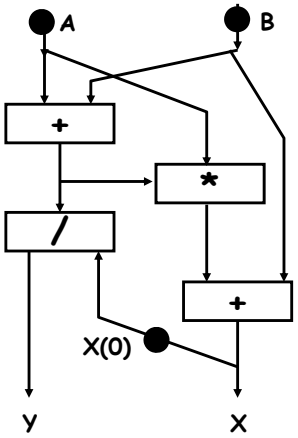
Note: Need less registers for software pipelining (only using 7 registers here, was using 15)

Data-Flow Architectures

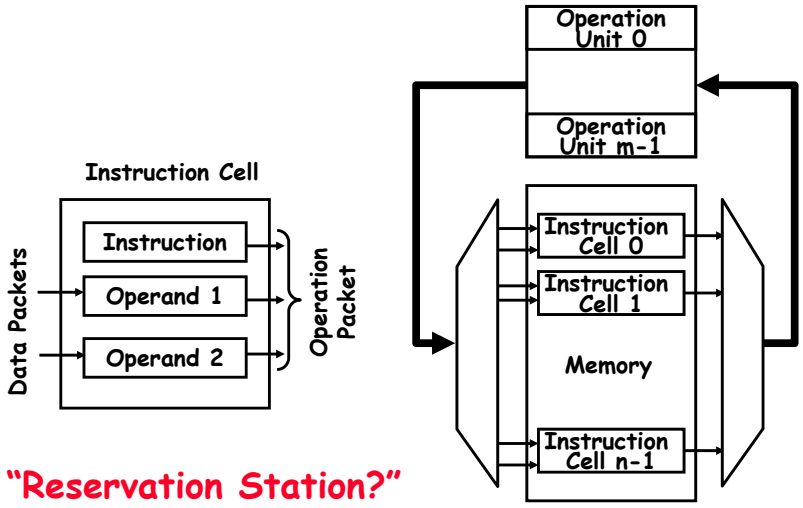
- Basic Idea: Hardware respresents direct encoding of compiler dataflow graphs:

```
Input: a,b
   y:= (a+b)/x
   x:= (a*(a+b))+b
output: y,x
```

- Data flows along arcs in "Tokens".
- When two tokens arrive at compute box, box "fires" and produces new token.
- Split operations produce copies of tokens



Paper by Dennis and Misunas



Compiler Perspectives on Code Movement

- Compiler concerned about dependencies in program
- Whether or not a HW hazard depends on pipeline
- Try to schedule to avoid hazards that cause performance losses
- (True) Data dependencies (RAW if a hazard for HW)
  - Instruction i produces a result used by instruction j, or
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
- If dependent, can't execute in parallel
- Easy to determine for registers (fixed names)
- Hard for memory ("memory disambiguation" problem):
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?

Where are the data dependencies?

```
1 Loop: LD    F0,0(R1)
2       ADDD  F4,F0,F2
3       SUBI  R1,R1,8
4       BNEZ  R1,Loop   ;delayed branch
5       SD    8(R1),F4  ;altered when move past SUBI
```


Compiler Perspectives on Code Movement

- Another kind of dependence called name dependence: two instructions use same name (register or memory location) but don't exchange data
- Antidependence (WAR if a hazard for HW)
  - Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first
- Output dependence (WAW if a hazard for HW)
  - Instruction i and instruction j write the same register or memory location; ordering between instructions must be preserved.

Where are the name dependencies?

```
1 Loop:LD     F0,0(R1)
2      ADDD   F4,F0,F2
3      SD     0(R1),F4     ;drop SUBI & BNEZ
4      LD     F0,-8(R1)
5      ADDD   F4,F0,F2
6      SD     -8(R1),F4    ;drop SUBI & BNEZ
7      LD     F0,-16(R1)
8      ADDD   F4,F0,F2
9      SD     -16(R1),F4   ;drop SUBI & BNEZ
10     LD     F0,-24(R1)
11     ADDD   F4,F0,F2
12     SD     -24(R1),F4
13     SUBI   R1,R1,#32    ;alter to 4*8
14     BNEZ   R1,LOOP
15     NOP
```

How can remove them?

Where are the name dependencies?

```
1 Loop:LD     F0,0(R1)
2      ADDD   F4,F0,F2
3      SD     0(R1),F4     ;drop SUBI & BNEZ
4      LD     F6,-8(R1)
5      ADDD   F8,F6,F2
6      SD     -8(R1),F8    ;drop SUBI & BNEZ
7      LD     F10,-16(R1)
8      ADDD   F12,F10,F2
9      SD     -16(R1),F12  ;drop SUBI & BNEZ
10     LD     F14,-24(R1)
11     ADDD   F16,F14,F2
12     SD     -24(R1),F16
13     SUBI   R1,R1,#32    ;alter to 4*8
14     BNEZ   R1,LOOP
15     NOP
```

Called "register renaming"

Compiler Perspectives on Code Movement

- Name Dependencies are Hard to discover for Memory Accesses
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?
- Our example required compiler to know that if R1 doesn't change then:

  `0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)`

There were no dependencies between some loads and stores so they could be moved by each other

Compiler Perspectives on Code Movement

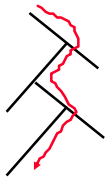
- Final kind of dependence called control dependence. Example:

```
if p1 {S1;};
if p2 {S2;};
```

  S1 is control dependent on p1 and S2 is control dependent on p2 but not on p1.
- Two (obvious?) constraints on control dependences:
  - An instruction that is control dependent on a branch cannot be moved before the branch.
  - An instruction that is not control dependent on a branch cannot be moved to after the branch
- Control dependencies relaxed to get parallelism:
  - Can occasionally move dependent loads before branch to get early start on cache miss
  - get same effect if preserve order of exceptions (address in register checked by branch before use) and data flow (value in register depends on branch)

Trace Scheduling in VLIW

- Parallelism across IF branches vs. LOOP branches
- Two steps:
  - *Trace Selection*
    - » Find likely sequence of basic blocks (*trace*) of (statically predicted or profile predicted) long sequence of straight-line code
  - *Trace Compaction*
    - » Squeeze trace into few VLIW instructions
    - » Need bookkeeping code in case prediction is wrong



- This is a form of compiler-generated speculation
  - Compiler must generate "fixup" code to handle cases in which trace is not the taken branch
  - Needs extra registers: undoes bad guess by discarding
- Subtle compiler bugs mean wrong answer vs. poorer performance; no hardware interlocks

When Safe to Unroll Loop?

- Example: Where are data dependencies? (A,B,C distinct & nonoverlapping)

```
for (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i];    /* S1 */
      B[i+1] = B[i] + A[i+1];  /* S2 */
}
```

1. S2 uses the value, A[i+1], computed by S1 in the same iteration.

2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].

This is a "loop-carried dependence": between iterations

- For our prior example, each iteration was distinct
  - In this case, iterations can't be executed in parallel, Right????

Does a loop-carried dependence mean there is no parallelism???

- Consider:

```
for (i=0; i< 8; i=i+1) {
      A = A + C[i];    /* S1 */
}
```

⇒Could compute:

```
"Cycle 1":  temp0 = C[0] + C[1];
            temp1 = C[2] + C[3];
            temp2 = C[4] + C[5];
            temp3 = C[6] + C[7];
"Cycle 2":  temp4 = temp0 + temp1;
            temp5 = temp2 + temp3;
"Cycle 3":  A = temp4 + temp5;
```

- Relies on associative nature of "+".

Can we use HW to get CPI closer to 1?

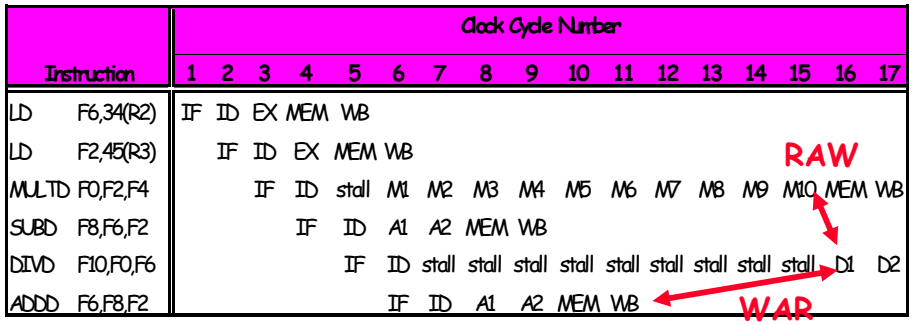
- Why in HW at run time?
  - Works when can't know real dependence at compile time
  - Compiler simpler
  - Code for one machine runs well on another
- Key idea: Allow instructions behind stall to proceed

```
DIVD  F0,F2,F4
ADDD  F10,F0,F8
SUBD  F12,F8,F14
```

- Out-of-order execution => out-of-order completion.

Problems?

- How do we prevent WAR and WAW hazards?
- How do we deal with variable latency?
  - Forwarding for RAW hazards harder.



- How to get precise exceptions?

Summary: Static Scheduling

- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
- Increasing length of pipe increases impact of hazards
  - pipelining helps instruction bandwidth, not latency!
- Instruction Level Parallelism (ILP) found either by compiler or hardware.
- DataFlow view:
  - Data triggers execution rather than instructions triggering data