Lecture 26: Recap

- Announcements:
  - Assgn 9 (and earlier assignments) will be ready for pick-up from the CS front office later this week
  - Office hours: all day next Tuesday
  - Final exam: Wednesday 13\textsuperscript{th}, 7:50-10am, EMCB 101
  - Same rules as mid-term, except no laptops (open book, open notes/slides/assignments) (print pages from the textbook CD if necessary)
  - 20\% pre-midterm, 80\% post-midterm
  - Advanced course in Spring: CS 7820 Parallel Computer Architecture – more on multi-cores, multi-thread programming, cache coherence and synchronization, interconnection networks
Cache Organizations for Multi-cores

• L1 caches are always private to a core

• L2 caches can be private or shared – which is better?
Cache Organizations for Multi-cores

- L1 caches are always private to a core
- L2 caches can be private or shared

- Advantages of a shared L2 cache:
  - efficient dynamic allocation of space to each core
  - data shared by multiple cores is not replicated
  - every block has a fixed “home” – hence, easy to find the latest copy

- Advantages of a private L2 cache:
  - quick access to private L2 – good for small working sets
  - private bus to private L2 → less contention
View from 5,000 Feet
5-Stage Pipeline and Bypassing

- Some data hazard stalls can be eliminated: bypassing

Must worry about data, control, and structural hazards
Example

\textbf{lw} $1, 8($2)

\textbf{lw} $4, 8($1)
Example

lw  $1, 8($2)

sw  $1, 8($3)
Branch Delay Slots

a. From before

add $s1, $s2, $s3
if $s2 = 0 then
  Delay slot

Becomes

if $s2 = 0 then
  add $s1, $s2, $s3

b. From target

sub $t4, $t5, $t6
...
add $s1, $s2, $s3
if $s1 = 0 then
  Delay slot

Becomes

add $s1, $s2, $s3
if $s1 = 0 then
  sub $t4, $t5, $t6
Pipeline with Branch Predictor

IF (br) -> Reg Read Compare Br-target
Bimodal Predictor

14 bits

Branch PC

Table of 16K entries of 2-bit saturating counters
An Out-of-Order Processor Implementation

- Branch prediction and instr fetch
- Instr Fetch Queue
- Decode & Rename
- Reorder Buffer (ROB)
- Register File R1-R32
- Issue Queue (IQ)
- Results written to ROB and tags broadcast to IQ

Instructions:
- R1 ← R1+R2
- R2 ← R1+R3
- BEQZ R2
- R3 ← R1+R2
- R1 ← R3+R2
- Instr 1
- Instr 2
- Instr 3
- Instr 4
- Instr 5
- Instr 6
- T1 ← R1+R2
- T2 ← R1+R3
- BEQZ T2
- T4 ← T1+T2
- T5 ← T4+T2
- T1 ← T2
- T2 ← T3
- T3 ← T4
- T4 ← T5
- T5 ← T6
- ALU
- ALU
- ALU

Results written to ROB and tags broadcast to IQ
Cache Organization

How many offset/index/tag bits if the cache has 64 sets, each set has 64 bytes, 4 ways

Tag array

Tag

Byte address

10100000

Compare

Way-1

Way-2

Data array

12
Virtual Memory

- The virtual and physical memory are broken up into pages

8KB page size

Virtual address

- virtual page number
- page offset

Translated to physical page number

Physical address
TLB

• Since the number of pages is very high, the page table capacity is too large to fit on chip

• A translation lookaside buffer (TLB) caches the virtual to physical page number translation for recent accesses

• A TLB miss requires us to access the page table, which may not even be found in the cache – two expensive memory look-ups to access one word of data!

• A large page size can increase the coverage of the TLB and reduce the capacity of the page table, but also increases memory wastage
Cache and TLB Pipeline

Virtually Indexed; Physically Tagged Cache
I/O Hierarchy

CPU

Cache

Memory Bus

Memory

I/O Controller

Disk

I/O Bus

Network

USB

DVD

⋯
RAID 3

• Data is bit-interleaved across several disks and a separate disk maintains parity information for a set of bits

• For example: with 8 disks, bit 0 is in disk-0, bit 1 is in disk-1, …, bit 7 is in disk-7; disk-8 maintains parity for all 8 bits

• For any read, 8 disks must be accessed (as we usually read more than a byte at a time) and for any write, 9 disks must be accessed as parity has to be re-calculated

• High throughput for a single request, low cost for redundancy (overhead: 12.5%), low task-level parallelism
RAID 4 and RAID 5

• Data is block interleaved – this allows us to get all our data from a single disk on a read – in case of a disk error, read all 9 disks

• Block interleaving reduces throughput for a single request (as only a single disk drive is servicing the request), but improves task-level parallelism as other disk drives are free to service other requests

• On a write, we access the disk that stores the data and the parity disk – parity information can be updated simply by checking if the new data differs from the old data
RAID 5

• If we have a single disk for parity, multiple writes can not happen in parallel (as all writes must update parity info)

• RAID 5 distributes the parity block to allow simultaneous writes
Example

- P1 reads X: not found in cache-1, request sent on bus, memory responds, X is placed in cache-1 in shared state
- P2 reads X: not found in cache-2, request sent on bus, everyone snoops this request, cache-1 does nothing because this is just a read request, memory responds, X is placed in cache-2 in shared state

- P1 writes X: cache-1 has data in shared state (shared only provides read perms), request sent on bus, cache-2 snoops and then invalidates its copy of X, cache-1 moves its state to modified
- P2 reads X: cache-2 has data in invalid state, request sent on bus, cache-1 snoops and realizes it has the only valid copy, so it downgrades itself to shared state and responds with data, X is placed in cache-2 in shared state
Directory-Based Example

A: Rd X
B: Rd X
C: Rd X
A: Wr X
A: Wr X
C: Wr X
B: Rd X
A: Rd X
A: Rd Y
B: Wr X
B: Rd Y
B: Wr X
B: Wr Y
Basic MIPS Instructions

- lw    $t1, 16($t2)
- add   $t3, $t1, $t2
- addi  $t3, $t3, 16
- sw    $t3, 16($t2)
- beq   $t1, $t2, 16
- blt   is implemented as slt and bne
- j     64
- jr    $t1
- sll   $t1, $t1, 2

Convert to assembly:
while (save[i] == k)
    i += 1;

i and k are in $s3 and $s5 and base of array save[] is in $s6

Loop:  sll  $t1, $s3, 2
       add  $t1, $t1, $s6
       lw   $t0, 0($t1)
       bne  $t0, $s5, Exit
       addi $s3, $s3, 1
       j    Loop

Exit:
Registers

• The 32 MIPS registers are partitioned as follows:
  - Register 0 : $zero    always stores the constant 0
  - Regs 2-3  : $v0, $v1  return values of a procedure
  - Regs 4-7  : $a0-$a3   input arguments to a procedure
  - Regs 8-15 : $t0-$t7   temporaries
  - Regs 16-23: $s0-$s7   variables
  - Regs 24-25: $t8-$t9   more temporaries
  - Reg  28   : $gp      global pointer
  - Reg  29   : $sp      stack pointer
  - Reg  30   : $fp      frame pointer
  - Reg  31   : $ra      return address
Memory Organization

- Stack grows this way
- Static data (globals)
- Dynamic data (heap)
- Text (instructions)
- Proc A’s values
- Proc B’s values
- Proc C’s values
- Stack grows this way
- $fp
- $sp
- $gp
- High address
- Low address
Procedure Calls/Returns

```
procA
{
    int j;
    j = ...;
    call procB(j);
    ... = j;
}

procB (int j)
{
    int k;
    ... = j;
    k = ...;
    return k;
}
```

```
procA:
    $s0 = ... # value of j
    $t0  = ... # some tempval
    $a0  = $s0  # the argument
    ...
    jal procB
    ...
    ... = $v0

procB:
    $t0  = ... # some tempval
    ... = $a0  # using the argument
    $s0  = ... # value of k
    $v0  = $s0;
    jr  $ra
```
Saves and Restores

- Caller saves:
  - $ra, $a0, $t0, $fp
- Callee saves:
  - $s0

- As every element is saved on stack, the stack pointer is decremented
- If the callee’s values cannot remain in registers, they will also be spilled into the stack (don’t have to create space for them at the start of the proc)

procA:
$s0 = ... # value of j
$t0 = ... # some tempval
$a0 = $s0 # the argument
...
jal procB
...
... = $v0

procB:
$t0 = ... # some tempval
... = $a0 # using the argument
$s0 = ... # value of k
$v0 = $s0;
jr $ra
Recap – Numeric Representations

- **Decimal**  \( 35_{10} = 3 \times 10^1 + 5 \times 10^0 \)

- **Binary**  \( 00100011_2 = 1 \times 2^5 + 1 \times 2^1 + 1 \times 2^0 \)

- **Hexadecimal (compact representation)**
  \[ 0 \times 23 \text{ or } 23_{\text{hex}} = 2 \times 16^1 + 3 \times 16^0 \]

- 0-15 (decimal) → 0-9, a-f (hex)

<table>
<thead>
<tr>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>00</td>
<td>4</td>
<td>0100</td>
<td>04</td>
<td>8</td>
<td>1000</td>
<td>08</td>
<td>12</td>
<td>1100</td>
<td>0c</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>01</td>
<td>5</td>
<td>0101</td>
<td>05</td>
<td>9</td>
<td>1001</td>
<td>09</td>
<td>13</td>
<td>1101</td>
<td>0d</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>02</td>
<td>6</td>
<td>0110</td>
<td>06</td>
<td>10</td>
<td>1010</td>
<td>0a</td>
<td>14</td>
<td>1110</td>
<td>0e</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>03</td>
<td>7</td>
<td>0111</td>
<td>07</td>
<td>11</td>
<td>1011</td>
<td>0b</td>
<td>15</td>
<td>1111</td>
<td>0f</td>
</tr>
</tbody>
</table>
2’s Complement

```
0000 0000 0000 0000 0000 0000 0000 0000\textsubscript{two} = 0\textsubscript{ten}
0000 0000 0000 0000 0000 0000 0000 0001\textsubscript{two} = 1\textsubscript{ten}
...
0111 1111 1111 1111 1111 1111 1111 1111\textsubscript{two} = 2^{31} - 1
1000 0000 0000 0000 0000 0000 0000 0000\textsubscript{two} = -2^{31}
1000 0000 0000 0000 0000 0000 0000 0001\textsubscript{two} = -(2^{31} - 1)
1000 0000 0000 0000 0000 0000 0000 0010\textsubscript{two} = -(2^{31} - 2)
...
1111 1111 1111 1111 1111 1111 1111 1110\textsubscript{two} = -2
1111 1111 1111 1111 1111 1111 1111 1111\textsubscript{two} = -1
```

Note that the sum of a number $x$ and its inverted representation $x'$ always equals a string of 1s (-1).

- $x + x' = -1$
- $x' + 1 = -x$ \quad ... hence, can compute the negative of a number by
- $-x = x' + 1$ \quad inverting all bits and adding 1

This format can directly undergo addition without any conversions!

Each number represents the quantity

$$x_{31} \cdot 2^{31} + x_{30} \cdot 2^{30} + x_{29} \cdot 2^{29} + \ldots + x_1 \cdot 2^1 + x_0 \cdot 2^0$$
Multiplication Example

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>$1000_{ten}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>$\times \ 1001_{ten}$</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1000</td>
<td>0000</td>
<td>0000</td>
<td>1000</td>
</tr>
</tbody>
</table>

Product: $1001000_{ten}$

In every step
- multiplicand is shifted
- next bit of multiplier is examined (also a shifting step)
- if this bit is 1, shifted multiplicand is added to the product
Division

Divisor $1000_{ten}$ | $1001010_{ten}$ Dividend

Quotient

-1000

10

101

1010

-1000

10_{ten} Remainder

At every step,
- shift divisor right and compare it with current dividend
- if divisor is larger, shift 0 as the next bit of the quotient
- if divisor is smaller, subtract to get new dividend and shift 1 as the next bit of the quotient
## Division

<table>
<thead>
<tr>
<th>Divisor</th>
<th>$1000_{\text{ten}}$</th>
<th>$1001_{\text{ten}}$</th>
<th>Quotient</th>
<th>Dividend</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001001010</td>
<td>0001001010</td>
<td>0000001010</td>
<td>0000001010</td>
<td></td>
</tr>
<tr>
<td>1000000000000</td>
<td>0001000000</td>
<td>0000100000</td>
<td>0000001000</td>
<td></td>
</tr>
<tr>
<td>Quo: 0</td>
<td>000001</td>
<td>0000010</td>
<td>000001001</td>
<td></td>
</tr>
</tbody>
</table>

At every step,
- shift divisor right and compare it with current dividend
- if divisor is larger, shift 0 as the next bit of the quotient
- if divisor is smaller, subtract to get new dividend and shift 1 as the next bit of the quotient
Binary FP Numbers

• 20.45 decimal = ? Binary

• 20 decimal = 10100 binary

  • 0.45 \times 2 = 0.9 \quad \text{(not greater than 1, first bit after binary point is 0)}
  
  • 0.90 \times 2 = 1.8 \quad \text{(greater than 1, second bit is 1, subtract 1 from 1.8)}
  
  • 0.80 \times 2 = 1.6 \quad \text{(greater than 1, third bit is 1, subtract 1 from 1.6)}
  
  • 0.60 \times 2 = 1.2 \quad \text{(greater than 1, fourth bit is 1, subtract 1 from 1.2)}
  
  • 0.20 \times 2 = 0.4 \quad \text{(less than 1, fifth bit is 0)}
  
  • 0.40 \times 2 = 0.8 \quad \text{(less than 1, sixth bit is 0)}
  
  • 0.80 \times 2 = 1.6 \quad \text{(greater than 1, seventh bit is 1, subtract 1 from 1.6)}

... and the pattern repeats

\[10100.011100110011001100\ldots\]

Normalized form = \[1.0100011100110011\ldots \times 2^4\]
IEEE 754 Format

Final representation: \((-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}\)

- Represent \(-0.75_{\text{ten}}\) in single and double-precision formats

  Single: \((1 + 8 + 23)\)
  \[
  1 \ 0111 \ 1110 \ 1000\ldots000
  \]

  Double: \((1 + 11 + 52)\)
  \[
  1 \ 0111 \ 1111 \ 110 \ 1000\ldots000
  \]

- What decimal number is represented by the following single-precision number?
  \[
  1 \ 1000 \ 0001 \ 01000\ldots0000
  \]
  \[-5.0\]
FP Addition

- Consider the following decimal example (can maintain only 4 decimal digits and 2 exponent digits)

\[9.999 \times 10^1 + 1.610 \times 10^{-1}\]

Convert to the larger exponent:
\[9.999 \times 10^1 + 0.016 \times 10^1\]

Add
\[10.015 \times 10^1\]

Normalize
\[1.0015 \times 10^2\]

Check for overflow/underflow

Round
\[1.002 \times 10^2\]

Re-normalize
Performance Measures

- Performance = 1 / execution time
- Speedup = ratio of performance
- Performance improvement = speedup - 1
- Execution time = clock cycle time x CPI x number of instrs

Program takes 100 seconds on ProcA and 150 seconds on ProcB

Speedup of A over B = 150/100 = 1.5
Performance improvement of A over B = 1.5 – 1 = 0.5 = 50%

Speedup of B over A = 100/150 = 0.66 (speedup less than 1 means performance went down)
Performance improvement of B over A = 0.66 – 1 = -0.33 = -33%
or Performance degradation of B, relative to A = 33%

If multiple programs are executed, the execution times are combined into a single number using AM, weighted AM, or GM
Boolean Algebra

- $A + B = \overline{A} \cdot \overline{B}$

- $A \cdot B = \overline{A} + \overline{B}$

Any truth table can be expressed as a sum of products

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$(A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (C \cdot B \cdot \overline{A})$

- Can also use “product of sums”
- Any equation can be implemented with an array of ANDs, followed by an array of ORs
Adder Implementations

• Ripple-Carry adder – each 1-bit adder feeds its carry-out to next stage – simple design, but we must wait for the carry to propagate thru all bits

• Carry-Lookahead adder – each bit can be represented by an equation that only involves input bits \((a_i, b_i)\) and initial carry-in \((c_0)\) -- this is a complex equation, so it’s broken into sub-parts

For bits \(a_i, b_i,\) and \(c_i,\) a carry is generated if \(a_i \cdot b_i = 1\) and a carry is propagated if \(a_i + b_i = 1\)

\[
C_{i+1} = g_i + p_i \cdot C_i
\]

Similarly, compute these values for a block of 4 bits, then for a block of 16 bits, then for a block of 64 bits….Finally, the carry-out for the 64\(^{th}\) bit is represented by an equation such as this:

\[
C_4 = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3
\]

Each of the sub-terms is also a similar expression
Title

- Bullet