Lecture 6
MIPS R4000 and
Instruction Level Parallelism

Computer Architectures
521480S
Case Study: MIPS R4000
(200 MHz, 64-bit instructions, MIPS-3 instruction set)

• 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.

• 8 Stages: What is impact on Load delay? Branch delay? Why?
## Case Study: MIPS R4000

### TWO Cycle Load Latency

<table>
<thead>
<tr>
<th>IF</th>
<th>IS</th>
<th>RF</th>
<th>EX</th>
<th>DS</th>
<th>TC</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>IS</td>
<td>RF</td>
<td>EX</td>
<td>DF</td>
<td>DS</td>
<td>TC</td>
</tr>
<tr>
<td>IF</td>
<td>IS</td>
<td>RF</td>
<td>EX</td>
<td>DF</td>
<td>DS</td>
<td>TC</td>
</tr>
</tbody>
</table>

### THREE Cycle Branch Latency

- Conditions evaluated during EX phase
- Delay slot plus two stalls
- Branch likely cancels delay slot if not taken

<table>
<thead>
<tr>
<th>IF</th>
<th>IS</th>
<th>RF</th>
<th>EX</th>
<th>DF</th>
<th>DS</th>
<th>TC</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>IS</td>
<td>RF</td>
<td>EX</td>
<td>DF</td>
<td>DS</td>
<td>TC</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>IS</td>
<td>RF</td>
<td>EX</td>
<td>DF</td>
<td>DS</td>
<td>TC</td>
<td></td>
</tr>
</tbody>
</table>

(conditions evaluated during EX phase)

Branch likely cancels delay slot if not taken
MIPS R4000 Floating Point

- FP Adder, FP Multiplier, FP Divider
- Require multiple execution stages
- 8 kinds of stages in FP units:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Functional unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>FP adder</td>
<td>Mantissa ADD stage</td>
</tr>
<tr>
<td>D</td>
<td>FP divider</td>
<td>Divide pipeline stage</td>
</tr>
<tr>
<td>E</td>
<td>FP multiplier</td>
<td>Exception test stage</td>
</tr>
<tr>
<td>M</td>
<td>FP multiplier</td>
<td>First stage of multiplier</td>
</tr>
<tr>
<td>N</td>
<td>FP multiplier</td>
<td>Second stage of multiplier</td>
</tr>
<tr>
<td>R</td>
<td>FP adder</td>
<td>Rounding stage</td>
</tr>
<tr>
<td>S</td>
<td>FP adder</td>
<td>Operand shift stage</td>
</tr>
<tr>
<td>U</td>
<td></td>
<td>Unpack FP numbers</td>
</tr>
</tbody>
</table>

- Single copy of each stage: various instructions may use a stage zero or more times in different orders
## MIPS FP Pipe Stages

<table>
<thead>
<tr>
<th>FP Instr</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>… (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, Subtract</td>
<td>U</td>
<td>S+A</td>
<td>A+R</td>
<td>R+S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply</td>
<td>U</td>
<td>E+M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>N</td>
<td>N+A</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>Square root</td>
<td>U</td>
<td>E</td>
<td>(A+R)&lt;sup&gt;108&lt;/sup&gt;</td>
<td>A</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate</td>
<td>U</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute value</td>
<td>U</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP compare</td>
<td>U</td>
<td>A</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**Note:** This is very non-conventional (FP operations share common computational resources)
Latency and Initiation Intervals

• The **latency** is the number of cycles between an instruction that produces a result and an instruction that uses the result (book definition)

• The **initiation interval** is the number of cycles that must elapse before issuing two operations of a given type.

<table>
<thead>
<tr>
<th>FP Instruction</th>
<th>Latency</th>
<th>Initiation interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Subtract</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Multiply</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Divide</td>
<td>36</td>
<td>35</td>
</tr>
<tr>
<td>Square root</td>
<td>112</td>
<td>111</td>
</tr>
<tr>
<td>Negate</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Absolute value</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>FP compare</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
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)
DLX Floating Point Pipeline

- The DLX floating point pipeline is more conventional.
- Add, subtract, and multiply are fully pipelined. Divide is not pipelined.

<table>
<thead>
<tr>
<th>FP Instruction</th>
<th>Latency</th>
<th>Initiation interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Subtract</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Multiply</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Divide</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

- High-performance machines tend to have even shorter latencies

<table>
<thead>
<tr>
<th>FP Instruction</th>
<th>Latency</th>
<th>Initiation interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Subtract</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Multiply</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Divide (DP)</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>
DLX Floating Point Pipeline

If-then-else (IF)

Immediate Data (ID)

Memory Access (MEM)

Write Back (WB)

Integer unit

FP/integer multiply

FP adder

FP/integer divider
Instruction Level Parallelism (ILP)

- **ILP**: Overlap execution of unrelated instructions
- **gcc**: 17% of control transfers
  - 5 instructions + 1 branch (i.e. one branch for every 6 instructions)
  - Beyond single block to get more instruction level parallelism
    » a longer sequential block of code exposes more computation that can be scheduled to minimize stalls
- **Loop level parallelism** one opportunity
- **DLX Floating Point** as example
### Instruction Latencies with Forwarding

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>0</td>
</tr>
</tbody>
</table>

**Note:** for simplicity, these latencies are used in the examples. Furthermore, functional units are assumed to be fully pipelined, so that an operation of any type can be issued on every clock cycle and there are no structural hazards. These modifications can be done without loss of generality.
Simple Loop and Assembly Code

for \( i=1; i\leq 1000; i++ \)  
\[ x(i) = x(i) + s; \]

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 by 8B (DW)  
BNEZ R1,Loop ;branch R1!=zero
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

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>0</td>
</tr>
</tbody>
</table>

• Where are the stalls?
FP Loop Showing Stalls

1 Loop:  LD   F0,0(R1) ;F0=vector element
2         stall ;stall one cycle between LD and FP ALU op
3     ADDD   F4,F0,F2 ;add scalar in F2
4         stall ;stall two cycles between FP ALU op and SD
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

- 9 clock cycles per loop iteration.
- Rewrite code to minimize stalls?
Revised FP Loop Minimizing Stalls

1 Loop:  
1. LD   F0,0(R1)
2. SUBI R1,R1,8
3. ADDD F4,F0,F2 ;
4. NOP  ;Need 2 instr. between FP ALU op and SD
5. BNEZ R1,Loop ;delayed branch
6. SD   8(R1),F4 ;altered when move past SUBI

- Perform SUBI earlier in the loop
- Swap BNEZ and SD by changing address of SD
- Code now takes 6 clock cycles per loop iteration
- Speedup = 10/6 = 1.67
- Unroll loop 4 times 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

• 15 + 4 x 3 + 1= 28 clock cycles, or 7 cycles per iteration
• What does this assume about the number iterations?
Unrolled Loop That Minimizes Stalls

• What assumptions made when moved code?
  – OK to move store past SUBI even though SUBI changes register
  – OK to move loads before stores: get right data?

• When is it safe for compiler to make such changes?

1 Loop: 

1. 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. SUBI R1,R1,#32
12. BNEZ R1,LOOP
13. SD 16(R1),F12 ;16-32 = -16 (book incorrect - pg 227)
14. SD 8(R1),F16 ;8-32 = -24

• 14 clock cycles or 3.5 clock cycles per iteration
• What was the cost for this speedup?
Summary of Loop Unrolling Example

- Determine that it was legal to move the `SD` after the `SUBI` and `BNE` and find the amount to adjust the `SD` offset.
- Determine that unrolling the loop would be useful by finding that the loop iterations are independent, except for the loop maintenance code.
- Use extra registers to avoid unnecessary constraints that would be forced by using the same registers for different computations.
- Eliminate the extra tests and branches and adjust the loop maintenance code.
- Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent. This requires analyzing the memory addresses and finding that they do not refer to the same address, which would create data dependencies between iterations.
- Schedule the code, preserving any dependences needed to yield the same result as the original code.
Compiler Perspectives on Code Movement

- Definitions: compiler concerned about dependencies in program, whether or not a HW hazard depends on a given pipeline
- Try to schedule to avoid hazards
- 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 (i.e. the computation order matters)
- Easy to determine for registers (fixed names)
- Hard for memory:
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?
Where are the data dependencies?

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
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.

Name dependences involving registers can be removed by register renaming.
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

Name dependencies are removed by register renaming
Compiler Perspectives on Code Movement

• Again Name Dependencies are Hard 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:

$$0(R1) \neq -8(R1) \neq -16(R1) \neq -24(R1)$$

There were no dependencies between some loads and stores so they could be moved by each other (i.e. the result of a computation was stored at the same address from where the operand was loaded).
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 so that its execution is no longer controlled by the branch.
  - An instruction that is not control dependent on a branch cannot be moved to after the branch so that its execution is controlled by the branch.
- Sometimes, it is possible to violate these constraints and still get correct execution.
Where are the control dependencies?

1 Loop:  
   1 LD  F0,0(R1)  
   2 ADDD F4,F0,F2  
   3 SD  0(R1),F4  
   4 SUBI R1,R1,8  
   5 BEQZ R1,exit  
   6 LD  F0,0(R1)  
   7 ADDD F4,F0,F2  
   8 SD  0(R1),F4  
   9 SUBI R1,R1,8  
  10 BEQZ R1,exit  
  11 LD  F0,0(R1)  
  12 ADDD F4,F0,F2  
  13 SD  0(R1),F4  
  14 SUBI R1,R1,8  
  15 BEQZ R1,exit  

exit:

What allowed us to remove these control dependencies?
When is it Safe to Unroll Loop?

- Example: Where are data dependencies? (A,B,C distinct & nonoverlapping)
  
  ```c
  for (i=1; 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

- Implies that iterations are dependent, and can’t be executed in parallel

- Not the case for our prior example; each iteration was distinct
When Safe to Unroll Loop?

• Example: Where are data dependencies? (A,B,C,D distinct & nonoverlapping)

```c
for (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i]; /* S1 */
    B[i+1] = C[i] + D[i]; /* S2 */
}
```

1. No dependence from S1 to S2. If there were, then there would be a cycle in the dependencies and the loop would not be parallel. Since this other dependence is absent, interchanging the two statements will not affect the execution of S2.

2. On the first iteration of the loop, statement S1 depends on the value of B[1] computed prior to initiating the loop.
Now Safe to Unroll Loop? (p. 240)

OLD:

for (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i];  /* S1 */
    B[i+1] = C[i] + D[i];} /* S2 */

NEW:

for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}

B[101] = C[100] + D[100];

At interval 2-100 S2 produces a value used by S1 at the same iteration: there is no loop carried dependence any more