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 (2.1)
• Compiler techniques to increase ILP (2.1)
• Loop Unrolling (2.2)
• Static Branch Prediction (2.3)
• Dynamic Branch Prediction (2.3)
• Overcoming Data Hazards with Dynamic Scheduling (2.4)
• Tomasulo Algorithm (2.5)
• Conclusion

Subset of MIPS64

<table>
<thead>
<tr>
<th>Data transfers</th>
<th>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</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB, LH, SH</td>
<td>Load byte, load byte unsigned, store byte (to/from integer registers)</td>
</tr>
<tr>
<td>LH, LW, SW</td>
<td>Load half word, load half word unsigned, store half word (to/from integer registers)</td>
</tr>
<tr>
<td>LD, SD</td>
<td>Load word, load word unsigned, store word (to/from integer registers)</td>
</tr>
<tr>
<td>L.S.L, S,S, I, D</td>
<td>Load double word, store double word (to/from integer registers)</td>
</tr>
<tr>
<td>MFCIC, MFCIC</td>
<td>Load SP float, load DP float, store SP float, store DP float</td>
</tr>
<tr>
<td>MFCIC, MFCIC</td>
<td>Copy from/to GPR to/from a special register</td>
</tr>
<tr>
<td>MFCIC, MFCIC</td>
<td>Copy one SP or DP FP register to another FP register</td>
</tr>
<tr>
<td>MFCIC, MFCIC</td>
<td>Copy 32 bits from/to FP registers to/from integer registers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arithmetic/logical</th>
<th>Operations on integer or logical data is GPR: signed arithmetic trap on overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD, ADDU, ADDU</td>
<td>Add, add immediate (all immediates are 16 bits): signed and unsigned</td>
</tr>
<tr>
<td>SUB, SUBU</td>
<td>Subtract: signed and unsigned</td>
</tr>
<tr>
<td>DMU. DMUI, DIV,</td>
<td>Multiply and divide, signed and unsigned; multiply-add; all operations take and yield 64-bit values</td>
</tr>
<tr>
<td>DIV, MAD</td>
<td>And, and immediate:</td>
</tr>
<tr>
<td>OR, ORI, XOR, XORI</td>
<td>Or, or immediate, exclusive or, exclusive or immediate</td>
</tr>
<tr>
<td>LUI</td>
<td>Load upper immediate; loads bits 32 to 47 of register with immediate, then sign-extends</td>
</tr>
<tr>
<td>DSHL, DSHL, DSRA, DSHL, DSHL</td>
<td>Shifts: both immediate (05 [ ]) and variable form (05 [ ]); shifts are shift left logical, right logical, right arithmetic</td>
</tr>
<tr>
<td>SLT, SLTL, SLTH, SLTIF</td>
<td>Set less than, set less than immediate; signed and unsigned</td>
</tr>
</tbody>
</table>
Subset of MIPS64

<table>
<thead>
<tr>
<th>Control</th>
<th>Conditional branches and jumps: PC-relative or through register</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ, JNZ</td>
<td>Branch GPR equal/not equal to zero; 16-bit offset from PC + 4</td>
</tr>
<tr>
<td>BEQ, JNE</td>
<td>Branch GPR equal/not equal; 16-bit offset from PC + 4</td>
</tr>
<tr>
<td>BGT, JGE</td>
<td>Test comparison bit in the FP status register and branch; 16-bit offset from PC + 4</td>
</tr>
<tr>
<td>MOV, MOVZ</td>
<td>Copy GPR to another GPR if third GPR is negative, zero</td>
</tr>
<tr>
<td>J, JR</td>
<td>Jump: 26-bit offset from PC + 4 (.J) or target in register (.JR)</td>
</tr>
<tr>
<td>JAL, JALR</td>
<td>Jump and link: save PC + 4 in R11, target is PC-relative (JAL) or a register (JALR)</td>
</tr>
<tr>
<td>TRAP</td>
<td>Transfer to operating system at a vectored address</td>
</tr>
<tr>
<td>EXIT</td>
<td>Return to user code from an exception; restore user mode</td>
</tr>
</tbody>
</table>

Floating point

- ADD, ADD.S, ADD.PS: Add DF, SP numbers, and pairs of SP numbers
- SUB, SUB.S, SUB.PS: Subtract DF, SP numbers, and pairs of SP numbers
- MUL, MUL.S, MUL.PS: Multiply DF, SP floating point, and pairs of SP numbers
- MADDS.D, MADDS,S, MADDS.PS: Multiply-add DF, SP numbers and pairs of SP numbers
- DIV, DIV.S, DIV.PS: Divide DF, 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), v (32-bit integer), D (DF), or S (SP). Both operands are FPRs.
- C.__, D, E.__, S: DF and SP compares: “__” = LT, GT, LE, GE, EQ, NE; sets bit in FP status register

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
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

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 x 10^9
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

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

\[
\begin{array}{c|c|c}
\text{s} & e & M \\
\end{array}
\]

- Exponent: e-1023 = -4
  => e=1019=011 1111 1011

- Mantissa: 1+M = 1.01₂
  => M=0.01₂

- IEEE format: 1 0111111011 010...00


IEEE 754 Floating-Point Standard

- Example:
  - -0.078125 = -1.01₂ \times 2^{-4}
  - Exponent: e-1023 = -4
    => e=1019=011 1111 1011₂
  - Mantissa: 1+M = 1.01₂ => M=0.01₂
  - IEEE format: 1 0111111011 010...00
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

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)
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 \( i=1; i<=1000; i=i+1 \)
  \[ x[i] = x[i] + y[i]; \]

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
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 ;assume can’t forward to branch
9 BNEZ R1,Loop ;branch R1!=zero

Assume these stall cycles

• 9 clock cycles: Rewrite code to minimize stalls?

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?
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

Unroll Loop Four Times
(straightforward way)

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

Rewrite loop to minimize stalls?

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

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

5 Loop Unrolling Decisions

- Requires understanding the dependency among instructions and iterations

- Use different registers to avoid unnecessary constraints forced by using same registers for different computations

- Eliminate the extra test and branch instructions (loop termination and iteration code)

- Determine that loads and stores in unrolled loop can be interchanged when loads and stores from different iterations are independent

- Schedule the code, preserving any dependences needed to yield the same result as the original code
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)

Determining **instruction dependence** is critical to Loop Level Parallelism

---

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
    
    \[
    \begin{align*}
    \text{I: } & \text{ add } r1, r2, r3 \\
    \text{J: } & \text{ sub } r4, r1, r3
    \end{align*}
    \]
  - or **Instr** \(j\) is data dependent on **Instr** \(k\) which is dependent on **Instr** \(i\)

- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped

- **Causes a Read After Write (RAW) hazard** (avoidable via “forwarding”)
Name Dependence

- **Name dependence:**
  - when 2 instructions use same register, called a name,
  - but no flow of data between the instructions associated with that name;

- **2 versions of name dependence**
  - Anti-dependence (causes write-after-read or WAR hazard)
  - Output dependence (causes write-after-write or WAW hazard)

- **Simple solution:** change the names used in instructions
  - **Register renaming** resolves name dependence for regs
  - Either by compiler or by HW

- **No trouble in a traditional pipeline, then why study?**

---

Name Dependence #1: Anti-dependence

- **Instr\(_j\)** writes operand **before** **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”

- Causes a **Write After Read (WAR) hazard**
Name Dependence #2: Output dependence

- Instr\textsubscript{J} writes operand \textit{before} Instr\textsubscript{I} writes it.

\begin{align*}
  & I: \text{sub } r1, r4, r3 \\
  & J: \text{add } r1, r2, r3 \\
  & K: \text{mul } r6, r1, r7
\end{align*}

- Called an “\textit{output dependence}” by compiler writers
  This also results from the reuse of name “r1”

- Causes a \textit{Write After Write (WAW) hazard}

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

\begin{verbatim}
  if p1 {
    S1;
  }
  if p2 {
    S2;

  s1 is control dependent on p1, and s2 is control dependent on p2 but not on p1.
\end{verbatim}
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=aa+1e60;  aa=aa+1e60;
}
if !p1 {
    aa=20;  aa=20;
}
```

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

Exception Behavior

- Preserving exception behavior
  \(\implies\) must not cause any new exceptions

- Example:
  ```
  if p1 {
      aa=aa+1e60;  aa=aa+1e60;
  }
  if !p1 {
      aa=20;  aa=20;
  }
  ```

- Problem with eliminating “if p1”?
  - No data dependence \(\implies\) OK
  - What if “aa=aa+1e60” causes an overflow exception?
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:

  \[
  \begin{align*}
  &\text{DADDU} \quad R_1, R_2, R_3 \\
  &\text{BEQZ} \quad R_4, L \\
  &\text{DSUBU} \quad R_1, R_5, R_6 \\
  &L: \quad \ldots \\
  &\text{OR} \quad R_7, R_1, R_8
  \end{align*}
  \]

  Preserving that order alone is insufficient for correct execution

  • OR depends on DADDU or DSUBU?
  Must preserve data flow on execution

Outline

• ILP (2.1)
• Compiler techniques to increase ILP (2.1)
• Loop Unrolling (2.2)
• Static Branch Prediction (2.3)
• Dynamic Branch Prediction (2.3)
• Overcoming Data Hazards with Dynamic Scheduling (2.4)
• Tomasulo Algorithm (2.5)
• Conclusion