

UNIVERSITY OF MASSACHUSETTS
Dept. of Electrical & Computer Engineering

Computer Architecture
ECE 568

Part 10

Compiler Techniques / VLIW

Israel Koren
Fall 2009

ECE568/Koren Part.10 .1

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann

FP Loop Example

- ◆ Add a scalar to a vector:

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

Where are the Hazards?

- First translate into MIPS code:

-To simplify, assume 8 is lowest address

```
Loop: L.D    F0,0(R1) ;F0=vector element
      ADD.D  F4,F0,F2 ;add scalar from F2
      S.D    0(R1),F4 ;store result
      DSUBUI R1,R1,8 ;decrement pointer 8B (DW)
      BNEZ   R1,Loop ;branch R1!=zero
      NOP                    ;delayed branch slot
```

Where are the stalls?

ECE568/Koren Part.10 .2

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann

FP Loop Showing Stalls

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1


```

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

◆ 9 clocks: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

```

1 Loop: L.D   F0, 0(R1)
2           stall
3           ADD.D F4, F0, F2
4           DSUBUI R1, R1, 8
5           BNEZ  R1, Loop ;delayed branch
6           S.D   8(R1), F4 ;altered when move past DSUBUI
    
```

Move S.D after BNEZ by changing address of S.D

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

6 clocks, but just 3 for execution, 3 for loop overhead; How can we make it faster?

Unroll Loop Four Times (straightforward way)

```

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

```

1 cycle stall (pointing to line 1)

2 cycles stall (pointing to line 2)

Rewrite loop to minimize stalls?

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

Assumes R1 is multiple of 4

ECE568/Koren Part.10.5

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann

Unrolled Loop Details

- ◆ Do not usually know upper bound of loop (at compile time)
- ◆ Suppose it is n , and we would like to unroll the loop to make k copies of the body
- ◆ Instead of a single unrolled loop, we generate a pair of consecutive loops:
 - 1st executes $(n \bmod k)$ times and has a body that is the original loop
 - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
 - For large values of n , most of the execution time will be spent in the unrolled loop

ECE568/Koren Part.10.6

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann

Unrolled Loop That Minimizes Stalls

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

◆ What checks needed when moving code?

- OK to move store past DSUBUI 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

Compiler Perspectives on Code Movement

- ◆ Compiler concerned about potential dependencies in program
 - Whether or not a HW hazard depends on pipeline
- ◆ (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 not 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 $20(R6) = 20(R6)$?
- ◆ Our example required compiler to know that if R1 doesn't change then:
 $0(R1) \neq -8(R1) \neq -16(R1) \neq -24(R1)$

Steps Compiler Performed to Unroll

- ◆ Check OK to move the S.D after DSUBUI and BNEZ, and calculate amount to adjust S.D offset
- ◆ Determine unrolling the loop would be useful by finding that the loop iterations are independent (GCD test)
 - Determine loads and stores from different iterations are independent
 - Requires analyzing memory addresses and finding that they do not refer to the same address
- ◆ Rename registers to avoid name dependencies
- ◆ Eliminate extra test and branch instructions and adjust the loop termination and iteration code
- ◆ Schedule the code, preserving any true dependencies needed to yield same result as the original code

When Safe to Unroll Loop?

- ◆ Example: A,B,C distinct & non-overlapping

```
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 */  
}
```

Where are data dependencies?

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

The same is true of S2 for B[i]

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

- ◆ Greatest Common Divisor (GCD) test

$a j + b = c k + d$; $a j - c k = b - d$; denote $x = \text{gcd}(a, c)$ then
 $a = xy$ and $c = xz$; $x(yj - zk) = d - b \Rightarrow yj - zk = (d - b)/x$

$(yj - zk)$ is an integer only if $x = \text{gcd}(a, c)$ divides $(d - b)$

```
for (i=0; i<100; i=i+1) {  
    A[2i] = A[2i-1] + B[i];  
}
```

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 "+".

◆ "Parallelizing Complex Scans and Reductions" by A. Fisher and A. Ghuloum

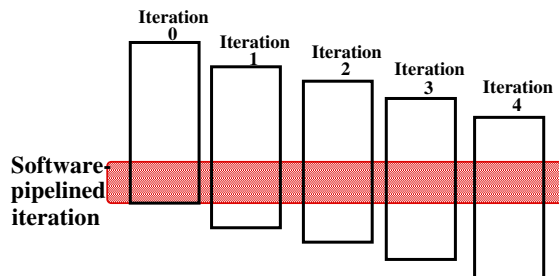
ECE568/Koren Part.10 .11

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann

Another possibility: Software Pipelining

- ◆ Observation: if iterations from loops are independent, we 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)



ECE568/Koren Part.10 .12

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann

Software Pipelining Example

Before: Unrolled 3 times

```

1 L.D  F0,0(R1)
2 ADD.D F4,F0,F2
3 S.D  0(R1),F4
4 L.D  F6,-8(R1)
5 ADD.D F8,F6,F2
6 S.D  -8(R1),F8
7 L.D  F10,-16(R1)
8 ADD.D F12,F10,F2
9 S.D  -16(R1),F12
10 DSUBUI R1,R1,#24
11 BNEZ R1,LOOP
    
```

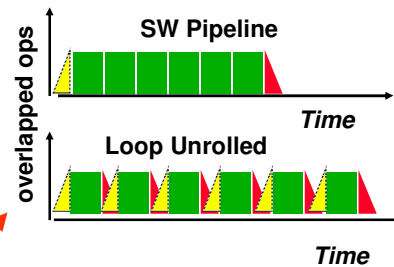
After: Software Pipelined

```

1 S.D  0(R1),F4 ; Stores M[i]
2 ADD.D F4,F0,F2 ; Adds to M[i-1]
3 L.D  F0,-16(R1); Loads M[i-2]
4 DSUBUI 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



ECE568/Koren Part.10 .13

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann

Aggressive Compilers Require Hardware Support

- ◆ **For Exceptions:** Several mechanisms to ensure that speculation by compiler does not violate exception behavior
 - Example: Prefetch should not cause exceptions
- ◆ **For Memory Reference Speculation:** For compiler to move loads across stores, when it cannot be absolutely certain that such a movement is correct, a special instruction to check for address conflicts can be included in the architecture

ECE568/Koren Part.10 .14

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann

What if we Can Change Instruction Set?

- ◆ **Superscalar processors decide on the fly how many instructions to issue**
 - HW complexity of Number of instructions to issue: $O(n^2)$
 - Must therefore limit n
- ◆ **Why not allow compiler to schedule instruction level parallelism explicitly?**
- ◆ **Format the instructions in a potential issue packet so that HW need not check explicitly for dependencies**

VLIW: Very Large Instruction Word

- ◆ **Each "instruction" has explicit coding for multiple operations**
 - In IA-64, grouping called a "packet"
 - In Transmeta, grouping called a "molecule" (with "atoms" as ops)
- ◆ **Tradeoff instruction space for simple decoding/issuing**
 - The long instruction word has room for many operations
 - By definition, all the operations the compiler puts in the long instruction word are independent => can execute in parallel
 - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
 - » 16 to 24 bits per field => $7*16=112$ bits to $7*24=168$ bits wide
 - Need compiling technique that schedules across several branches

Recall: Unrolled Loop that Minimizes Stalls for Scalar

1	Loop:	L.D	F0, 0 (R1)	
2		L.D	F6, -8 (R1)	L.D to ADD.D: 1 Cycle
3		L.D	F10, -16 (R1)	ADD.D to S.D: 2 Cycles
4		L.D	F14, -24 (R1)	
5		ADD.D	F4, F0, F2	
6		ADD.D	F8, F6, F2	
7		ADD.D	F12, F10, F2	
8		ADD.D	F16, F14, F2	
9		S.D	0 (R1), F4	
10		S.D	-8 (R1), F8	
11		S.D	-16 (R1), F12	
12		DSUBUI	R1, R1, #32	
13		BNEZ	R1, LOOP	
14		S.D	8 (R1), F16	; 8-32 = -24

14 clock cycles, or 3.5 per iteration

ECE568/Koren Part.10 .17

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann

Loop Unrolling in VLIW (5 ops per packet)

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

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration (2.3X)

Average: 2.5 ops per clock, 50% efficiency

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

ECE568/Koren Part.10 .18

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann

Software Pipelining with Loop Unrolling in VLIW

<i>Memory reference 1</i>	<i>Memory reference 2</i>	<i>FP operation 1</i>	<i>FP op. 2</i>	<i>Int. op/ branch</i>	<i>Clock</i>
L.D F0,-48(R1)	ST 0(R1),F4	ADD.D F16,F14,F2			1
L.D F6,-56(R1)	ST -8(R1),F8	ADD.D F20,F18,F2		DSUBUI R1,R1,#24	2
L.D F10,-40(R1)	ST 8(R1),F12	ADD.D F24,F22,F2		BNEZ R1,LOOP	3

Software pipelined across 9 iterations of original loop

• In each "iteration" (3 cycles) 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 fewer registers for software pipelining
(only using 12 registers here, was using 15)

ECE568/Koren Part.10 .19

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann

HW (Superscaler w/Tomasulo) vs. SW (VLIW)

♦ HW advantages:

- HW better at memory disambiguation since knows actual addresses
- HW better at branch prediction with low overhead
- HW maintains precise exception model
- Same software works across multiple implementations
- Smaller code size (not as many nops filling blank instructions)

♦ SW advantages:

- Window of instructions that is examined for parallelism much higher
- Speculation can be based on large-scale program behavior, not just local information
- Much less hardware involved in VLIW (for issuing instructions)
- More involved types of speculation can be done more easily

ECE568/Koren Part.10 .20

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann

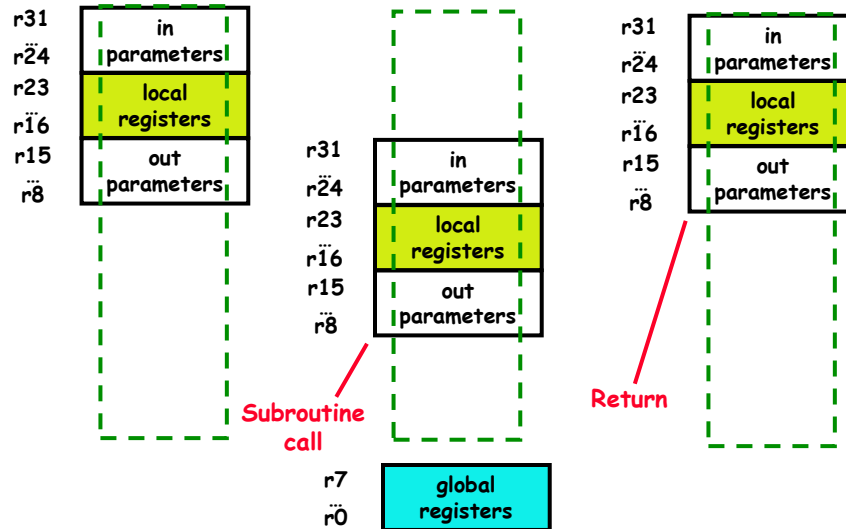
Problems with First Generation VLIW

- ◆ **Increase in code size**
 - Generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
 - Whenever VLIW instructions are not full: unused functional units and wasted bits in instruction
- ◆ **Operated in lock-step**
 - A stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
 - e.g., compiler can predict function units, but caches hard to predict
- ◆ **Binary code incompatibility**
 - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code

Intel/HP IA-64 "Explicitly Parallel Instruction Computer (EPIC)"

- ◆ **IA-64**: instruction set architecture; EPIC is type (2nd generation VLIW?)
- ◆ **Itanium™** is name of first implementation (2001)
 - 6-wide, 10-stage pipeline at 800Mhz
- ◆ 128 64-bit integer reg + 128 82-bit floating point registers
- ◆ Hardware checks dependencies
- ◆ Predicated execution
- ◆ Integer registers configured to accelerate procedure calls using a register stack
 - mechanism similar to that used in SPARC architecture
 - Registers 0-31 are always accessible and addressed as 0-31
 - Registers 32-127 are used as a register stack and each procedure is allocated a set of registers
 - The current frame pointer (CFM) points to the set of registers to be used by a given procedure

SPARC Register Window Mechanism



ECE568/Koren Part.10 .23

Copyright 2003 Koren UMass

Intel/HP IA-64 "Explicitly Parallel Instruction Computer (EPIC)"

- ◆ **Instruction group:** a sequence of consecutive instructions with no register data dependencies
 - All the instructions in a group could be executed in parallel, if sufficient hardware resources existed and if any dependencies through memory were preserved
 - An instruction group can be arbitrarily long, but the compiler must explicitly indicate the boundary between one instruction group and another by placing a **stop** between 2 instructions that belong to different groups
- ◆ IA-64 instructions are encoded in **bundles**, which are 128 bits wide
 - Each bundle consists of a 5-bit template field and 3 instructions, each 41 bits in length
- ◆ 3 Instructions in 128 bit "groups"; field determines if instructions dependent or independent
 - Smaller code size than old VLIW, larger than x86/RISC
 - Groups can be linked to show independence > 3 instructions

ECE568/Koren Part.10 .24

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann

5 Types of Execution in Bundle

<i>Execution Unit Slot</i>	<i>Instruction type</i>	<i>Instruction Description</i>	<i>Example Instructions</i>
I-unit	A	Integer ALU	add, subtract, and, or, cmp
	I	Non-ALU Int	shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, cmp
	M	Memory access	Loads, stores for int/FP regs
F-unit	F	Floating point	Floating point instructions
B-unit	B	Branches	Conditional branches, calls
L+X	L+X	Extended	Extended immediates, stops

- 5-bit template field within each bundle describes both the presence of any **stops** associated with the bundle **and** the execution unit type required by each instruction within the bundle (see Fig 4.12 page 354)

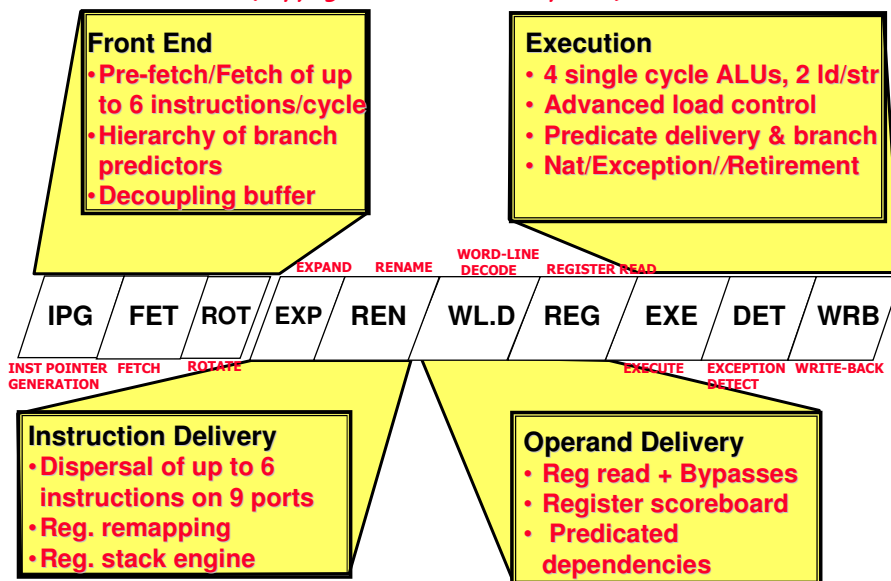
ECE568/Koren Part.10 .25

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann

Itanium™ EPIC: 10 Stage In-Order Core Pipeline

(Copyright: Intel at Hotchips '00)



ECE568/Koren Part.10 .26

Adapted from Patterson, Katz and Culler © UCB

Copyright 2005 UCB & Morgan Kaufmann