Unit 17

Instruction Sets
Picoblaze Processor

INSTRUCTION SET OVERVIEW

Instruction Set Review

- Defines the software interface of the processor and memory system
- Instruction set is the vocabulary the HW processor can understand and the SW is composed with
  - Usually the compiler is the one that translates the software
- Most assembly/machine instructions fall into one of three categories
  - Arithmetic/Logic
  - Data Transfer (to and from memory)
  - Control (branch, subroutine call, etc.)
- 2 historic approaches to instruction set design:
  - RISC and CISC

Historical Instruction Format Options

- Instruction sets limit the number of operands used in an instruction due to...
  - To limit the complexity of the __________
  - So that when an instruction is coded to binary it can _____ in a certain # of bits
- Different instruction sets specify these differently
  - 3 operand instruction set (ARM, PPC) -> (32-bit processors)
    - Usually all 3 operands in registers
    - Format: ADD DST, SRC1, SRC2 (DST = SRC1 + SRC2)
  - 2 operand instructions (Intel / Motorola 68K)
    - Second operand doubles as source and destination
    - Format: ADD SRC1, S2/D (S2/D = SRC1 + S2/D)
  - 1 operand instructions (Low-End Embedded, Java Virtual Machine)
    - Implicit operand to every instruction usually known as the Accumulator (or ACC) register
    - Format: ADD SRC1 (ACC = ACC + SRC1)
  - 0 operand instructions / stack architecture
    - Push operands on a stack: PUSH X, PUSH Y
    - ALU operation: ADD (Implicitly adds top two items on stack: X + Y & replaces them with the sum)
### General Instruction Format Issues

- Consider the high-level code
  - \( F = X + Y - Z \)
  - \( G = A + B \)
- Simple embedded computers often use single operand format
  - Smaller data size (8-bit or 16-bit machines) means limited instruction size
- Modern, high performance processors (Intel, ARM) use 2- and 3-operand formats

<table>
<thead>
<tr>
<th>Three-Operand</th>
<th>Two-Operand</th>
<th>Single-Operand</th>
<th>Stack Arch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD ( F, X, Y )</td>
<td>MOVE ( F, X )</td>
<td>LOAD ( X )</td>
<td>PUSH ( Z )</td>
</tr>
<tr>
<td>SUB ( F, Z )</td>
<td>ADD ( Y )</td>
<td>ADD ( Y )</td>
<td>PUSH ( Y )</td>
</tr>
<tr>
<td>ADD ( A, B )</td>
<td>STORE ( P )</td>
<td>ADD ( A )</td>
<td>POP ( F )</td>
</tr>
<tr>
<td>MOVE ( G, A )</td>
<td>LOAD ( A )</td>
<td>STORE ( G )</td>
<td></td>
</tr>
</tbody>
</table>

(+) More natural program style
(+) Smaller instruction count
(+) Smaller size to encode each instruction

### Operand Addressing

- Most modern processors use a ________ architecture
  - Load operands from memory into a register
  - Perform operations on registers and put results back into other registers
  - Store results back to memory
  - **Format**: \( \text{add} \quad r1, r2 \quad (rx = \text{reg. operand}) \)
  - Because ALU instructions only access registers, the CPU design can be simpler and thus faster
- Older designs
  - Register/Memory Architecture (Intel)
    - Operands of ALU instruc. can be in a reg. or mem.
    - **Format**: \( \text{add} \quad r1, m2 \quad (mx = \text{mem. operand}) \)
  - Memory/Memory Architecture (DEC VAX)
    - Operands of ALU instruc. Can be in memory
    - **Format**: \( \text{add} \quad m1, m2 \)

### Addressing Modes

- Addressing modes refers to how an instruction specifies **where** the operands are
  - Can be in a ________, ________ location, or a ________ that is part of the instruction itself (aka, ____________ value)
- Most RISC processors: All data operands for arithmetic instructions must be in a ____________
  - This allows the hardware to be simpler and faster
- But what about something like: \( r8 = r8 + \text{A}[i] \) (\( \text{A}[i] \) is in mem.)
  - Intel instructions would allow: ADD \( r8, A[i] \)
    - \( A[i] \) is read from memory AND added to \( r8 \) in a single instruction
  - Other processors requires all data to be in a register before performing an arithmetic or logic operation (aka Load/Store Architecture)
    - Must use a separate instruction to read data from memory into a register
    - LOAD \( r9, A[i] \)
    - ADD \( r8, r8, r9 \) (\( r8 = r8 + r9 \))

### Direct Memory Addressing

- When we load or store from/to memory we must specify the address of the desired memory location?
  - Note: Everything is a pointer at the instruction level
- Option 1: Direct Memory Addressing
  - Address must be a constant: LOAD \( r8, \ (0xa140) \)
    - 0xa140 is just a made up address where we will assume \( \text{A}[0] \) lives
  - _____________!
    - Would have to translate to:
      - CLR \( r1 \) \# \( x = r1 = 0 \)
      - LOAD \( r8, (0xa140) \)
      - ADD \( r1, r1, r8 \)
      - LOAD \( r9, (0xa144) \)
      - ADD \( r1, r1, r8 \)
      - ...

### Direct Memory Addressing Diagram

- Load/Store Architecture
  - 1.) Load operands to proc. registers
  - 2.) Proc. Performs operation using register values
  - 3.) Store results back to memory

- Direct Memory Addressing
  - i = 0;
    - while(\( i < \text{MAX} \)){
      - \( x = x + \text{A}[i++] \);
    }

- Load/Store Architecture Table:

<table>
<thead>
<tr>
<th>Proc.</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0] @ 0xa140</td>
<td>00</td>
</tr>
<tr>
<td>A[1] @ 0xa144</td>
<td>00</td>
</tr>
<tr>
<td>A[2] @ 0xa146</td>
<td>00</td>
</tr>
<tr>
<td>A[3] @ 0xa14C</td>
<td>00</td>
</tr>
</tbody>
</table>
**Indirect Memory Addressing**

- Option 2: Indirect Memory Addressing
  - Put address in a register: \( r9 = 0xa140 \)
  - LOAD uses contents of reg. as the address
  - Then we can increment the address to prepare for next iteration
  - MOVE \( r9, 0xa140 \)
  - **load:**
    - LOAD \( r8, (r9) \)
    - ADD \( r1, r1, r8 \)
    - ADD \( r9, __, __ \)
    - repeat

```plaintext
Pseudo Code:

```i = 0; while(i < MAX){
  x = x + A[i++];
}
```

**Picoblaze**

- Picoblaze (aka KCPSM6) is an 8-bit soft-processor
  - It is not "hard" in that there is \__________\ you can buy with just a Picoblaze processor
  - It is "soft" in that the processor design is given as \____________________\)
  - It is intended to be integrated with other hardware designs and used to execute software to control those other hardware designs
  - The whole system can then be implemented on a chip or FPGA

**Picoblaze Internals**

- \__________\ registers named \__________\)
  - Each register stores an 8-bit value
  - PC is 12-bits allowing it to handle programs of up to \__________\ instructions

---

**Picoblaze**

- **Picoblaze**
  - Custom HW I/O Device
  - 3rd Party IP I/O Device
  - ALU
  - Control
  - PC 01c

**Hardware/Software Interfacing**

**PICOBLAZE**
Normal Processor Bus Topology

- Most processors talk to memory and I/O devices over a common bus (aka Von Neumann architecture)

![Diagram of Normal Processor Bus Topology]

PicoBlaze Processor Bus Topology

- PicoBlaze has a separate:
  - __________ memory / bus (Harvard architecture)
  - __________ memory / bus
  - _______ bus

![Diagram of PicoBlaze Processor Bus Topology]

PICOBLAZE INSTRUCTION SET

- Adds a register value with a constant or two register values
  - `add sx, _______ // sx = sx + _____`
  - `add sx, _______ // sx = sx + _____`

SAMPLE ARITHMETIC/LOGIC INSTRUCTIONS

- Example: `add s3, 01`
  - Performs register `s3 = s3 + 1`

- Example: `add s3, sb`
  - Performs register `s3 = s3 + sb`

Derived from the KCPSM6 Manual
**SUB Instruction**

- Subtracts a register value with a constant or two register values
  - `sub sx, constant` \( \rightarrow \) `sx = sx - \text{const.}`
  - `sub sx, sy` \( \rightarrow \) `sx = sx - sy`

- Example: `sub s3, 01`
  - Performs register `s3 = s3 - 1`
  - Before: `18\ s3`
  - After: `17\ s3`

- Example: `sub s3, sb`
  - Performs register `s3 = s3 - sb`
  - Before: `15\ s3`
  - After: `18\ sb`

Derived from the KCPSM6 Manual

**AND Instruction**

- ANDs a register value with a constant or two register values
  - `and sx, constant` \( \rightarrow \) `sx = sx & \text{const.}`
  - `and sx, sy` \( \rightarrow \) `sx = sx & sy`

- Example: `and s3, 01`
  - Performs register `s3 = s3 & 1`
  - Before: `0xcf\ s3`
  - After: `0x01\ s3`

- Example: `and s3, sb`
  - Performs register `s3 = s3 & sb`
  - Before: `0x18\ s3`
  - After: `0x0f\ sb`

Derived from the KCPSM6 Manual

**LOAD Instruction**

- Loads a register value with a constant
  - `load sx, constant` \( \rightarrow \) `sx = \text{const.}`
  - `load sx, sy` \( \rightarrow \) `sx = \text{xy}`

- Example: `load s3, 05`
  - Performs register `s3 = 05`
  - Before: `??\ s3`
  - After: `s3`

- Example: `load s3, s9`
  - Performs register `s3 = s9`
  - Before: `??\ s3`
  - After: `12\ s3`

Derived from the KCPSM6 Manual

**DATA TRANSFER INSTRUCTIONS**
**FETCH Instruction**
- Reads (loads, fetches) data from a given address in memory into a register
  - `fetch sx, const_addr`
  - `fetch sx, (sy)`

**Example:** `fetch s3, 20`
- Reads data from memory address 20 and puts result into register s3

**Example:** `fetch s3, (sf)`
- Uses value in reg. sf as the mem. address, reading the data and placing it into register s3

Derived from the KCPSM6 Manual

**STORE Instruction**
- Writes (stores) data from a processor register into memory at a given address
  - `store sx, const_addr`
  - `store sx, (sy)`

**Example:** `store s3, 20`
- Stores data from s3 to memory address 20

**Example:** `store s3, (sf)`
- Stores data in s3 using the value in reg. sf as the mem. address

Derived from the KCPSM6 Manual

**Output Instruction**
- Writes (stores) data from a processor register onto the I/O bus for the given port_id (I/O address)
  - `output sx, const_addr` // out_port = sx
    - port_id = const_addr
  - `output sx, (sy)` // out_port = sx
    - port_id = sy

**Example:** `output s3, 40`
- Outputs data in s3 and sets the port_id (I/O address) to 40

**Example:** `output s3, (sf)`
- Outputs data in s3 and uses the value in sf as the port_id (I/O address)

Derived from the KCPSM6 Manual

**Input Instruction**
- Reads (loads) data from an I/O register at the given port_id (I/O address) into a processor register
  - `input sx, const_addr` // sx = in_port = sx
    - port_id = const_addr
  - `input sx, (sy)` // sx = in_port
    - port_id = sy

**Example:** `input s3, 40`
- Reads the data at I/O port address 40 and places the data into processor reg. s3

**Example:** `input s3, (sf)`
- Uses the contents of sf as the I/O port address and reads the data into processor reg. s3

Derived from the KCPSM6 Manual
### PROGRAM (CONTROL) FLOW INSTRUCTIONS

**COMPARE Instruction**

- Compares a register value with a constant or two register values by performing subtraction and updating the condition codes based on the result [if it is Negative (C) or Zero (Z)]
  - `compare sx, constant` // sx <=> const.
  - `compare sx, sy` // sx <=> sy

**Example: compare s3, 17**
- Performs register s3-17

**Example: compare s3, sf**
- Performs register s3-sf

```
Derived from the KCPSM6 Manual
```

**JUMP Instruction**

- Jumps (changes the PC) to a new instruction if the given condition is true, or continues sequentially if condition is false
  - `jump const_addr` // PC=const_addr
  - `jump Z, const_addr` // if(z) PC=const_addr
  - `jump NZ, const_addr` // if(!z) PC=const_addr
  - `jump {C,NC}, cons_addr`

**Example: jump Z, 100**
- Sets PC=100 only if Z=1, else PC++

**Example: jump NC, 100**
- Sets PC=100 only if C=0, else PC++

```
Derived from the KCPSM6 Manual
```

### Picoblaze Assembly 1

- Suppose a button is attached to the Picoblaze responding to PORT_ID=4
- Suppose an LED is attached to the Picoblaze responding to PORT_ID=12
- Turn on the LED when the button is pressed (i.e. btn => 0) and off when not pressed (i.e. btn => 1)

```
while(1) {
    if( btn == 0) // pressed
        LED = 1;
    else
        LED = 0;
}
```

```
L1: input  s1, __ // read button
    __            // btn == 1
    load s3, __   // LED = 1
    output s3, __ // LED = 1
    jump __;      // loop to top

L2: // btn was not pressed (btn == 1)
    load s3, __   // LED = 0
    output s3, __ // LED = 0
    jump __;      // loop to top
```
LED/Button Example

- HW/SW connections for a Button and LED

![LED Interface Diagram]

Keyboard Interface

Addr: 12

<table>
<thead>
<tr>
<th>Instruc Memory</th>
<th>Processor Data</th>
<th>Data Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>addr (PC)</td>
<td>addr</td>
</tr>
<tr>
<td>255</td>
<td>addr (instruct)</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>out_port</td>
<td>in_port</td>
</tr>
<tr>
<td>port_id</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

04 hex = 0001 0010 bin.

12 hex = 0001 0010 bin.

wire from bit 0 of register to LED

Picoblaze Assembly 2

- Suppose an ADC is connected to our Picoblaze with the ADCSRA at PORT_ID 20 and ADCH at PORTID 21
- Use polling to take a conversion and add that value to 10 elements in an array starting at memory address 80

```c
while(1) {
    ADCSRA |= (1 << 6);
    while((ADCSRA & (1 << 6)) != 0); // wait for conversion to finish
    unsigned char res = ADCH;
    for(int i=0; i < 10; i++){
    }
}
```

A-to-D Example

- HW/SW connections for communicating with the A-to-D converter

![A-to-D Converter Diagram]

Reminder: Register w/ Load Enable

- Registers (D-FF’s) will sample the D bit every clock edge and pass it to Q
- Sometimes we may want to hold the value of Q and ignore D even at a clock edge
- We can add an enable input and some logic in front of the D-FF to accomplish this

<table>
<thead>
<tr>
<th>CLK</th>
<th>RST</th>
<th>EN</th>
<th>D</th>
<th>Q*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>Q0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Q1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

4-bit register with 4-bit wide 2-to-1 mux in front of the D inputs