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 hardware
  - So that when an instruction is coded to binary it can fit 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>ADD F,Y</td>
<td>ADD X</td>
<td>PUSH Z</td>
</tr>
<tr>
<td>SUB F,F,Z</td>
<td>SUB F,Z</td>
<td>ADD Y</td>
<td>PUSH Y</td>
</tr>
<tr>
<td>ADD G,A,B</td>
<td>MOV G,A</td>
<td>ADD Z</td>
<td>SUB</td>
</tr>
<tr>
<td></td>
<td>ADD G,B</td>
<td>STORE F</td>
<td>PUSH X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOAD A</td>
<td>ADD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD B</td>
<td>POP F</td>
</tr>
<tr>
<td></td>
<td></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 Load/Store 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:** `add r1, r2 (rx = 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:** `add r1, m2 (mx = mem. operand)`
  - Memory/Memory Architecture (DEC VAX)
    - Operands of ALU instruc. Can be in memory
    - **Format:** `add m1, m2`
Addressing Modes

• Addressing modes refers to how an instruction specifies where the operands are
  – Can be in a register, memory location, or a constant that is part of the instruction itself (aka. immediate value)

• Most RISC processors: All data operands for arithmetic instructions must be in a register
  – This allows the hardware to be simpler and faster

• But what about something like: \( r8 = r8 + A[i] \) (\( A[i] \) 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 A[0] lives
  - Inefficient/Insufficient!
  - Would have to translate to:
    - CLR r1  # x = r1 = 0
    - LOAD r8, (0xa140)
    - ADD r1, r1, r8
    - LOAD r9, (0xa144)
    - ADD r1, r1, r8
    - ...

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

    loop:
    LOAD r8, (r9) # r8 = *r9
    ADD r1, r1, r8
    ADD r9, r9, 4
    repeat
  – Efficient/Sufficient!
Hardware/Software Interfacing

PICOBLAZE
Picoblaze

- Picoblaze (aka KCPSM6) is an 8-bit **soft-processor**
  - It is not "hard" in that there is no chip you can buy with just a Picoblaze processor
  - It is "soft" in that the processor design is given as intellectual property (IP)
  - 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

• 16 registers named s0-sf
  – Each register stores an 8-bit value
• PC is 12-bits allowing it to handle programs of up to 4K instructions
Normal Processor Bus Topology

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

[Diagram showing a typical processor bus topology with connections to memory, processor, video interface, and keyboard interface.]
PicoBlaze Processor Bus Topology

- Picoblaze has a separate:
  - Instruction memory / bus (Harvard architecture)
  - Data memory / bus
  - I/O bus
Performing operations on our data

SAMPLE ARITHMETIC/LOGIC INSTRUCTIONS
ADD Instruction

• Adds a register value with a constant or two register values
  – add sx, constant \( \text{// } sx = sx + \text{const.} \)
  – add sx, sy \( \text{// } sx = sx + sy \)

• 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       // sx = sx - const.
  – sub sx, sy           // sx = sx - sy

• Example: sub s3, 01
  – Performs register s3 = s3 - 1

• Example: sub s3, sb
  – Performs register s3 = s3 - sb

 Derived from the KCPSM6 Manual
AND Instruction

• AND a register value with a constant or two register values
  – and sx, constant // sx = sx & const.
  – and sx, sy // sx = sx & sy

• Example: \texttt{and s3, 01}
  – Performs register \texttt{s3 = s3 & 1}

• Example: \texttt{and s3, sb}
  – Performs register \texttt{s3 = s3 & sb}

Derived from the KCPSM6 Manual
Getting data in and out of our processor

DATA TRANSFER INSTRUCTIONS
LOAD Instruction

• Loads a register value with a constant
  – load sx, constant  // sx = const.
  – load sx, sy        // sx = xy

• Example: load s3, 05
  – Performs register s3 = 05

• Example: load s3, s9
  – Performs register s3 = s9

Derived from the KCPSM6 Manual
FETCH Instruction

• Reads (loads, fetches) **data** from a given **address** in memory into a **register**
  
  – **fetch**  **s3**,  **const_addr**

  – **fetch**  **s3**,  **(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)
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

Before: 16 s3
        - 17
After (sets condition codes): 1,0 C,Z

Before: 85 s3
        - 85 sf
After (sets condition codes): 0,1 C,Z

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, 04  // read button
    compare s1, 01  // btn == 1
    jump   Z, L2   // jump if btn==1
    // btn was pressed (btn == 0)
    load   s3, 1
    output s3, 12  // LED = 1
    jump   L1      // loop to top
L2: // btn was not pressed (btn == 1)
    load   s3, 0
    output s3, 12  // LED = 0
    jump   L1;     // loop to top
LED/Button Example

- HW/SW connections for a Button and LED

Instruct Memory

Processor

Data Memory

LED Interface

Addr: 12

0

255

port_id

out_port

in_port

04 hex = 0000 0100 bin.

12 hex = 0001 0010 bin.

wire from bit 0 of register to LED

Addr: 04

Q[7:0] EN CLK

D[7:0]

Addr (PC)

data (instruc)

addr
data

port_id

addr (PC)

data (instruc)

Addr: 04

Q[7:0] EN CLK

D[7:0]
• 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

```plaintext
while(1)
{
    ADCSRA |= (1 << 6);
    while((ADCSRA & (1 << 6)) != 0);

    unsigned char res = ADCH;
    for(int i=0; i < 10; i++){
    }
}
```

L1: load    s1, 40  // (1 << 6)=0x40
input   s2, 20  // get ADCSRA
or      s2, s1  // OR w/ (1 << 6)
output  s2, 20  // set ADCSRA
L2: input   s3, 20  // get ADCSRA
and     s3, s1  // AND w/ (1<<6)
compare s3, 0   // Check if 0
jump    NZ, L2  // Loop if not
input   s4, 21  // res = ADCH
load    s5, 0   // i=0
load    s6, 80  // array address
L3: fetch   s7, (s6)// Load A[i]
add     s7, s4  // A[i] += res
store   s7, (s6)// Store A[i]
add     s5, 1   // i++
add     s6, 1   // Move ptr over
compare s5, 10 // Check if last
jump    NZ, L3  // i != 10, loop
jump    L1      // Done, goto top
A-to-D Example

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

20 hex = 0010 0000 bin.
21 hex = 0010 0001 bin.
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_i )</th>
<th>( Q_i^* )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>( Q_i )</td>
</tr>
<tr>
<td>↑↑</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>↑↑</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>( Q_i )</td>
</tr>
<tr>
<td>↑↑</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>↑↑</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