
EEC 581 Computer Architecture

Lec 4 – Instruction Level Parallelism

Chansu Yu
Electrical and Computer Engineering
Cleveland State University

Acknowledgement ...

- **Part of class notes are from**
 - David Patterson
 - Electrical Engineering and Computer Sciences
 - University of California, Berkeley

 - <http://www.eecs.berkeley.edu/~pattsrn>
 - <http://www-inst.eecs.berkeley.edu/~cs252>

Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- Tomasulo Algorithm
- Conclusion

9/27/2007

3

Subset of MIPS64

<i>Data transfers</i>	<i>Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR</i>
LB, LBU, SB	Load byte, load byte unsigned, store byte (to/from integer registers)
LH, LHU, SH	Load half word, load half word unsigned, store half word (to/from integer registers)
LW, LMU, SW	Load word, load word unsigned, store word (to/from integer registers)
LD, SD	Load double word, store double word (to/from integer registers)
L.S, L.D, S.S, S.D	Load SP float, load DP float, store SP float, store DP float
MFC0, MTC0	Copy from/to GPR to/from a special register
MOV.S, MOV.D	Copy one SP or DP FP register to another FP register
MFC1, MTC1	Copy 32 bits from/to FP registers to/from integer registers
<i>Arithmetic/logical</i>	<i>Operations on integer or logical data in GPRs; signed arithmetic trap on overflow</i>
DADD, DADDI, DADDU, DADDIU	Add, add immediate (all immediates are 16 bits); signed and unsigned
DSUB, DSUBU	Subtract; signed and unsigned
DMUL, DMULU, DDIV, DDIVU, MADD	Multiply and divide, signed and unsigned; multiply-add; all operations take and yield 64-bit values
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI	Or, or immediate, exclusive or, exclusive or immediate
LUI	Load upper immediate; loads bits 32 to 47 of register with immediate, then sign-extends
DSLL, DSRL, DSRA, DSSLV, DSRLV, DSRAV	Shifts: both immediate (DS_) and variable form (DS_V); shifts are shift left logical, right logical, right arithmetic
SLT, SLTI, SLTU, SLTIU	Set less than, set less than immediate; signed and unsigned

9/27/2007

4

Subset of MIPS64

<i>Control</i>	<i>Conditional branches and jumps; PC-relative or through register</i>
BEQZ, BNEZ	Branch GPR equal/not equal to zero; 16-bit offset from PC + 4
BEQ, BNE	Branch GPR equal/not equal; 16-bit offset from PC + 4
BC1T, BC1F	Test comparison bit in the FP status register and branch; 16-bit offset from PC + 4
MOVN, MOVZ	Copy GPR to another GPR if third GPR is negative, zero
J, JR	Jumps: 26-bit offset from PC + 4 (J) or target in register (JR)
JAL, JALR	Jump and link: save PC + 4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
ERET	Return to user code from an exception; restore user mode
<i>Floating point</i>	<i>FP operations on DP and SP formats</i>
ADD.D, ADD.S, ADD.PS	Add DP, SP numbers, and pairs of SP numbers
SUB.D, SUB.S, ADD.PS	Subtract DP, SP numbers, and pairs of SP numbers
MUL.D, MUL.S, MUL.PS	Multiply DP, SP floating point, and pairs of SP numbers
MADD.D, MADD.S, MADD.PS	Multiply-add DP, SP numbers and pairs of SP numbers
DIV.D, DIV.S, DIV.PS	Divide DP, SP floating point, and pairs of SP numbers
CVT._._	Convert instructions: CVT.x.y converts from type x to type y, where x and y are L (64-bit integer), W (32-bit integer), D (DP), or S (SP). Both operands are FPRs.
C._.D, C._.S	DP and SP compares: “_.” = LT, GT, LE, GE, EQ, NE; sets bit in FP status register

9/27/2007

5

Code Sample

```

Loop: L.D      F0, 0(R1)
      ADD.D    F4, F0, F2
      S.D      F4, 0(R1)
      DADDUI   R1, R1, #-8
      BNE     R1, R2, LOOP
      ...
      BEQZ    R1, LOOP
    
```

9/27/2007

6

Code Sample

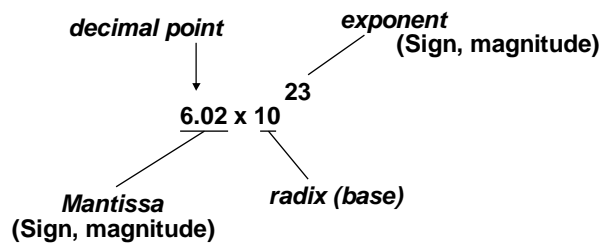
```
Loop: L.D      F0, 0(R1)    ; load floating point
      ADD.D    F4, F0, F2    ; add fp
      S.D      F4, 0(R1)    ; store fp
      DADDUI   R1, R1, #-8   ; add unsigned immediate
      BNE     R1, R2, LOOP  ; branch if not equal
      ...
      BEQZ    R1, LOOP      ; branch if R1=zero
```

9/27/2007

7

Floating Point

- We need a way to represent
 - numbers with fractions, e.g., 3.1416
 - very small numbers, e.g., .000000001
 - very large numbers, e.g., 3.15576×10^9



9/27/2007

8

IEEE 754 Floating-Point Standard

- Standard form for double precision:

$$(-1)^{\text{sign}} \times (1 + \text{mantissa}) \times 2^{\text{exponent} - 1023}$$

- 1-bit for sign, 11-bit for exponent, and 52-bit for mantissa = 64-bit

$$\text{IEEE F.P.} \quad \pm 1.M \times 2^{e - 1023}$$



exponent=e-1023, mantissa=1.M

or

$$(-1)^s \times 1.M \times 2^{e-1023}$$

9/27/2007

9

IEEE 754 Floating-Point Standard

- Example:

– $-0.078125 = -1.01_2 \times 2^{-4}$

– Exponent: $e-1023 = -4$
 $\Rightarrow e=1019=011\ 1111\ 1011_2$

– Mantissa: $1+M = 1.01_2 \Rightarrow M=0.01_2$

– IEEE format: 1 01111111011 010...00

9/27/2007

10

Overview of Chap. 2 & 3

- **Pipelined architecture allows multiple instructions run in parallel (ILP)**
- **But, it has data and control hazard problems**

- **How can we avoid or alleviate the hazard problems in pipelined architecture?**

- **Key idea is to “reorder” the execution of instructions !!!**
 - Partially or entirely
 - By software (compiler, static) or hardware (dynamic)
 - “Dependence analysis” is important

9/27/2007

11

Instruction Level Parallelism

- **Instruction-Level Parallelism (ILP): overlap the execution of instructions to improve performance**

- **2 approaches to exploit ILP:**
 - 1) Rely on hardware to help discover and exploit the parallelism dynamically (e.g., Pentium 4, AMD Opteron, IBM Power) , and
 - 2) Rely on software technology to find parallelism, statically at compile-time (e.g., Itanium 2)

9/27/2007

12

Instruction-Level Parallelism (ILP)

- **Basic Block (BB) ILP is quite small**
 - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
 - average dynamic branch frequency 15% to 25%
=> 4 to 7 instructions execute between a pair of branches
 - Plus instructions in BB likely to depend on each other
- **To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks**
- **Simplest: loop-level parallelism to exploit parallelism among iterations of a loop. E.g.,**
for (i=1; i<=1000; i=i+1)
 x[i] = x[i] + y[i];

9/27/2007

13

Loop-Level Parallelism

- **Exploit loop-level parallelism to parallelism by “unrolling loop” either by**
 - dynamic via branch prediction or
 - static via loop unrolling by compiler
- **Determining instruction dependence is critical to Loop Level Parallelism**
- **If 2 instructions are**
 - parallel, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming no structural hazards)
 - dependent, they are not parallel and must be executed in order, although they may often be partially overlapped

9/27/2007

14

Data Dependence and Hazards

- Instr_j is data dependent (aka true dependence) on Instr_i :
 - Instr_j tries to read operand before Instr_i writes it
- ```
 I: add r1,r2,r3
 ↘
 J: sub r4,r1,r3
```
- or  $\text{Instr}_j$  is data dependent on  $\text{Instr}_k$  which is dependent on  $\text{Instr}_i$
  - If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
  - Data dependence in instruction sequence  
⇒ data dependence in source code ⇒ effect of original data dependence must be preserved
  - If data dependence caused a hazard in pipeline, called a Read After Write (RAW) hazard

9/27/2007

15

## Name Dependence #1: Anti-dependence

---

- Name dependence: when 2 instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name; 2 versions of name dependence
  - $\text{Instr}_j$  writes operand *before*  $\text{Instr}_i$  reads it
- ```
      I: sub r4,r1,r3
     ↘
      J: add r1,r2,r3
      K: mul r6,r1,r7
```
- Called an “anti-dependence” by compiler writers.
This results from reuse of the name “r1”
- If anti-dependence caused a hazard in the pipeline, called a Write After Read (WAR) hazard


9/27/2007

16

Name Dependence #2: Output dependence

- Instr_j writes operand *before* Instr_i writes it.

```
  I: sub r1,r4,r3
  J: add r1,r2,r3
  K: mul r6,r1,r7
```



- Called an “output dependence” by compiler writers
This also results from the reuse of name “r1”
- If anti-dependence caused a hazard in the pipeline,
called a Write After Write (WAW) hazard
- Instructions involved in a name dependence can
execute simultaneously if name used in instructions is
changed so instructions do not conflict
 - Register renaming resolves name dependence for regs
 - Either by compiler or by HW

9/27/2007

17

Control Dependencies

- Every instruction is control dependent on
some set of branches, and, in general, these
control dependencies must be preserved to
preserve program order

```
if p1 {
  s1;
};
if p2 {
  s2;
}
```

- s1 is control dependent on p1, and s2 is
control dependent on p2 but not on p1.


9/27/2007

18

Control Dependence Ignored

- **Control dependence need not be preserved**
 - willing to execute instructions that should not have been executed, thereby violating the control dependences, if we can do so without affecting correctness of the program

```
if p1 {                aa=10;
  aa=10;                ;if p1 {}
};                      if p2 {
if p2 {                aa = 20;
  aa=20;                }
}
```



- **Instead, 2 properties critical to program correctness are**
 - exception behavior and
 - data flow

9/27/2007

19

Exception Behavior

- **Preserving exception behavior**
 - ⇒ any changes in instruction execution order must not change how exceptions are raised in program
 - (⇒ no new exceptions)

- **Example:**

```
DADDU    R2,R3,R4
BEQZ     R2,L1
LW       R1,0(R2)
```

L1:

- (Assume branches not delayed)

- **Problem with moving LW before BEQZ?**

- No data dependence => OK
- Control dependence exists and it could cause memory protection exception (new exception)

9/27/2007

20

Data Flow

- **Data flow: actual flow of data values among instructions that produce results and those that consume them**
 - branches make flow dynamic, determine which instruction is supplier of data
- **Example:**

```
DADDU R1, R2, R3
BEQZ  R4, L
DSUBU R1, R5, R6
L: ...
OR    R7, R1, R8
```

Preserving that order alone is insufficient for correct execution
- **OR depends on DADDU or DSUBU?**
Must preserve data flow on execution

9/27/2007

21

Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- (Start) Tomasulo Algorithm
- Conclusion

9/27/2007

22

Software Techniques - Example

- This code, add a scalar to a vector:

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

- First translate into MIPS code:

```
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
      DADDUI R1,R1,-8   ;decrement pointer 8B
      BNEZ R1,Loop     ;branch R1!=zero
```

9/27/2007

23

FP Loop Showing Stalls

```
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      DADDUI R1,R1,-8 ;decrement pointer 8B (DW)
8      stall ;assumes can't forward to branch
9      BNEZ R1,Loop ;branch R1!=zero
```

Assume these avoidable stall cycles

- 9 clock cycles: Rewrite code to minimize stalls?

9/27/2007

24

Revised FP Loop Minimizing Stalls

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

7 clock cycles,
but just 3 for execution (L.D, ADD.D,S.D),
and 4 for loop overhead;

How can we make it faster?

9/27/2007

25

Unroll Loop

```
1 Loop: L.D    F0,0(R1)
2      DADDUI R1,R1,-8
3      ADD.D  F4,F0,F2
4      stall
5      stall
6      S.D   8(R1),F4
7      BNEZ  R1,Loop
      stall
      stall
```

1 Loop: L.D F0,0(R1) → -8
2 DADDUI R1,R1,-8 → -16
3 ADD.D F4,F0,F2
4 stall
5 stall
6 S.D 8(R1),F4
7 BNEZ R1,Loop

9/27/2007

26

Unroll Loop Four Times (straightforward way)

```

1 Loop:L.D   F0,0(R1)
3   ADD.D  F4,F0,F2
6   S.D    0(R1),F4      ;drop DSUBUI & BNEZ
7   L.D    F6,-8(R1)
9   ADD.D  F8,F6,F2
12  S.D    -8(R1),F8     ;drop DSUBUI & BNEZ
13  L.D    F10,-16(R1)
15  ADD.D  F12,F10,F2
18  S.D    -16(R1),F12   ;drop DSUBUI & BNEZ
19  L.D    F14,-24(R1)
21  ADD.D  F16,F14,F2
24  S.D    -24(R1),F16
25  DADDUI R1,R1,#-32   ;alter to 4*8
26  BNEZ   R1,LOOP
  
```

1 cycle stall (pointing to line 1)

2 cycles stall (pointing to line 3)

Rewrite loop to minimize stalls?

27 clock cycles, or 6.75 per iteration
(Assumes R1 is multiple of 4)

9/27/2007

27

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  S.D    8(R1),F16 ; 8-32 = -24
14  BNEZ   R1,LOOP
  
```

14 clock cycles, or 3.5 per iteration

9/27/2007

28

5 Loop Unrolling Decisions

- Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:
- Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code)
- Use different registers to avoid unnecessary constraints forced by using same registers for different computations
- Eliminate the extra test and branch instructions and adjust the loop termination and iteration code
- Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent
 - Transformation requires analyzing memory addresses and finding that they do not refer to the same address
- Schedule the code, preserving any dependences needed to yield the same result as the original code

9/27/2007

29

3 Limits to Loop Unrolling

- Decrease in amount of overhead amortized with each extra unrolling
- Growth in code size
- Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling

Loop unrolling reduces impact of branches on pipeline; another way is branch prediction (Section 2.3)

9/27/2007

30