Lecture 12
Software Scheduling to Exploit more ILP

Instructor: Jun Yang

Example

for (I=1000; I>0; I--)
x[I] = x[I] +s;

- The loop is parallel – every iteration of the body is independent
- Assume the following latencies

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

Where are the hazards?

```
Loop:  LD      F0, 0(R1)
       ADDD    F4, F0, F2
       SD      F4, 0(R1)
       DADDUI  R1, R1, # -8

       BNE     R1, R2, Loop
```

- Without any scheduling …

			Clock cycle issued
Loop:	LD	F0, 0(R1)	1
	stall		2
	ADDD	F4, F0, F2	3
	stall		4
	stall		5
	SD	F4, 0(R1)	6
	DADDUI	R1, R1, # -8	7
	stall		8
	BNE	R1, R2, Loop	9
	stall		10

After Scheduling the Code

Loop:	LD	F0, 0(R1)	1
	stall		2
	ADDD	F4, F0, F2	3
	stall		4
	stall		5
	SD	F4, 0(R1)	6
	DADDUI	R1, R1, # -8	7
	stall		8
	BNE	R1, R2, Loop	9
	stall		10

Loop:	LD	F0, 0(R1)	1
	DADDUI	R1, R1, # -8	2
	ADDD	F4, F0, F2	3
	stall		4
	BNE	R1, R2, Loop	5
	SD	F4, 8(R1)	6

- SD address is altered when it is moved past DADDUI
  - While it is easy for human being, it is non trivial for compilers since swapping SD and DADDUI would create a dependency between them
- 6 clock cycles per iteration now: 3 for loop body (load, add and store) and 3 for loop overhead (decrement, branch and a stall)
- Can we do better? What about 2 iterations (6 cycles) and 1 loop overhead (3 cycles)
- Loop unrolling

Loop Unrolling

- Replicates loop body multiple times, adjusting loop termination code
- Improved scheduling opportunity since instructions from different iterations can be scheduled together
- Eliminate certain number of branches
- But…
- Simply replicating loop body could introduce name dependences since different iterations are using same destination/source registers
- Use different registers for each iterations (register renaming by software) – increases required register count (*register pressure*)
- Increases code size – for embedded applications, this may not be beneficial since the cache miss rate may go up

Unroll Loop Four Times (Straightforward Way)

```
1 Loop:  LD      F0,0(R1)
2        ADDD    F4,F0,F2
3        SD      F4, 0(R1)      ;drop SUBI & BNEZ
4        LD      F6,-8(R1)
5        ADDD    F8,F6,F2
6        SD      F8, -8(R1)     ;drop SUBI & BNEZ
7        LD      F10,-16(R1)
8        ADDD    F12,F10,F2
9        SD      F12, -16(R1)   ;drop SUBI & BNEZ
10       LD      F14,-24(R1)
11       ADDD    F16,F14,F2
12       SD      F16, -24(R1)
13       DADDUI  R1,R1,# -32    ;alter to 4*8
15       BNE     R1, R2, LOOP
16       NOP
```

1 cycle stall
2 cycles stall
1 cycle stall

Schedule loop to minimize stalls.

16 + 4 x (1+2) = 28 clock cycles, or 7 per iteration
Assumes R1 is multiple of 32

- Upper bound *n* of the loop is unknown
  - If we were to unroll a loop *k* times
  - Generate a pair of consecutive loops
  - One loop iterates (*n*/*k*) times. This is the unrolled loop.
  - Another loop iterates (*n* mod *k*) times. This is the original loop.

Scheduled Unrolled Loop

```
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      F4, 0(R1)
10       SD      F8, -8(R1)
11       DADDUI  R1,R1,# -32
12       SD      F12, 16(R1)
13       BNE     R1, R2, LOOP
14       SD      F16, 8(R1); 8-32 = -24
```

- 14 clock cycles, or 3.5 per iteration.
- No stalls
- Actually, unroll 3 times is enough to remove all stalls

Compiler determines when is it safe to move instructions around

- It determines that loop iterations are independent (so that unrolling is useful)
- Adjust loop termination and iteration code
- Determines that ld/st can be interchanged by analyzing memory addresses (they do not refer to the same address)

But …

Problems with loop unrolling

- Large code size
- Register Pressure

Another scheduling method: Software Pipelining

- Goals are:
  - Keep code size small
  - Reduce register pressure
  - And still reduce the stalls
- Reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop.
  - Software version of Tomasulo

Software Pipelining

```
1 LD F0,0(R1)
2 ADDD F4,F0,F2
3 SD F4, 0(R1)
4 LD F6,-8(R1)
5 ADDD F8,F6,F2
6 SD F8, -8(R1)
7 LD F10,-16(R1)
8 ADDD    F12,F10,F2
9 SD F12, -16(R1)
10 LD F14,-24(R1)
11 ADDD F16,F14,F2
12 SD F16, -24(R1)
13 DADDUI R1,R1,# -32
15 BNE R1, R2, LOOP
16 NOP
```

1	LD F0,0(R1)			
2				
3	ADDD F4,F0,F2			
4				
5		LD F0,-8(R1)		
6	SD F4, 0(R1)			
7				
8		ADDD F4,F0,F2		
9				
10			LD F0,-16(R1)	
11		SD F4, -8(R1)		
12				
13			ADDD F4,F0,F2	
14				
15				LD F0, -24(R1)
16			SD F4, -16(R1)	

Software Pipelining

```
1 LD F0,0(R1)
2 ADDD F4,F0,F2
3 SD F4, 0(R1)
4 LD F6,-8(R1)
5 ADDD F8,F6,F2
6 SD F8, -8(R1)
7 LD F10,-16(R1)
8 ADDD    F12,F10,F2
9 SD F12, -16(R1)
10 LD F14,-24(R1)
11 ADDD F16,F14,F2
12 SD F16, -24(R1)
13 DADDUI R1,R1,# -32
15 BNE R1, R2, LOOP
16 NOP
```

1	LD F0,0(R1)			
2				
3	ADDD F4,F0,F2			
4				
5		LD F0,-8(R1)		
6	SD F4, 0(R1)			
7		DADDUI R1,R1,# -8		
8		ADDD F4,F0,F2		
9			BNE R1, R2, Loop	
10			LD F0,-8(R1)	
11		SD F4, 0(R1)		
12			DADDUI R1,R1,# -8	
13			ADDD F4,F0,F2	
14				BNE R1, R2, Loop
15				LD F0, -8(R1)
16			SD F4, 0(R1)	

Software Pipelining

```
Start-up   LD      F0, 0(R1)      (1)
code       ADDD    F4, F0, F2     (1)
           LD      F0, -8(R1)     (2)
Loop:      SD      F4, 0(R1)      (1)
           DADDUI R1, R1 #-8
           ADDD    F4, F0, F2     (2)
           BNE     R1, R2, Loop
           LD      F0, -8(R1)     (3)
Wind-down  SD      F4, 0(R1)      (2)
code       ADDD    F4, F0, F2     (3)
           SD      F4, -16(R1)    (3)
```

- 5 cycles per iteration
- Major pluses:
  - Less code space
  - Fewer registers needed
- Major minuses:
  - Loop overhead in each iteration (loop unrolling reduces this overhead)
- Best performance can come from doing both

We said software scheduling would be good for statically issued processor. Let's apply the unrolled loop to a dual issue MIPS superscalar pipeline

•One instruction can be ld,st,branch or int.

•Another instruction can be FP op.

Unrolled, Scheduled Loop on a Superscalar MIPS

	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 F4, 0(R1)	**ADDD F16,F14,F2**	6
	SD F8, -8(R1)	**ADDD F20, F18, F2**	7
	SD F12, -16(R1)		8
	DADDUI R1, R1, # -40		9
	SD F16, 16(R1)		10
	BNE R1, R2, LOOP		11
	SD F20, 8(R1)		12

Unrolled 5 times to avoid delays (+1 due to SS)

12 clocks, or 2.4 clocks per iteration

Unrolled, Scheduled Loop on a VLIW processor

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer op./branch
LD F0,0(R1)	LD F6,-8(R1)			
LD F10,-16(R1)	LD F14,-24(R1)			
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4, F0, F2	ADDD F8,F6,F2	
LD F26,-48(R1)		ADDD F12, F10, F2	ADDD F16,F14, F2	
		ADDD F20, F18, F2	ADDD F24, F22, F2	
SD F4, 0(R1)	SD F8, -8(R1)	ADDD F28, F26, F2		
SD F12, -16(R1)	SD F16, -24(R1)			DADDUI R1, R1, # -56
SD F20, **24**(R1)	SD F24, **16**(R1)			
SD F28, **8**(R1)				BNE R1, R2, Loop

Assume no branch delay. Unrolled 7 times to avoid delays. 9 clocks, 23 operations, or 2.5 operations per cycle; 1.3 cycles per iteration. Need many registers.

VLIW Processors – The Concept

- Each long instruction contains multiple operations (branches, Loads and Stores, Int/FP operations, reg-reg transfers) that are executed in parallel.
- Execution fully synchronized by a single program counter.
- Tradeoff instruction space for simple decoding.
- No need to track data dependences because compiler packs independent operations into instructions after analyzing dependences.
- Complexity of HW is replaced by complexity in the compiler – need compiling technique that schedules across many branches.
- Compiler performs speculations and recovery.
- Pipelining exposed to the compiler & all machine resources managed by the compiler.
- Memory subsystem is complex - multiple controllers, banks, interleaving etc.

Limits to VLIW Processors

- Duplicate FUs to get parallel execution
- Increase ports to Register File
  - E.g. needs 7 read and 3 write for Int. reg., 5 read and 3 write for FP register
- Increase ports to memory
- VLIW code size: unroll loops + wasted fields in VLIW
- 1 hazard & all instructions stall
- VLIW & binary compatibility is practical weakness

Summary

- Loop unrolling $\Rightarrow$ Multiple iterations of loop in software:
  - Amortizes loop overhead over several iterations
  - Gives more opportunity for scheduling around stalls
- Software Pipelining $\Rightarrow$ take one instruction from each of several iterations of the loop
  - Software overlapping of loop iterations
- Very Long Instruction Word machines (VLIW) $\Rightarrow$ Multiple operations coded in single, long instruction
  - Requires compiler to decide which ops can be done in parallel

Homework

- 3.21 on page 296 – on your own
- 4.11 on page 382
  - Due 11/21 (Friday), in class