CIS 314

Midterm Review

Shreeya Khadka, Brian Gravelle
& Prof. Michel A. Kinsky
Bit & Byte

• A bit is the smallest unit of information
  ‣ It represents one 2-way decision or a 2-way choice
  ‣ yes / no, true / false, on / off ...
  ‣ Abstraction of all of these is represented as 0 or 1
    • A single digit with one of two values
    • binary digit → bit

• All information in a computer is stored and processed as bits
  ‣ High voltage/low voltage, current flowing/not flowing

• A byte is 8 bits that are treated as a unit
Binary Number System

- Binary numbers are short-hands for sums of powers of 2
  - $11011 = 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$
  - $= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
- Most of us count in "base 10", using powers of 10
- Computer counts in "base 2", using power of 2
## Binary arithmetic

- It works just like decimal arithmetic

### Addition

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 + 0</td>
<td>0</td>
</tr>
<tr>
<td>0 + 1</td>
<td>1</td>
</tr>
<tr>
<td>1 + 0</td>
<td>1</td>
</tr>
<tr>
<td>1 + 1</td>
<td>10</td>
</tr>
</tbody>
</table>

### Subtraction

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 0</td>
<td>0</td>
</tr>
<tr>
<td>0 - 1</td>
<td>-1</td>
</tr>
<tr>
<td>1 - 0</td>
<td>1</td>
</tr>
<tr>
<td>1 - 1</td>
<td>0</td>
</tr>
</tbody>
</table>

Small Problem

We will see how to solve it in next lecture
Converting binary to decimal

- Converting to decimal, so we can use polynomial evaluation
  \[
  10110101_2 = 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0
  \]
  
  \[
  = 128 + 32 + 16 + 4 + 1 = 181_{10}
  \]
Converting decimal to binary

1. Divide by target base (2 in this case)
2. Remainders become digits in the new representation
3. Digits produced in right to left order
4. Quotient is used as next dividend
5. Stop when the quotient becomes zero, but use the corresponding remainder
Converting decimal to binary

97 ÷ 2 \(\Rightarrow\) quotient = 48, remainder = 1 (LSB)

48 ÷ 2 \(\Rightarrow\) quotient = 24, remainder = 0

24 ÷ 2 \(\Rightarrow\) quotient = 12, remainder = 0

12 ÷ 2 \(\Rightarrow\) quotient = 6, remainder = 0

6 ÷ 2 \(\Rightarrow\) quotient = 3, remainder = 0

3 ÷ 2 \(\Rightarrow\) quotient = 1, remainder = 1

1 ÷ 2 \(\Rightarrow\) quotient = 0 (Stop), remainder = 1 (MSB)

Answer = 1 1 0 0 0 0 1₂
Hexadecimal notation

- Binary number can be long and hard to read, so hexadecimal numbers were introduced.
- It combines 4 bits into a single digit, written in base 16.
- A more compact and more readable representation of the same information.
- Hexadecimal number system uses the symbols A, B, C, D, E, F for the digits 10, 11, 12, 13, 14, and 15, respectively.
# Numbers and Bases

<table>
<thead>
<tr>
<th>Decimal Base-10</th>
<th>Binary Base-2</th>
<th>Hexadecimal Base-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>00010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>00011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>00100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>00101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>00110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>00111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>01000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>01001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>01010</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>01011</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>01100</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>01101</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>01110</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>01111</td>
<td>F</td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
<td>10</td>
</tr>
<tr>
<td>17</td>
<td>10001</td>
<td>11</td>
</tr>
</tbody>
</table>
Signed Integer Representation

• There are three ways in which signed binary numbers may be expressed:
  ‣ Signed magnitude
  ‣ One’s complement
  ‣ Two’s complement

• In an 8-bit binary number, signed magnitude representation places the absolute value of the number in the 7 bits to the right of the sign bit
  ‣ 4  = 0 0000100
  ‣ -4  = 1 0000100
One’s complement

- It amounts to little more than flipping the bits of a binary number.

- In an 8-bit binary number using One’s complement:
  - $4 = 0000100$
  - $-4 = 1111011$

- In one’s complement, as with signed magnitude, negative values are indicated by a 1 in the high order bit.
  - Complement systems are useful because they eliminate the need for subtraction.
Two’s complement

• To express a value in two’s complement:

• If the number is positive, convert it to binary and it is done

• If the number is negative, find the one’s complement of the number and then add 1

  ‧ 4  = 0 0000100

  ‧ -4  = 1 1111100
Two’s complement

- With Two’s complement arithmetic
- Just add the two binary numbers
- Discard any carries emitting from the high order bit
- Sum of 48 and –19
  - 19 in binary is 00010011
  - -19 in one’s complement is 11101100
  - -19 in two’s complement is 11101101
  - Answer is 29
Floating-Point Representation

• Computer systems use a form of scientific notation for floating-point representation

• Numbers written in scientific notation have three components:
  - Sign
  - Mantissa
  - Exponent

• Computer representation of a floating-point number consists of three fixed-size fields:

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1.5</td>
<td>x 10(^{-1})</td>
</tr>
</tbody>
</table>
Floating-Point Representation

- The one-bit sign field is the sign of the stored value
- The size of the exponent field, determines the range of values that can be represented
- The size of the significand determines the precision of the representation
Floating-Point Representation

- The IEEE-754 single precision floating point standard uses an 8-bit exponent and a 23-bit significand.
- The IEEE-754 double precision standard uses an 11-bit exponent and a 52-bit significand.
Floating-Point Representation

- The significand of a floating-point number is always preceded by an implied binary point.
- The significand always contains a fractional binary value.
- The exponent indicates the power of 2 to which the significand is raised.
Floating-Point Representation

• Express $32_{10}$ in the simplified 14-bit floating-point representation

  $32 = 2^5 = 1.0 \times 2^5 = +0.1 \times 2^6$

• Exponent field = $110_2 = 6_{10}$

• Significand field 1

  $+00110\ 10000000$

• How do we express fractional numbers $0.5 = 2^{-1}$?
Floating-Point Representation

- To provide for negative exponents, a biased exponent is used.

- The IEEE-754 single precision floating point standard uses a bias of 127 over its 8-bit exponent:
  - An exponent of 255 indicates:
    - Infinity if significand is zero
    - NaN, “not a number,” often used to flag an error condition if the significand is nonzero

- The double precision standard has a bias of 1023 over its 11-bit exponent.
Floating-Point Representation

- Sum of $12_{10}$ and $1.25_{10}$ using the 14-bit floating-point representation
  - $12_{10} = 0.1100 \times 2^4$
  - $1.25_{10} = 0.101 \times 2^1 = 0.000101 \times 2^4$
  - Thus, the sum is $0.110101 \times 2^4$
Program Memory Allocation

• There are two types of memory allocation
  ‣ Static memory allocation: Memory is allocated at the start of the program, and freed when program exits
    • Done by the compiler automatically (implicitly)
    • Global variables or objects
      ‣ Alive throughout program execution
      ‣ Can be access anywhere in the program
    • Local variables (inside a function)
      ‣ Memory is allocated when the function starts and freed when the routine returns
      ‣ A local variable cannot be accessed from another function
There are two types of memory allocation:

- Dynamic memory allocation deals with objects whose size can be adjusted depending on needs.
  - Dynamic – Done explicitly by programmer.
  - Programmer explicitly requests the system to allocate memory and return the starting address of the allocated memory.
  - This address can be used by the programmer to access the allocated memory.
  - When done using memory, it must be explicitly freed.
Application Compiling Process

- C Language

- Human Readable
  - C program
    - compiler
    - assembly code
    - assembler
    - object code
    - linker
    - library routines
    - executable
    - loader
    - memory

- Machine Code
if ($t5 < 0) then
{
    $s0 = 0 - $t5;
    $t1 = $t1 +1;
}
else
{
    $s0 = $t5;
    $t2 = $t2 + 1;
}

bgez $t5, else
# if ($t5 is ≥ zero) branch to else
sub $s0, $zero, $t5
# $s0 gets the negative of $t5
addi $t1, $t1, 1
# increment $t1 by 1
b next
# branch around the else code
else:
    ori $s0, $t5, 0
    # $s0 gets a copy of $t5
    addi $t2, $t2, 1
    # increment $t2 by 1
next:
\textbf{Assembly Through MIPS}

\begin{minipage}{0.5\textwidth}
\begin{verbatim}
$v0 = 1
\textbf{While} ($a1 < $a2) \textbf{do}
\{
\quad $t1 = \text{mem}[\textcolor{red}{$a1}]$
\quad $t2 = \text{mem}[\textcolor{red}{$a2}]$
\quad \textbf{If} ($t1 \neq t2$) \textbf{go to break;}
\quad $a1 = a1 + 1$
\quad $a2 = a2 - 1$
\}
\textbf{break:} \quad $v0 = 0$
\end{verbatim}
\end{minipage}
\begin{minipage}{0.5\textwidth}
\begin{verbatim}
li \quad $v0, 1 \quad \# \text{Load } v0 \text{ with the value 1}$
\textbf{loop:}
\quad bgeu \quad $a1, a2, done$
\quad \# \text{If } (a1 \geq a2) \text{ Branch to done}
\quad lb \quad $t1, 0(a1)$
\quad \# \text{Load a Byte: } t1 = \text{mem}[a1 + 0]
\quad lb \quad $t2, 0(a2)$
\quad \# \text{Load a Byte: } t2 = \text{mem}[a2 + 0]
\quad bne \quad $t1, t2, break$
\quad \# \text{If } (t1 \neq t2) \text{ Branch to break}
\quad addi \quad $a1, a1, 1 \quad \# a1 = a1 + 1$
\quad addi \quad $a2, a2, -1 \quad \# a2 = a2 - 1$
\quad b \quad \text{loop} \quad \# \text{Branch to loop}
\textbf{break:}
\quad li \quad $v0, 0$
\quad \# \text{Load } v0 \text{ with the value 0}
\end{verbatim}
\end{minipage}
$a0 = 0;
For ( $t0 =10;
       $t0 > 0;
       $t0 = $t0 -1)
do { $a0 = $a0 + $t0}

li $a0, 0      # $a0 = 0
li $t0, 10
# Initialize loop counter to 10
loop:
add $a0, $a0, $t0
addi $t0, $t0, -1
# Decrement loop counter
bgtz $t0, loop
# If ($t0 >0) Branch to loop
RISC: MIPS

• Case study

  ‣ Assume that A is an array of 64 words and the compiler has associated registers $s1$ and $s2$ with the variables x and y. Also assume that the starting address, or base address is contained in register $s3$. Determine the MIPS instructions associated with the following C statement:

  • $x = y + A[4]$; // adds 4th element in array A to y and stores result in x
RISC: MIPS

• Case study

  › Assume that A is an array of 64 words and the compiler has associated registers $s1$ and $s2$ with the variables $x$ and $y$. Also assume that the starting address, or base address is contained in register $s3$.

  • $x = y + A[4]$; // adds 4th element in array A to y and stores result in x

  › Solution:

  • $lw \$t0, 16(\$s3)$ # $s3$ contains the base address of array and # 16 is the offset address of the 4th element

  • $add \$s1, \$s2, \$t0$ # performs addition
Instruction Set Architecture (ISA)

• Instructions are the language the computer understand
• Instruction Set is the vocabulary of that language
• It serves as the hardware/software interface
  ‣ Defines data types
    • byte, int, float, double, string, vector…
  ‣ Defines set of programmer visible state
    • Known as the programmer’s model of the machine
  ‣ Defines instruction semantics (operations, sequencing)
    • operand location: register, immediate, indirect, . . .
    • add, sub, mul, move, compare, …
Instruction Set Architecture (ISA)

• Instructions are the language the computer understand

• Instruction Set is the vocabulary of that language

• It serves as the hardware/software interface
  ‣ Defines instruction format (bit encoding)
    • Number of explicit operands per instruction
    • Operand location
    • Number of bits per instruction
    • Instruction length: fixed, short, long, or variable., …
  ‣ Examples: MIPS, Alpha, x86, IBM 360, VAX, ARM, JVM
• Instructions can be divided into 3 classes

1. Data movement instructions

2. Arithmetic and logic (ALU) instructions

3. Branch instructions (control flow instructions)
   • Alter the normal flow of control from executing the next instruction in sequence
   • `bgez $t8, else`
ISA and Performance

- Instructions per program depends on source code, compiler technology and ISA
- Cycles per instructions (CPI) depends upon the ISA and the microarchitecture
- Time per cycle depends upon the microarchitecture and the base technology

\[
\text{Time Program} = \frac{\text{Instructions Program}}{\text{Cycles Instruction}} \times \text{Time Cycle}
\]
Complex Instruction Set Computer

• Large number of instructions (~200-300 instructions)
• Specialized complex instructions
• Many different addressing modes
  ‣ Including specialized modes for indexing through arrays
  ‣ 12 addressing modes available in x86
  • Immediate, Register operand, Displacement, Base, Base with displacement, Scaled index with displacement, Base with index and displacement, Base scaled index with displacement and Relative
Reduced Instruction Set Computer

- Relatively few number of instructions (~50)
- Basic instructions
- Relatively few different addressing modes
- Fixed length instruction format
- Only load/store instructions can access memory
- Large number of registers
- Hardwired rather than micro-program control
- Example: MIPS, ARM
Reduced Instruction Set Computer

• Simpler to design
• Higher Performance
  ‣ Smaller die size
• Lower power consumption
• Easier to develop compilers to take advantage of all features
  ‣ Simple code generation
  ‣ Regularity in CPI
MIPS Instructions

- There are 3 types of instruction in MIPS:

  1. **R-Type**

     | 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
     |----|----|----|----|----|----|----|----|----|----|----|----|
     | op | rs | rt | rd | shamt | funct |
     | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

  2. **I-Type**

     | 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
     |----|----|----|----|----|----|----|----|
     | op | rs | rt | Immediate |

     | 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
     |----|----|----|----|----|----|----|----|
     | op | rs | rt/funct | Displacement |

  3. **J-Type**

     | 31 | 26 | 25 | 0 |
     |----|----|----|----|
     | op | target |
Types of Digital Circuits

- Combinatorial logic
  - A combinational circuit consists of logic gates whose outputs, at any time, are determined by combining the values of the inputs

- Sequential logic
  - Output depends not only on the present value of its input signals but on the sequence of past inputs
Clock

- A synchronous system is synchronized according to a clock.
- A clock cycle or cycle time or clock period is the duration between two consecutive rising or falling edges.

4 GHz = clock speed = $\frac{1}{cycle\ time} = \frac{1}{250\ ps}$
Recall our second view of computer organization.
Arithmetic Logic Unit (ALU)

- Operation Examples

<table>
<thead>
<tr>
<th>ALU control lines</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
</tr>
<tr>
<td>0001</td>
<td>OR</td>
</tr>
<tr>
<td>0010</td>
<td>ADD</td>
</tr>
<tr>
<td>0110</td>
<td>SUB</td>
</tr>
<tr>
<td>0111</td>
<td>SLT</td>
</tr>
<tr>
<td>1100</td>
<td>NOR</td>
</tr>
</tbody>
</table>
Central Processing Unit (CPU) Organization

- CPU = Control Unit + ALU + Registers
- Control Unit: monitors and directs sequences of instructions
- ALU (Arithmetic-Logic Unit): performs arithmetic and logical operations
Central Processing Unit (CPU)

- Central Processing Unit (CPU) Organization
- CPU Execution Process
  1. Fetch Instruction
  2. Decode Instruction
  3. Execute Operation
  4. Memory Operation
  5. Register Writeback Operation
Next Class

- Midterm