CS 24: Introduction to Computing Systems

Spring 2017
Lecture 2
**Last Time**

- Began exploring the concepts behind a simple programmable computer
- Construct the computer using Boolean values (a.k.a. “bits”) and logic gates to process them
- Represent unsigned and signed integers as vectors of bits
  
  $B2U_w(x) = \sum_{i=0}^{w-1} x_i 2^i$
  
  $B2T_w(x) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$

- Briefly explored how to construct more complex computations using gates
  - e.g. unsigned and signed arithmetic
FUNCTIONAL COMPONENTS OF A SIMPLE PROCESSOR

- Can use our logic gates to construct various components to use in a processor
  - Already saw how to implement addition with logic
- Minimal components for a simple processor:
  - Signal Buses
    - Ability to route signals within our processor
  - Arithmetic/Logic Unit (ALU)
    - Performs various arithmetic and logical operations on data inputs, based on control inputs
- Memory
  - Addressable locations to store and retrieve values
**Buses**

- A **bus** is a set of wires that transfer signals from one component to another
  - Transmits values of a fixed bit-width, e.g. 64 bits
- Common uses for buses in a computer:
  - Transfer data between CPU and memory
  - Transfer data between CPU and peripherals
- Buses often drawn as a single line with a slash across it
- Individual signals drawn as a line with no slash
**Routing Buses**

- Multiplexers and demultiplexers (decoders) are used to route buses between multiple components.
- **Example: a 4-input multiplexer (MUX)**
  - Has two address inputs.
  - Address selects one of 4 data inputs.
  - Corresponding data input is fed to the data output.
- **A 4-input demultiplexer (DEMUX)**
  - Again, two address inputs.
  - Address selects one of 4 data outputs.
  - Single data input fed to corresponding data output.
ARITHMETIC/LOGIC UNIT

- A component that can perform various arithmetic and logic functions
- Symbol:

- Given two \( w \)-bit inputs and a set of control inputs
  - Control inputs specify the operation to perform
- Produces a \( w \)-bit result, and status outputs
  - Example status outputs:
    - sign flag (topmost bit of R)
    - carry-out flag (unsigned overflow)
    - zero flag (is R == 0?)
    - overflow flag (signed overflow)
EXAMPLE ALU OPERATIONS

- Control signals specify what operation to perform
- Example: for our contrived ALU

<table>
<thead>
<tr>
<th>Control</th>
<th>Operation</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>ADD</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0011</td>
<td>SUB</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0100</td>
<td>NEG</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>AND</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1001</td>
<td>OR</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1010</td>
<td>XOR</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1011</td>
<td>INV</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>SHL</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>SHR</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

- By feeding appropriate control and data signals to ALU in sequence, can perform computations
- Some operations require only one argument
  - Second argument ignored
MEMORY

- Need a component to store both instructions and data to feed to the ALU

Memory:
  - An array of linearly addressable locations
  - Each location has its own address
  - Each location can hold a single $w$-bit value

Inputs and outputs:
  - Address of location to read or write
  - Read/Write control signal
  - Data-input bus
  - Data-output bus
ASIDE: CPU COMPONENTS AND GATES

- It is a big claim that we can construct all of these components entirely from logic gates...
- Unfortunately, beyond the scope of CS24 to explore all the ways such components can be constructed, different approaches, etc.
- If you are curious how these things work, see the primer on the CS24 Moodle
  - Shows some basic ways these components can be constructed
- Don’t need to know this material in depth!
  - For CS24, really only need to understand the basics of how to implement logic equations with gates
ASSEMBLING THE COMPUTER

- Hook these components together, like this:

- Simplifications in our computer:
  - Two memory banks; identical copies of each other
  - Don’t care about ALU status outputs
INSTRUCTING THE COMPUTER

- This set of inputs forms an instruction
- Consists of:
  - The operation the ALU should perform
  - Addresses of two input values
  - Whether result should be stored
  - If so, what address to store the result at
- To program our computer:
  - Devise a sequence of instructions to implement our desired computation
INSTRUCTING THE COMPUTER (2)

- Need a way to feed instructions to our computer

- Add an instruction memory to our system
- Also, add a program counter to track current instruction
  - Configured to auto-increment through the instruction memory
WRITING A PROGRAM

- Instructions for the processor are very limited
  - Can only compute one value, from one or two values
- Usually can’t implement a program in only one instruction
- Instead:
  - String together a sequence of instructions to implement the computation
  - Instructions will communicate via memory locations
- Computation we will implement:
  - \( C = (A - 2B) \land 00001111_2 \)
  - Given inputs A and B
  - Multiply B by 2, subtract result from A, then bitwise-AND with a mask
IMPLEMENTING OUR COMPUTATION (1)

- Computation: $C = (A - 2B) \& 00001111_2$

- Step 1: Assign locations for inputs and outputs
  - Inputs:
    - A and B (obvious)
    - Also, our mask: $00001111_2$
    - Program needs to include our constants, too
  - Output:
    - C

- Givens:
  - Our memory has 8 locations
  - Memory addresses are 3 bits wide
  - Data values are 8 bits wide ($w = 8$)
ASSIGNING DATA LOCATIONS

Locations for our initial and final data values:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>00001111₂</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>C</td>
</tr>
</tbody>
</table>
IMPLEMENTING OUR COMPUTATION (2)

- Step 2: decompose our program into instructions the processor can actually handle

- Program:
  - \( C = (A - 2B) \& 00001111_2 \)

- Need to know processor’s operations for this step.

- Steps:
  - Perform 2B first, as \( B + B \)
  - Then, subtract previous result from \( A \)
  - Finally, bitwise-AND this with mask to produce \( C \)

<table>
<thead>
<tr>
<th>Control</th>
<th>Operation</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>ADD</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0011</td>
<td>SUB</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0100</td>
<td>NEG</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>AND</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1001</td>
<td>OR</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1010</td>
<td>XOR</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1011</td>
<td>INV</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>SHL</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>SHR</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>
IMPLEMENTING OUR COMPUTATION (3)

- Step 3: need to assign locations to these intermediate values!

- Result of \( B + B = \) location 2
- Result of \( A - 2B = \) location 3
- Result of bitwise-AND stored in location 7
  - This is our result

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>2B</td>
</tr>
<tr>
<td>3</td>
<td>( A - 2B )</td>
</tr>
<tr>
<td>4</td>
<td>00001111&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>C</td>
</tr>
</tbody>
</table>
IMPLEMENTING OUR COMPUTATION (4)

- Step 4: Translate our program into instructions!
- Need to know form of instructions:
  - Operation Rd1Addr Rd2Addr Wr WrAddr
- Also need our memory layout and operation codes

<table>
<thead>
<tr>
<th>Control</th>
<th>Operation</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>ADD</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>0011</td>
<td>SUB</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>0100</td>
<td>NEG</td>
<td>2</td>
<td>2B</td>
</tr>
<tr>
<td>1000</td>
<td>AND</td>
<td>3</td>
<td>A – 2B</td>
</tr>
<tr>
<td>1001</td>
<td>OR</td>
<td>4</td>
<td>00001111₂</td>
</tr>
<tr>
<td>1010</td>
<td>XOR</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>INV</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>SHL</td>
<td>7</td>
<td>C</td>
</tr>
<tr>
<td>1110</td>
<td>SHR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementing our Computation (5)

<table>
<thead>
<tr>
<th>Control</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>ADD A B</td>
</tr>
<tr>
<td>0011</td>
<td>SUB A B</td>
</tr>
<tr>
<td>0100</td>
<td>NEG A</td>
</tr>
<tr>
<td>1000</td>
<td>AND A B</td>
</tr>
<tr>
<td>1001</td>
<td>OR A B</td>
</tr>
<tr>
<td>1010</td>
<td>XOR A B</td>
</tr>
<tr>
<td>1011</td>
<td>INV A</td>
</tr>
<tr>
<td>1100</td>
<td>SHL A</td>
</tr>
<tr>
<td>1110</td>
<td>SHR A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>2B</td>
</tr>
<tr>
<td>3</td>
<td>A – 2B</td>
</tr>
<tr>
<td>4</td>
<td>00001111₂</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>C</td>
</tr>
</tbody>
</table>

- **Operation** Rd1Addr Rd2Addr Wr WrAddr
- **Writing our program:**
  - Slot 2 = 2B 000: 0001 001 001 1 010
  - Slot 3 = A – 2B 001: 0011 000 010 1 011
  - Slot 7 = (...) & mask 010: 1000 011 100 1 111
RUNNING OUR PROGRAM

- To run our program:
  - Load instructions into instruction memory
  - Load initial data into data memory
  - Start program counter at 0

- Each instruction executes in sequence, updating memory locations
  - Uses results of previous instructions

- State of our computer:
  - Instruction memory
  - Data memory
  - Program counter
RUNNING OUR PROGRAM: INITIAL STATE

- Instruction memory:
  000: 0001 001 001 1 010
  001: 0011 000 010 1 011
  010: 1000 011 100 1 111

- Data memory:
  0: A
  1: B
  2: ???
  3: ???
  4: 00001111_2
  5: ???
  6: ???
  7: ???

Program Counter: 000
**Step 1: Slot 2 = B + B**

- **Instruction memory:**
  
  000: 0001 001 001 1 010
  001: 0011 000 010 1 011
  010: 1000 011 100 1 111

- **Data memory:**
  
  0: A
  1: B
  2: \(2B = B + B\)
  3: ???
  4: 00001111₂
  5: ???
  6: ???
  7: ???

**Program Counter:** 000
**Step 1: Update Program Counter**

- **Instruction memory:**
  - 000: 0001 001 001 1 010
  - 001: 0011 000 010 1 011
  - 010: 1000 011 100 1 111

- **Data memory:**
  - 0: A
  - 1: B
  - 2: 2B
  - 3: ???
  - 4: 00001111₂
  - 5: ???
  - 6: ???
  - 7: ???

Program Counter: 001
**Step 2: Subtract 2B from A**

- **Instruction memory:**
  
  000: 0001 001 001 1 010  
  001: 0011 000 010 1 011  
  010: 1000 011 100 1 111

- **Data memory:**
  
  0: A  
  1: B  
  2: 2B  
  3: A – 2B  
  4: 00001111₂  
  5: ???  
  6: ???  
  7: ???

**Program Counter:** 001
**Step 2: Update Program Counter**

- Instruction memory:
  - 000: 0001 001 001 1 010
  - 001: 0011 000 010 1 011
  - 010: 1000 011 100 1 111

- Data memory:
  - 0: A
  - 1: B
  - 2: 2B
  - 3: A – 2B
  - 4: 00001111₂
  - 5: ???
  - 6: ???
  - 7: ???

Program Counter: 010
**Step 3:** \( C = (A - 2B) \& 00001111_2 \)

- **Instruction memory:**
  000: 0001 001 001 1 010
  001: 0011 000 010 1 011
  010: 1000 011 100 1 111

- **Data memory:**
  0: A
  1: B
  2: 2B
  3: A – 2B
  4: 00001111_2
  5: ???
  6: ???
  7: \((A - 2B) \& 00001111_2\)

**Program Counter:** 010
RUNNING OUR PROGRAM: FINAL RESULT

- Instruction memory:
  000: 0001 001 001 1 010
  001: 0011 000 010 1 011
  010: 1000 011 100 1 111

- Data memory:
  0: A
  1: B
  2: 2B
  3: A – 2B
  4: 00001111₂
  5: ???
  6: ???
  7: (A – 2B) & 00001111₂
A Programmable Computer!

- Using our basic functional components, we are able to build a simple programmable computer!
- Implemented a computation using our processor’s instruction set:
  1. Assigned memory locations to inputs and output
  2. Decomposed computation into processor instructions
  3. Assigned memory locations for intermediate values
  4.Encoded sequence of instructions for our program
- By feeding instructions to computer in sequence, we can perform our computation
  - Individual instructions communicate by reading and writing various memory locations
### Machine Code, Assembly Language

- **Our program:**
  
  000: 0001 001 001 1 010  
  001: 0011 000 010 1 011  
  010: 1000 011 100 1 111

- This is called **machine code**
  - The actual data values that comprise the program
  - Hard to read and write!

- **Humans normally use assembly language**
  - A more human-readable language that is translated into machine code using an assembler
    
    ADD   R1, R1, R2  # R2 = 2B  
    SUB   R0, R2, R3  # R3 = A − 2B  
    AND   R3, R4, R7  # C = R3 & 00001111

  - Allows human-readable names, operations, comments
Before going forward, need to review what C offers for logical and bitwise operations

C uses integers to represent Boolean values
- 0 = false; any nonzero value = true

Logical Boolean operators:
- Logical AND: \( a \&\& b \)
- Logical OR: \( a \|\| b \)
- Logical NOT: \( !a \)
- Result is 1 if true, 0 if false

\&\& and \|\| are short-circuit operators
- Evaluated left-to-right
- For \&\&, if LHS is false then RHS is not evaluated
- For \|\|, if LHS is true then RHS is not evaluated
C LOGICAL AND BITWISE OPERATIONS (2)

- C also has many bit-manipulation operations
- Given \( a = 00010100_2 \) (20\(_{10}\)), \( b = 00110010_2 \) (50\(_{10}\))
  - \( a \& b = 00010000 \) Bitwise AND
  - \( a | b = 00110110 \) Bitwise OR
  - \( \sim a = 11101011 \) Bitwise negation (invert)
  - \( a ^ b = 00100110 \) Bitwise XOR

- Note:
  - C has no way of specifying base-2 literals
  - Normal approach: use hexadecimal literals instead

- Hexadecimal: base-16 numbers
  - Digits are 0..9, A..F (or a..f, makes no difference)
  - A = 10, B = 11, ..., F = 15
HEXADECIMAL VALUES AND BIT-MASKS

Example: $0x0F$ is a hexadecimal literal in C
- Each digit of a hexadecimal value represents 4 bits – a compact, simple way to write bit-fields
  - $0x0F = 0000\ 1111$ (also $0x0f$)
  - $0x03C7 = 0000\ 0011\ 1100\ 0111$ (also $0x03c7$)

Use bitwise AND to mask out or clear specific bits
- $a \& \ 0x0F$
  - Clears high nibble of $a$; retains low nibble of $a$

Use bitwise OR to set specific bits
- $a = a \mid \ 0x28$
  - Sets bits 3 and 5 of value in $a$ ($0x28 = 0010\ 1000$)
  - Other bits in $a$ remain unchanged

Use bitwise XOR to toggle specific bits
- $a = a \ ^\ 0x28$
  - Toggles bits 3 and 5 of value in $a$; other bits are left unchanged
C BIT-SHIFTING OPERATIONS

- C also includes bit-shifting operations
- Shift bits in \( a \) left by \( n \) bits: \( a \ll n \)
  - New bits on right are 0
  - Shifting left by \( n \) bits is identical to multiplying by \( 2^n \)
    \[
    a = 42; \quad /* a = 00101010 = 42 */
    a = a \ll 1; \quad /* a = 01010100 = 84 */
    \]
- Shift bits in \( a \) right by \( n \) bits: \( a \gg n \)
  - Shifting right by \( n \) bits is identical to dividing by \( 2^n \)
- Question: What should new bits on left be?
  - Depends on whether \( a \) is signed or unsigned!
    \[
    a = -24; \quad /* Two's complement: 11101000 */
    a = a \gg 1; \quad /* Should be -12 (11110100) now */
    \]
  - Leftmost bit represents sign
  - Preserve sign by using same value as original sign-bit
ARITHMETIC VS. LOGICAL SHIFT-RIGHT

- Distinguish between arithmetic shift-right and logical (i.e. bitwise) shift-right
  - Arithmetic shift-right preserves the value’s sign
  - Logical shift-right always adds 0-bits to left of value
- Some languages make this distinction
  - Java: `>>` is arithmetic, `>>>` is logical
  - x86 assembly: `SAR` is arithmetic, `SHR` is logical
- In C:
  - If argument is signed, shift-right is arithmetic
    ```c
    char a = -24; /* -24 = 11101000 */
    printf("%d", a >> 1); /* Prints -12 = 11101000 */
    ```
  - If argument is unsigned, shift-right is logical
    ```c
    /* Prints 116 = 01110100; topmost bit is 0 */
    printf("%d", (unsigned char) a >> 1);
    ```
**Bit-Shifts and Masks**

- Can use bit-shifts with masks to extract sub-byte values

- \((a >> 4) & 0x0F\)
  - Retrieves high nibble of \(a\)

- Does it matter if \(a\) is signed or unsigned?
  - Nope. We chop off the sign bit after we shift.
MULTIPLICATION?

- Our processor’s instruction set:
- Hmm, no multiply instruction.
- No problem; implement multiply with addition and shifting

\[ \text{mul}_w(a, b) = \sum_{i=0}^{w-1} a_i b 2^i \]

```c
int mul(int a, int b) {
    int p = 0;
    while (a != 0) {
        if (a & 1 == 1)
            p = p + b;
        a = a >> 1;
        b = b << 1;
    }
    return p;
}
```

<table>
<thead>
<tr>
<th>Control</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>ADD A B</td>
</tr>
<tr>
<td>0011</td>
<td>SUB A B</td>
</tr>
<tr>
<td>0100</td>
<td>NEG A</td>
</tr>
<tr>
<td>1000</td>
<td>AND A B</td>
</tr>
<tr>
<td>1001</td>
<td>OR A B</td>
</tr>
<tr>
<td>1010</td>
<td>XOR A B</td>
</tr>
<tr>
<td>1011</td>
<td>INV A</td>
</tr>
<tr>
<td>1100</td>
<td>SHL A</td>
</tr>
<tr>
<td>1110</td>
<td>SHR A</td>
</tr>
</tbody>
</table>
MULTIPLICATION ???

- Our multiply program:
  ```
  int mul(int a, int b) {
    int res = 0;
    while (a != 0) {
      if (a & 1 == 1)
        p = p + b;
      a = a >> 1;
      b = b << 1;
    }
    return p;
  }
  ```

- Can we write a program to execute this code?
  - NO! 😞
  - Our processor doesn’t support any branching or jumping operations
BRANCHING SUPPORT

- Our current processor architecture can’t support branching or jumping!
  - Can only execute code in sequential order
- Need to extend the hardware to support branching and jumping
  - Need to be able to update the Program Counter field
- Note:
  - Not always essential to support branching and jumping!
  - Many dedicated graphics processors (GPUs) don’t support looping or branching at all
  - However, is essential for a general-purpose processor
SUMMARY

- We designed a simple programmable computer!
  - Assembled functional components so we can perform a variety of simple computations
  - Feed instructions into our processor in sequence, from instruction memory
  - Instructions communicate by reading and writing various memory locations

- But, our computer has substantial limitations...
  - Can’t even implement a simple loop yet. 😞
  - Need to extend our processor to support branching

- Also, our computer only has 8 bytes of memory
  - Need to examine impact of increasing memory size