Lecture 15: Recap

• Today’s topics:
  - Recap for mid-term
Modern Trends

Historical contributions to performance:
- Better processes (faster devices) ~20%
- Better circuits/pipelines ~15%
- Better organization/architecture ~15%

Today, annual improvement is closer to 20%; this is primarily because of slowly increasing transistor count and more cores.

Need multi-thread parallelism to boost performance every year.
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
Performance Equations

CPU execution time = CPU clock cycles \times \text{Clock cycle time}

CPU clock cycles = \text{number of instrs} \times \text{avg clock cycles per instruction (CPI)}

Substituting in previous equation,

Execution time = \text{clock cycle time} \times \text{number of instrs} \times \text{avg CPI}

If a 2 GHz processor graduates an instruction every third cycle, how many instructions are there in a program that runs for 10 seconds?
Power Consumption

• Dyn power $\propto$ activity x capacitance x voltage$^2$ x frequency

• Capacitance per transistor and voltage are decreasing, but number of transistors and frequency are increasing at a faster rate

• Leakage power is also rising and will soon match dynamic power

• Power consumption is already around 100W in some high-performance processors today
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
- Dynamic data (heap)
- Static data (globals)
- Text (instructions)

- Proc A’s values
- Proc B’s values
- Proc C’s values

Stack grows this way

$fp$

Low address

High address

$gp$

$sp$
Procedure Calls/Returns

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

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

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

`procB:`
```assembly
$t0 = ... # some tempval
... = $\textit{a0} # using the argument
$s0 = ... # value of k
$v0 = \textit{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.

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
Example 2

```c
int fact (int n)
{
    if (n < 1) return (1);
    else return (n * fact(n-1));
}
```

Notes:
The caller saves $a0 and $ra in its stack space.
Temps are never saved.

```assembly
fact:
    addi $sp, $sp, -8
    sw $ra, 4($sp)
    sw $a0, 0($sp)
    slti $t0, $a0, 1
    beq $t0, $zero, L1
    addi $v0, $zero, 1
    addi $sp, $sp, 8
    jr $ra
L1:
    addi $a0, $a0, -1
    jal fact
    lw $a0, 0($sp)
    lw $ra, 4($sp)
    addi $sp, $sp, 8
    mul $v0, $a0, $v0
    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):
  \(0x\ 23 \quad \text{or} \quad 23_{\text{hex}} = 2 \times 16^1 + 3 \times 16^0\)

0-15 (decimal) \(\rightarrow\) 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

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

- $x + x' = -1$
- $x' + 1 = -x$

... hence, can compute the negative of a number by

- $-x = x' + 1$ inverting all bits and adding 1

This format can directly undergo addition without any conversions!

Each number represents the quantity

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

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>$1000_{\text{ten}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>x $1001_{\text{ten}}$</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------</td>
</tr>
<tr>
<td></td>
<td>---------------------</td>
</tr>
<tr>
<td></td>
<td>$1000$</td>
</tr>
<tr>
<td></td>
<td>$0000$</td>
</tr>
<tr>
<td></td>
<td>$0000$</td>
</tr>
<tr>
<td></td>
<td>$1000$</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Product</td>
<td>$1001000_{\text{ten}}$</td>
</tr>
</tbody>
</table>

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
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

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
Divide Example

- Divide $7_{10} (0000\ 0111_{two})$ by $2_{10} (0010_{two})$

<table>
<thead>
<tr>
<th>Iter</th>
<th>Step</th>
<th>Quot</th>
<th>Divisor</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial values</td>
<td>0000</td>
<td>0010\ 0000</td>
<td>0000\ 0111</td>
</tr>
<tr>
<td>1</td>
<td>Rem = Rem – Div</td>
<td>0000</td>
<td>0010\ 0000</td>
<td>1110\ 0111</td>
</tr>
<tr>
<td></td>
<td>Rem &lt; 0 $\Rightarrow$ +Div, shift 0 into Q</td>
<td>0000</td>
<td>0010\ 0000</td>
<td>0000\ 0111</td>
</tr>
<tr>
<td></td>
<td>Shift Div right</td>
<td>0000</td>
<td>0001\ 0000</td>
<td>0000\ 0111</td>
</tr>
<tr>
<td>2</td>
<td>Same steps as 1</td>
<td>0000</td>
<td>0001\ 0000</td>
<td>1111\ 0111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000</td>
<td>0001\ 0000</td>
<td>0000\ 0111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000</td>
<td>0000\ 1000</td>
<td>0000\ 0111</td>
</tr>
<tr>
<td>3</td>
<td>Same steps as 1</td>
<td>0000</td>
<td>0000\ 0100</td>
<td>0000\ 0111</td>
</tr>
<tr>
<td>4</td>
<td>Rem = Rem – Div</td>
<td>0000</td>
<td>0000\ 0100</td>
<td>0000\ 0011</td>
</tr>
<tr>
<td></td>
<td>Rem $\geq$ 0 $\Rightarrow$ shift 1 into Q</td>
<td>0001</td>
<td>0000\ 0100</td>
<td>0000\ 0011</td>
</tr>
<tr>
<td></td>
<td>Shift Div right</td>
<td>0001</td>
<td>0000\ 0010</td>
<td>0000\ 0011</td>
</tr>
<tr>
<td>5</td>
<td>Same steps as 4</td>
<td>0011</td>
<td>0000\ 0001</td>
<td>0000\ 0001</td>
</tr>
</tbody>
</table>
Binary FP Numbers

• 20.45 decimal = ? Binary

• 20 decimal = 10100 binary

• 0.45 x 2 = 0.9 (not greater than 1, first bit after binary point is 0)
  0.90 x 2 = 1.8 (greater than 1, second bit is 1, subtract 1 from 1.8)
  0.80 x 2 = 1.6 (greater than 1, third bit is 1, subtract 1 from 1.6)
  0.60 x 2 = 1.2 (greater than 1, fourth bit is 1, subtract 1 from 1.2)
  0.20 x 2 = 0.4 (less than 1, fifth bit is 0)
  0.40 x 2 = 0.8 (less than 1, sixth bit is 0)
  0.80 x 2 = 1.6 (greater than 1, seventh bit is 1, subtract 1 from 1.6)
  … and the pattern repeats

  10100.011100110011001100…
  Normalized form = 1.0100011100110011… x 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...000

Double: \((1 + 11 + 52)\)
1 0111 1111 110 1000...000

• What decimal number is represented by the following single-precision number?
1 1000 0001 01000...0000
-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
Boolean Algebra

- \( A + B = A \cdot B \)

- \( A \cdot B = A + B \)

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 64th 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
32-bit ALU

Source: H&P textbook
Control Lines

What are the values of the control lines and what operations do they correspond to?

<table>
<thead>
<tr>
<th>Ai</th>
<th>Bn</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sub</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>SLT</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>NOR</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Source: H&P textbook
Problem description: A traffic light with only green and red; either the North-South road has green or the East-West road has green (both can’t be red); there are detectors on the roads to indicate if a car is on the road; the lights are updated every 30 seconds; a light must change only if a car is waiting on the other road.

State Transition Table:

<table>
<thead>
<tr>
<th>CurrState</th>
<th>InputEW</th>
<th>InputNS</th>
<th>NextState=Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>N</td>
<td>0</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>0</td>
<td>E</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>1</td>
<td>N</td>
</tr>
</tbody>
</table>
Title

- Bullet