CS 24: Introduction to Computing Systems

Spring 2015
Lecture 3
LAST TIME

- Basic components of processors:
  - Buses, multiplexers, demultiplexers
  - Arithmetic/Logic Unit (ALU)
  - Addressable memory

- Assembled components into a simple processor
  - Can perform a wide range of operations, but all very simple
  - Need to string together a sequence of instructions, which communicate via memory locations

- Implemented a computation on this processor:
  - Assigned memory locations for inputs, outputs, constants
  - Decomposed the computation into steps the processor could actually handle
  - Assigned locations to intermediate values
  - Translated the program into machine-code instructions
MULTIPLICATION...

Then, we wanted to implement multiplication:

```c
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;
}
```

But, we couldn’t write this program:
- Our processor doesn’t support branching operations!
Update our ISA and Processor

- Add a new instruction: BRZ A, Addr (Branch if Zero)
  - If value in slot A is 0, change Prog Ctr to address Addr

- New logic to support this instruction:

  **Branch Logic:**
  - If opcode is BRZ, and memory A outputs 0, then tell multiplexer to load Addr into the Program Counter.
**UPDATED PROCESSOR**

- With updated processor, can reuse instructions by creating loops in our programs
  - Perform many computations with just a few instructions

- Can now implement computations where number of steps is dependent on program’s inputs
BACK TO MULTIPLICATION

- Should have enough capability now to encode our program

  ```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;
  }
  ```

- Coding is more complex now!
  - Need to plan out our loops...
  - Need to know the *addresses* to jump to!

<table>
<thead>
<tr>
<th>Control</th>
<th>Operation</th>
<th>Control</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>ADD A B</td>
<td>0011</td>
<td>SUB A B</td>
</tr>
<tr>
<td>0100</td>
<td>NEG A</td>
<td>0111</td>
<td>BRZ A Addr</td>
</tr>
<tr>
<td>1000</td>
<td>AND A B</td>
<td>1001</td>
<td>OR A B</td>
</tr>
<tr>
<td>1010</td>
<td>XOR A B</td>
<td>1011</td>
<td>INV A</td>
</tr>
<tr>
<td>1100</td>
<td>SHL A</td>
<td>1110</td>
<td>SHR A</td>
</tr>
</tbody>
</table>
**Writing Our Program**

- Use same general process as before
- Step 1: identify inputs and outputs
  ```
  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;
  }
  ```
- Inputs: A, B, some constant(s)
- Outputs: P
Step 2:
- Decompose program into processor instructions.

This step will be much more involved now:
- May need quite a few temporary values
  - (Don’t know how many though...)
- Need to keep track of various addresses for branching
  - (Also don’t know how many...)

Strategy: use names for unknown values
- Put in placeholders for temp-variables, places to jump
- Write the code you do understand...
- Once code is written, replace names with addresses
Writing Our Program (3)

- Code:
  ```
  int p = 0;
  while (a != 0) {
      if (a & 1 == 1)
          p = p + b;
      a = a >> 1;
      b = b << 1;
  }
  
  First step: Set P = 0
  
  Options?
  - Subtract P from itself: `SUB P, P, P`
  - XOR P with itself: `XOR P, P, P`
  
  XOR option appears frequently in assembly code

<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>0111</td>
<td>BRZ A Addr</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>
WRITING OUR PROGRAM (4)

- **Code:**

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

- **Steps:**

```c
XOR P, P, P
WHILE:
    ...;
```

- Will certainly need to go back to start of loop
- **Add a label to the code**
  - Use the label in branching instructions
  - At end of translation, replace label with an actual address
- We already do this with our variables...
Writing Our Program (5)

- Code:
  ```c
  int p = 0;
  while (a != 0) {
      if (a & 1 == 1) 
          p = p + b;
      a = a >> 1;
      b = b << 1;
  }
  ```

- Steps:
  ```c
  XOR P, P, P
  WHILE:
  BRZ A, DONE
  ... 
  ```

- Don’t know what will go inside loop yet, but we know we will need to exit it when finished!

- Add a **DONE** label, and use **BRZ** instruction to test A.
WRITING OUR PROGRAM (6)

- Code:
  ```
  int p = 0;
  while (a != 0) {
    if (a & 1 == 1)
      p = p + b;
    a = a >> 1;
    b = b << 1;
  }
  ```
- Steps:
  ```
  XOR P, P, P
  WHILE:
    BRZ A, DONE
    AND A, 1, Tmp
    BRZ Tmp, SKIP
  ...
  SKIP:
    ...
  DONE:
  ```
- Compute A & 1, then store in a temporary location
- Then, use branching instruction to skip body of if statement
  (Can use BRZ since Tmp will be 0 or 1 here...)
Writing Our Program (7)

- **Code:**
  ```c
  int p = 0;
  while (a != 0) {
    if (a & 1 == 1)
      p = p + b;
    a = a >> 1;
    b = b << 1;
  }
  ```

- **Steps:**
  ```
  XOR P, P, P
  WHILE:
  BRZ A, DONE
  AND A, 1, Tmp
  BRZ Tmp, SKIP
  ADD P, B, P
  SKIP:
  SHR A, A
  SHL B, B
  BRZ 0, WHILE
  DONE:
  ```

- Remaining operations are mostly easy to write...
- Need an *unconditional* branching operation
  - Just use **BRZ** with a value of 0
CONTROL-FLOW IN OUR PROGRAM

- Code:
  ```c
  int p = 0;
  while (a != 0) {
    if (a & 1 == 1)
      p = p + b;
    a = a >> 1;
    b = b << 1;
  }
  ```

- Steps:
  ```
  XOR P, P, P
  WHILE:
  BRZ A, DONE
  AND A, 1, Tmp
  BRZ Tmp, SKIP
  ADD P, B, P
  SKIP:
  SHR A, A
  SHL B, B
  BRZ 0, WHILE
  DONE:
  ```

- Manually implemented our `while` loop:
  - Test condition at start; exit if result is false
  - At end of loop, unconditionally return to start
CONTROL-FLOW in Our Program (2)

- Code:
  ```c
  int p = 0;
  while (a != 0) {
    if (a & 1 == 1)
      p = p + b;
    a = a >> 1;
    b = b << 1;
  }
  ```

- Steps:
  ```
  XOR P, P, P
  WHILE:
  BRZ A, DONE
  AND A, 1, Tmp
  BRZ Tmp, SKIP
  ADD P, B, P
  SKIP:
  SHR A, A
  SHL B, B
  BRZ 0, WHILE
  DONE:
  ```

- Also manually implemented the **if** statement:
  - Tested inverse of the condition, and skip over if-body if result is false
FINISHING OUR PROGRAM

- Our assembly code so far:
  XOR P, P, P
  WHILE:
  BRZ A, DONE
  AND A, 1, Tmp
  BRZ Tmp, SKIP
  ADD P, B, P
  SKIP:
  SHR A, A
  SHL B, B
  BRZ 0, WHILE
  DONE:

- Assign locations to 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>Tmp</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>P</td>
</tr>
</tbody>
</table>

- Note: two constants to include as well...
Update our code with the memory addresses:

<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>Tmp</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>P</td>
</tr>
</tbody>
</table>

\[
\text{XOR 111, 111, 111} \\
\text{WHILE:} \\
\text{BRZ 000, DONE} \\
\text{AND 000, 011, 010} \\
\text{BRZ 010, SKIP} \\
\text{ADD 111, 001, 111} \\
\text{SKIP:} \\
\text{SHR 000, 000} \\
\text{SHL 001, 001} \\
\text{BRZ 100, WHILE} \\
\text{DONE:}
\]
**FINISHING OUR PROGRAM (3)**

- Now to figure out the instruction addresses

  0: XOR 111, 111, 111
  1: BRZ 000, DONE
  2: AND 000, 011, 010
  3: BRZ 010, SKIP
  4: ADD 111, 001, 111
  5: SHR 000, 000
  6: SHL 001, 001
  7: BRZ 100, WHILE
  8: DONE:

  **WHILE** = address 1
  **SKIP** = address 5
  **DONE** = address 8
# Finishing Our Program (3)

- Update code with the instruction addresses

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address 1</th>
<th>Address 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR</td>
<td>111, 111, 111</td>
<td></td>
</tr>
<tr>
<td>BRZ</td>
<td>000, 1000</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>000, 011, 010</td>
<td></td>
</tr>
<tr>
<td>BRZ</td>
<td>010, 0101</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>111, 001, 111</td>
<td></td>
</tr>
<tr>
<td>SHR</td>
<td>000, 000</td>
<td></td>
</tr>
<tr>
<td>SHR</td>
<td>001, 001</td>
<td></td>
</tr>
<tr>
<td>BRZ</td>
<td>100, 0001</td>
<td></td>
</tr>
</tbody>
</table>
Finally, encode each instruction into machine code

<table>
<thead>
<tr>
<th>Control</th>
<th>Operation</th>
<th>Control</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>ADD A B</td>
<td>0011</td>
<td>SUB A B</td>
</tr>
<tr>
<td>0100</td>
<td>NEG A</td>
<td>0111</td>
<td>BRZ A Addr</td>
</tr>
<tr>
<td>1000</td>
<td>AND A B</td>
<td>1001</td>
<td>OR A B</td>
</tr>
<tr>
<td>1010</td>
<td>XOR A B</td>
<td>1011</td>
<td>INV A</td>
</tr>
<tr>
<td>1100</td>
<td>SHL A</td>
<td>1110</td>
<td>SHR A</td>
</tr>
</tbody>
</table>

| XOR 111, 111, 111 | 1010 111 111 111 | 0000 |
| BRZ 000, 1000 | 0111 000 000 000 | 1000 |
| AND 000, 011, 010 | 1000 000 011 010 | 0000 |
| BRZ 010, 0101 | 0111 010 000 000 | 0101 |
| ADD 111, 001, 111 | 0001 111 001 111 | 0000 |
| SHR 000, 000 | 1110 000 000 000 | 0000 |
| SHL 001, 001 | 1100 001 000 001 | 0000 |
| BRZ 100, 0001 | 0111 100 000 000 | 0001 |
The Final Program

- Our final machine code: 1010 111 111 111 0000
  - Unused/ignored bits are grayed out
  - 136 bits; 44 bits unused
  - 30% of our program’s space is unused!

- This instruction encoding is not very efficient
- Many processors employ variable-length instruction encodings to reduce program size
  - Particularly common in CISC processors
  - Increases complexity in instruction processing logic
**Processor Memories**

- Our processor has a curious memory layout
  - Doesn’t seem much like our modern computers...

- Instructions are stored in a separate memory from data
- Uses two data memories that are mirror copies of each other
MEMORY ARCHITECTURES

- Several different memory architectures have been employed in computer systems
- Harvard Architecture:
  - Instruction memory and data memory are separate
    - May even have different word sizes from each other
  - Instruction memory is read-only
  - Data memory is read/write
  - *Cannot* treat instructions as data.
    - Constants must be explicitly loaded into data memory
- Named after Harvard Mark I computer
  - A relay-based computer with separate instruction and data memories
- Our example processor uses a Harvard architecture
MEMORY ARCHITECTURES (2)

- Von Neumann Architecture:
  - Instructions and data are stored in a single read/write memory

- A big benefit over Harvard architecture:
  - Much easier to load and manage programs on the computer

- Some big problems too:
  - Programs can easily corrupt their own code!
    - (Many exploits take advantage of this characteristic…)
  - CPU only has one bus to read both instructions and data
    - Harvard architecture has two buses; can use them in parallel

- Modern processors often blend these approaches
  - Overall system design uses von Neumann architecture
  - Internally, CPU has separate instruction, data memories
    - Used in very different ways, so separating them allows for much greater hardware optimization
INDIRECT MEMORY ACCESS

- Still plenty of problems our simple processor design can’t handle
- Example function:
  - Add two vectors together, component by component, storing the result into a third vector
  - Vectors to use are arguments to the function
- C code:
  ```c
  void vector_add(int *a, int *b, int *r, int length) {
    int i;
    for (i = 0; i < length; i++)
      r[i] = a[i] + b[i];
  }
  ```
  - This code is independent of where a, b, r are stored
    - Code can be reused on any set of three vectors
    - Addresses of the vectors are specified in the arguments
INDIRECT MEMORY ACCESS (2)

- Problem:
  - This processor can only specify literal address-values in the instructions

- Need to introduce a way to get memory addresses from variables instead...

- Need to update the hardware to support *indirect* memory access
LARGE MEMORIES

- But, before we talk about new functionality, this design has a problem.

- Our current memory only has 8 locations
  - Not very useful...

- Really need much larger memories than this!

- What would happen if we made our memories much larger?
  - e.g. 4GB, like modern 32-bit architectures
Large Memories and Instruction Size

- We directly encode memory addresses into our instructions.
  - Memory has 8 slots
  - \( \log_2(8) = 3 \) bits for memory addresses
- If memory is 4GB in size, we need 32 bits to specify addresses!

- Instruction size \( \approx \)
  - \( \log_2(\text{NumOps}) + 3 \times \log_2(\text{MemSize}) \)
  - \( \approx 100 \) bits for each instruction!
**Supporting Large Memories**

- A better design for large memories:
  - Small memory is called a *register file*
    - Individual locations are called *registers*
    - Small number of registers (e.g. 8, 64)
  - Instructions only use register addresses
    - Instructions stay small!
SUPPORTING LARGE MEMORIES (2)

- A better design for large memories:
  - ALU only operates on values stored in registers
  - To use main memory, must explicitly load or store values into registers
    - Need new instructions to work with main memory
**Loading Values from Main Memory**

To load values from main memory, introduce:

- **LD SRC, DST**
- SRC, DST are registers

**Meaning:**
- DST = Memory[SRC]

**LD** activates Load/Store Logic
- SRC value fed to RdAddr of main memory
- Output of main memory fed to DST register
- ALU is not utilized at all
STORING VALUES TO MAIN MEMORY

To store values to main memory, introduce:

- **ST SRC, DST**
- Again, SRC, DST are registers

**Meaning:**
- Memory[DST] = SRC

**ST** activates Load/Store Logic
- DST value fed to WrAddr of main memory
- SRC register fed to input of main memory
- Again, ALU not utilized
LOAD/STORE ARCHITECTURE

- This is called a Load/Store Architecture
  - ALU only operates on registers
  - Must explicitly load/store values to main memory
  - (Often seen in RISC ISAs)

- Benefits:
  - Very simple architecture
  - All instructions are same size
  - Instructions execute quickly!

- Drawbacks:
  - Memory-intensive programs require many operations to implement!
**Example: Vector-Add Function**

- Consider the body of our vector-add function:
  ```c
  int i;
  for (i = 0; i < length; i++)
    r[i] = a[i] + b[i];
  ```

- Arguments a, b, r are addresses of the *start* of their respective arrays
  - One implementation of `r[i] = a[i] + b[i]`:
    ```
    ADD A, I, ADDR   # Compute address of a[i]
    LD  ADDR, AI     #     and retrieve it.
    ADD B, I, ADDR   # Compute address of b[i]
    LD  ADDR, BI     #     and retrieve it.
    ADD AI, BI, RI   # Compute a[i] + b[i].
    ADD R, I, ADDR   # Compute address of r[i]
    ST  RI, ADDR     #     and store sum.
    ```

- *Hope the processor can make this fast, somehow...*
**Alternative to Load/Store**

- Instead of Load/Store architecture, we can also support a rich set of operand-types.

- For example, an operand could be:
  - A constant (i.e. an immediate value)
  - A register
    
    \[\text{ADD 14, R1, R2} \quad \# \quad R2 = 14 + R1\]
  - A direct address
    
    \[\text{ADD [150], 14, R7} \quad \# \quad R7 = \text{Memory[150]} + 14\]
  - An indirect address
    
    \[\text{ADD [R5 + 150], 14, R6} \quad \# \quad R6 = \text{Memory[R5+150]} + 14\]

- Instruction encodings may need to include:
  - Immediate value, Register index, Address, Address + Register index
  - Instruction encodings are variable-length, and must also include operand-mode bits to indicate types of operands.
Supporting Various Operand Types

- Instruction encodings can include:
  - Immediate value, Register index, Address, Address + Register index
- Clearly, we will need new logic to feed the ALU!
- Also, instructions become much more complicated
  - (CISC processors employ multiple operand types)
  - Requires complex, dedicated instruction-fetch and decode logic
- Finally, may need to do arithmetic on addresses fed to the main memory
**Operand Types**

- Example logic for supporting multiple operand types:
  - Operand A supports three operand types:
    - Constant
      - Encoded in CA value
    - Register
      - Encoded in A value
    - Memory[Const + Reg]
      - Uses both A and CA
  - New “operand mode” MA controls which value is fed into the ALU
**Operand Types (2)**

- Example logic for supporting multiple operand types:

- Similar logic would also be added to other ALU input

- Many other details too
  - Instruction decoding
  - Routing ALU output to registers/memory
  - etc.
EXAMPLE OPERAND TYPES

- Constant operand:
  - CA specifies constant value
  - MA feeds CA through to ALU

- Register A and memory are both unused
Example Operand Types (2)

- Register operand:
  - A specifies register address
  - Register file outputs the specified register value
  - MA feeds register to ALU

- Constant value and memory are both unused
**Example Operand Types (3)**

- Indirect memory access operand:
  - Constant and register value added to produce address for main memory
  - MA feeds memory output to ALU
SUMMARY

- Added branching to our processor
  - Can implement larger computations with fewer instructions
    - Reuse instructions on different data by looping
    - Tailor computations based on the specific data values
  - Allows us to perform computations where the work performed is proportional to the input values
- Explored various memory architecture details
  - Harvard vs. Von Neumann architectures
    - Separating instruction and data processing paths
  - Load/Store architecture vs. multiple operand types
    - Allows us to implement computations that are independent of the specific memory location
    - Allows us to access much larger memories, using a relatively small instruction set
Next Time

- Start looking at the Intel IA32 instruction set
- Start writing programs you can run on *real* computers! 😊
- Begin to develop abstractions to facilitate construction of larger programs
  - Subroutines, stacks, recursion