CS252
Graduate Computer Architecture
Lecture 5

Interrupts,
Software Scheduling around Hazards
February 1st, 2012

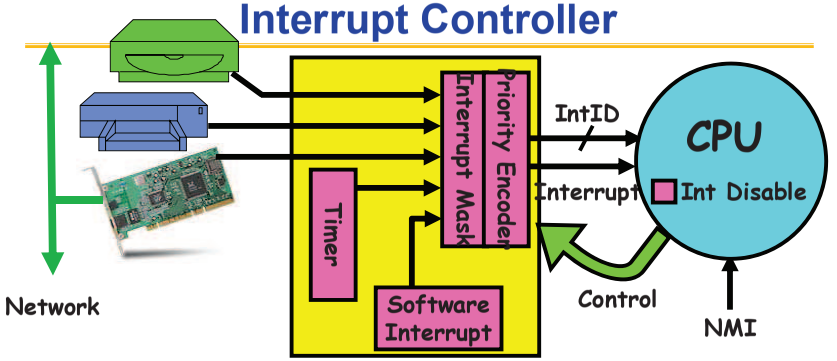
John Kubiatowicz

Electrical Engineering and Computer Sciences

University of California, Berkeley

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

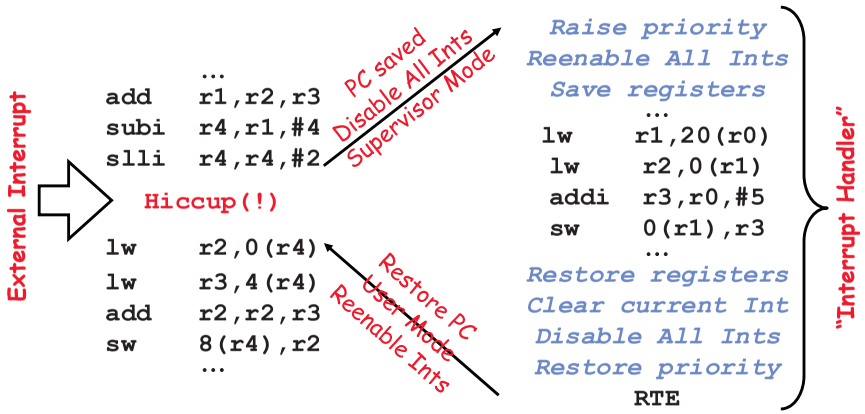
Interrupt Controller



- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-maskable interrupt line (NMI) can't be disabled

Example: Device Interrupt

(Say, arrival of network message)



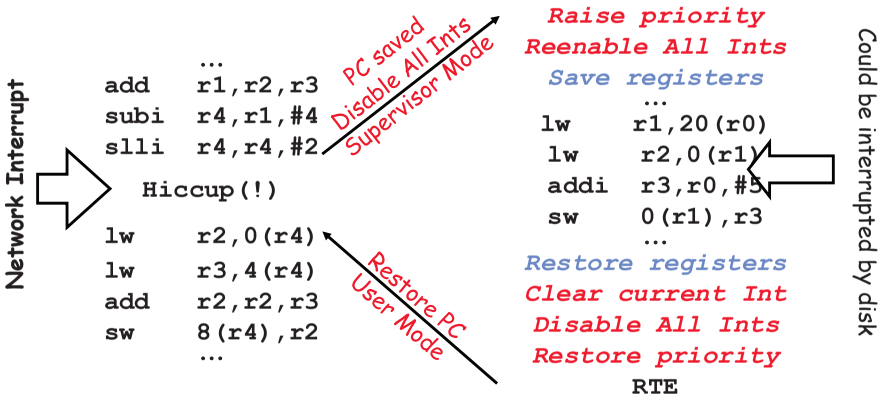
Trap/Interrupt classifications

- ***Traps:*** **relevant to the current process**
  - Faults, arithmetic traps, and synchronous traps
  - Invoke software on behalf of the currently executing process
- ***Interrupts:*** **caused by asynchronous, outside events**
  - I/O devices requiring service (DISK, network)
  - Clock interrupts (real time scheduling)
- ***Machine Checks:*** **caused by serious hardware failure**
  - Not always restartable
  - Indicate that bad things have happened.
    - » Non-recoverable ECC error
    - » Machine room fire
    - » Power outage

A related classification: Synchronous vs. Asynchronous

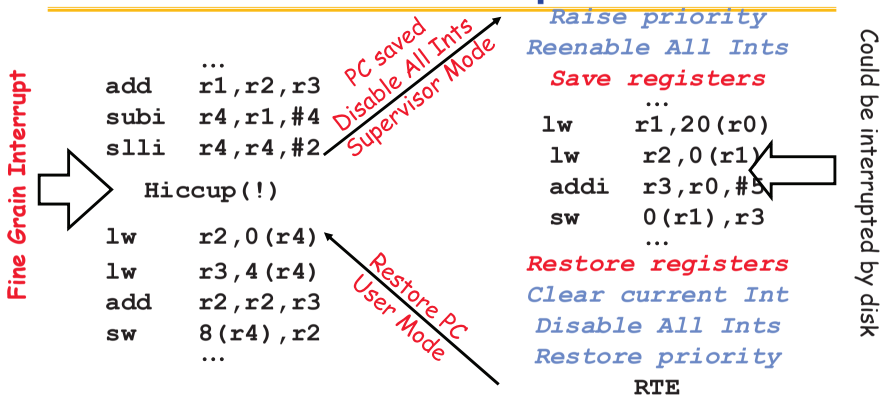
- ***Synchronous:*** **means related to the instruction stream, i.e. during the execution of an instruction**
  - **Must stop an instruction that is currently executing**
  - **Page fault on load or store instruction**
  - **Arithmetic exception**
  - **Software Trap Instructions**
- ***Asynchronous:*** **means unrelated to the instruction stream, i.e. caused by an outside event.**
  - **Does not have to disrupt instructions that are already executing**
  - **Interrupts are asynchronous**
  - **Machine checks are asynchronous**
- ***SemiSynchronous (or high-availability interrupts):***
  - **Caused by external event but may have to disrupt current instructions in order to guarantee service**

Interrupt Priorities Must be Handled



Note that priority must be raised to avoid recursive interrupts!

Can we have fast interrupts?



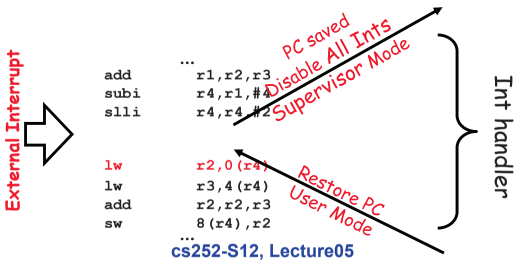
- **Pipeline Drain: Can be very Expensive**
- **Priority Manipulations**
- **Register Save/Restore**
  - **128 registers + cache misses + etc.**

SPARC (and RISC I) had register windows

- **On interrupt or procedure call, simply switch to a different set of registers**
- **Really saves on interrupt overhead**
  - **Interrupts can happen at any point in the execution, so compiler cannot help with knowledge of live registers.**
  - **Conservative handlers must save all registers**
  - **Short handlers might be able to save only a few, but this analysis is compilcated**
- **Not as big a deal with procedure calls**
  - **Original statement by Patterson was that Berkeley didn't have a compiler team, so they used a hardware solution**
  - **Good compilers can allocate registers across procedure boundaries**
  - **Good compilers know what registers are live at any one time**
- **However, register windows have returned!**
  - **IA64 has them**
  - **Many other processors have shadow registers for interrupts**

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



Precise interrupt point may require multiple PCs

```
      addi   r4,r3,#4
      sub    r1,r2,r3
PC:   bne    r1,there
PC+4: and    r2,r3,r5
      <other insts>
```

⇦ Interrupt point described as <PC,PC+4>

```
      addi   r4,r3,#4
      sub    r1,r2,r3
PC:   bne    r1,there
PC+4: and    r2,r3,r5
      <other insts>
```

Interrupt point described as:

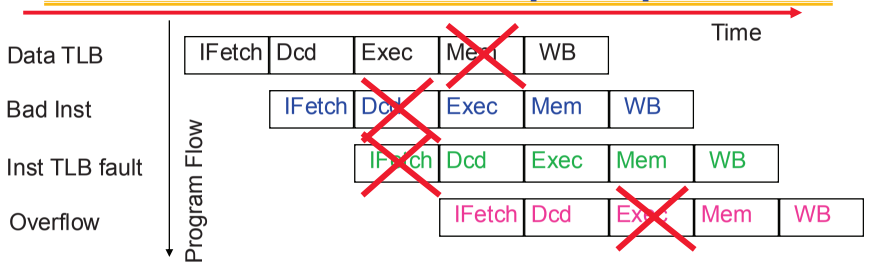
⇦ <PC+4,there> (branch was taken)
or
<PC+4,PC+8> (branch was not taken)

- On SPARC, interrupt hardware produces "pc" and "npc" (next pc)
- On MIPS, only "pc" – must fix point in software

Precise Exceptions in simple 5-stage pipeline:

- Exceptions may occur at different stages in pipeline (I.e. out of order):
  - Arithmetic exceptions occur in execution stage
  - TLB faults can occur in instruction fetch or memory stage
- What about interrupts? The doctor's mandate of "do no harm" applies here: try to interrupt the pipeline as little as possible
- All of this solved by tagging instructions in pipeline as "cause exception or not" and wait until end of memory stage to flag exception
  - Interrupts become marked NOPs (like bubbles) that are placed into pipeline instead of an instruction.
  - Assume that interrupt condition persists in case NOP flushed
  - Clever instruction fetch might start fetching instructions from interrupt vector, but this is complicated by need for supervisor mode switch, saving of one or more PCs, etc

Another look at the exception problem



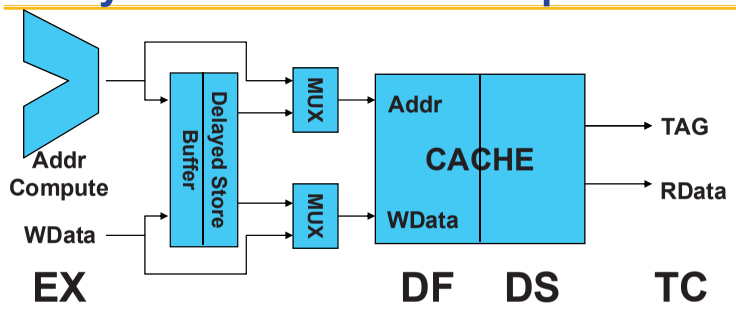
- Use pipeline to sort this out!
  - Pass exception status along with instruction.
  - Keep track of PCs for every instruction in pipeline.
  - Don't act on exception until it reache WB stage
- Handle interrupts through "faulting noop" in IF stage
- When instruction reaches WB stage:
  - Save PC ⇒ EPC, Interrupt vector addr ⇒ PC
  - Turn all instructions in earlier stages into noops!

Impact of Hazards on Performance

Case Study: MIPS R4000 (200 MHz)

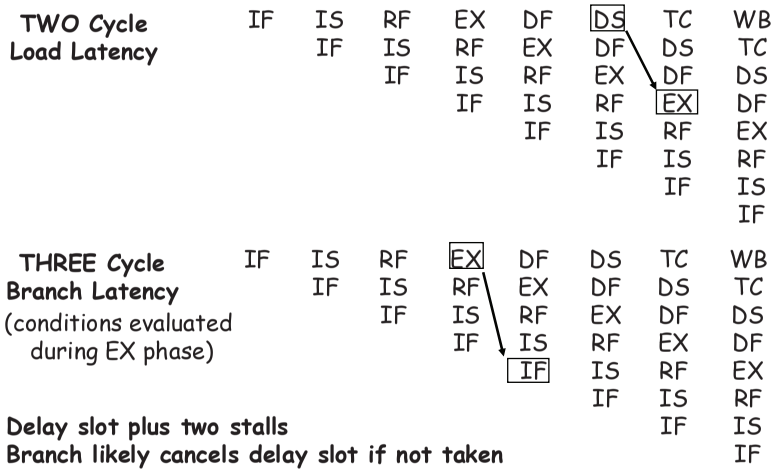
- 8 Stage Pipeline:
  - IF–first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  - IS–second half of access to instruction cache.
  - RF–instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  - EX–execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  - DF–data fetch, first half of access to data cache.
  - DS–second half of access to data cache.
  - TC–tag check, determine whether the data cache access hit.
  - WB–write back for loads and register-register operations.
- Questions:
  - How are stores handled?
  - Why TC stage for data but not instructions?
  - What is impact on Load delay? Branch delay? Why?

Delayed Stores for Write Pipeline



- Store Data placed into buffer until checked by TC stage
- Written to cache when bandwidth available
  - i.e. Written during following store's DF/DS slots or earlier
  - Must be checked against future loads until placed into cache
- Aside: Why TC stage, but no similar stage for Inst Cache?
  - Instruction Cache Tag check done during decode stage (overlapped)
  - Must check Data Cache tag before writeback! (can't be undone)

Case Study: MIPS R4000



R4000 Pipeline: Branch Behavior

Instruction	Clock Number								
	1	2	3	4	5	6	7	8	9
Branch inst	IF	IS	RF	EX	DF	DS	TC	WB	
Delay Slot		IF	IS	RF	EX	DF	DS	TC	WB
Branch Inst+8			IF	IS	null	null	null	null	null
Branch Inst+12				IF	null	null	null	null	null
Branch Targ					IF	IS	RF	EX	DF

- On a *taken branch*, there is a one cycle delay slot, followed by two lost cycles (nullified insts).
- On a *non-taken branch*, there is simply a delay slot (following two cycles *not* lost).
- This is bad for loops. Could benefit from Branch Prediction (later lecture)

MIPS R4000 Floating Point

- FP Adder, FP Multiplier, FP Divider
- Last step of FP Multiplier/Divider uses FP Adder HW
- 8 kinds of stages in FP units:

Stage	*Functional unit*	*Description*
A	FP adder	Mantissa ADD stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

MIPS FP Pipe Stages

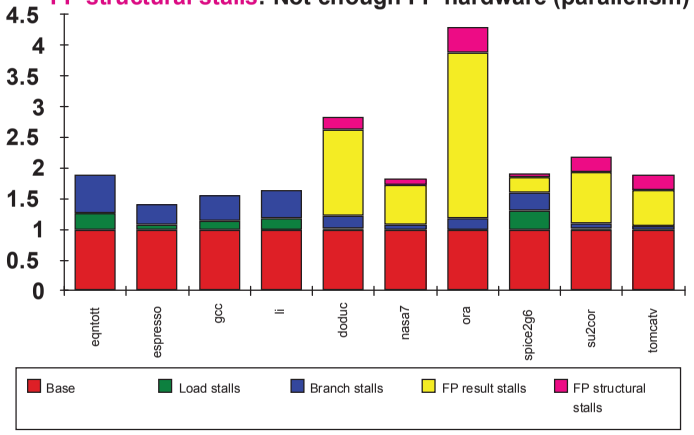
FP Instr	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*…*
Add, Subtract	U	S+A	A+R	R+S					
Multiply	U	E+M	M	M	M	N	N+A	R	
Divide	U	A	R	D^{28}	…	D+A	D+R, D+R, D+A, D+R, A, R		
Square root	U	E	$(A+R)^{108}$	…	A	R			
Negate	U	S							
Absolute value	U	S							
FP compare	U	A	R						

Stages:

M	*First stage of multiplier*	*A*	*Mantissa ADD stage*
N	*Second stage of multiplier*	*D*	*Divide pipeline stage*
R	*Rounding stage*	*E*	*Exception test stage*
S	*Operand shift stage*		
U	*Unpack FP numbers*		

R4000 Performance

- Not ideal CPI of 1:
  - Load stalls (1 or 2 clock cycles)
  - Branch stalls (2 cycles + unfilled slots)
  - FP result stalls: RAW data hazard (latency)
  - FP structural stalls: Not enough FP hardware (parallelism)



CS 252 Administrivia

- Interesting Resource: http://bitsavers.org
  - Has digital versions of users manuals for old machines
  - Quite interesting!
  - I'll link in some of them to your reading pages when it is appropriate
  - Very limited bandwidth: use mirrors such as: http://bitsavers.vt100.net
- Textbook Reading for next few lectures:
  - Computer Architecture: A Quantitative Approach, Chapter 2
- Office hours?
  - Turns out that I cannot have office hours on Monday as planned
  - What about Wednesday 3:00-4:00?

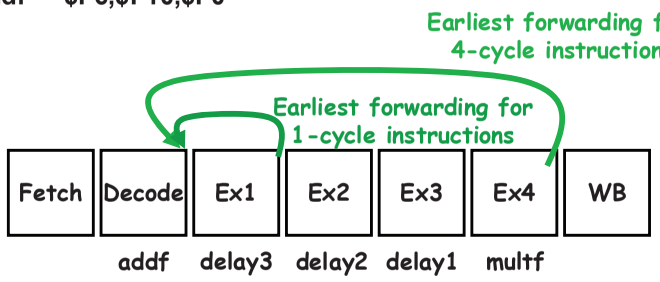
Paper Discussion (Reading #4)

- "Parallel Operation in the CDC 6600," James E. Thorton. *AFIPS Proc.* FJCC, pt. 2 vol. 03, pp. 33-40, 1964
  - Pushed the Load-Store architecture that became staple of RISC
  - Scoreboard for OOO execution (last lecture)
  - Separation of I/O processors from main processor
    - » Memory-mapped communication between them
    - » Very modern ideas
- "The CRAY-1 Computer System," Richard Russel. *Communications of the ACM*, 21(1) 63-72, January 1978
  - Very successful Vector Machine
  - Highly tuned physical implementation
  - No Virtual Memory: segmented protection + relocatable code

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

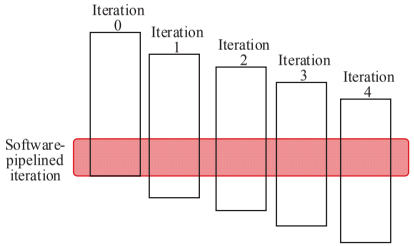
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

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)



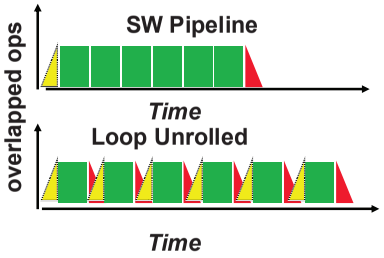
Software Pipelining Example

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)

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:

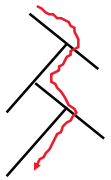
```
if p1 {S1;};
if p2 {S2;};
```

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.

Instruction		Clock Cycle Number																
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD	F6,34(R2)	IF	ID	EX	MEM	WB												
LD	F2,45(R3)		IF	ID	EX	MEM	WB											
MULTD	F0,F2,F4			IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	MEM	WB
SUBD	F8,F6,F2				IF	ID	A1	A2	MEM	WB								
DIVD	F10,F0,F6					IF	ID	stall	stall	stall	stall	stall	stall	stall	stall	stall	D1	D2
ADDD	F6,F8,F2						IF	ID	A1	A2	MEM	WB						

RAW

WAR

- How to get precise exceptions?

Summary

- Machines with precise exceptions provide one single point in the program to restart execution
  - All instructions before that point have completed
  - No instructions after or including that point have completed
- 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.