Unit 16

Computer Organization and Instruction Sets
You Can Do That...

Cloud & Distributed Computing (CyberPhysical, Databases, Data Mining, etc.)

Applications (AI, Robotics, Graphics, Mobile)

Systems & Networking (Embedded Systems, Networks)

Architecture (Processor & Embedded HW)

Devices & Integrated Circuits (Semiconductors & Fabrication)

Where we will head now...

- Scripting & Interfaces
- C / C++ / Java
- Assembly / Machine Code
- OS
- Libraries
- Networked Applications
- Applications
- Processor / Memory / I/O
- Functional Units (Registers, Adders, Muxes)
- Logic Gates
- Transistors
- Voltage / Currents

(C) USC Viterbi School of Engineering
Motivation

• Now that you have some understanding...
  – Of how hardware is designed and works
  – Of how software can be used to control hardware

• We will look at how to improve efficiency of computer systems and software so that...
  – ...we can start to understand why HW companies create the structures they do (multicore processors)
  – ...we can begin to intelligently take advantage of the capabilities the HW gives us
  – ...we can start to understand why SW companies deal with some of the issues they do (efficiencies, etc.)
Computer Organization

• Three primary sets of components
  – Processor
  – Memory
  – I/O (everything else)

• Tell us where things live?
  – Running code
  – Compiled program (not running)
  – Circuitry to execute code
  – Source code file
  – Data variables
  – Data for the pixels being displayed on your screen
Input / Output

- Processor performs reads and writes to communicate with I/O devices just as it does with memory
  - I/O devices have locations (i.e. **registers**) that contain data that the processor can access
  - These registers are assigned unique addresses just like memory

![Diagram showing processor, memory, video interface, and keyboard interface with data flow](image-url)

- FE may signify a white dot at a particular location
- This could just as easily be the command and data register from the LCD shield… Or the PORT/DDR registers.

- ‘a’ = 61 hex in ASCII
- Processor memory
- Video Interface
- Keyboard Interface
Processor

- 3 Primary Components inside a processor
  - ALU
  - Registers
  - Control Circuitry
- Connects to memory and I/O via address, data, and control buses (bus = group of wires)
Arithmetic and Logic Unit (ALU)

• Executes arithmetic operations like addition and subtraction along with logical operations (AND, OR, etc.)
Registers

• Some are for general use by software
  – Registers provide fast, temporary storage locations within the processor (to avoid having to read/write slow memory)
• Others are required for specific purposes to ensure proper operation of the hardware
General Purpose Registers

- Registers available to software instructions for use by the programmer/compiler
- Instructions use these registers as inputs (source locations) and outputs (destination locations)
What if we didn’t have registers?

- Example w/o registers: $F = (X+Y) - (X*Y)$
  - Requires an ADD instruction, MULtiply instruction, and SUBtract Instruction
  - w/o registers
    - ADD: Load X and Y from memory, store result to memory
    - MUL: Load X and Y again from mem., store result to memory
    - SUB: Load results from ADD and MUL and store result to memory
    - 9 memory accesses
What if we have registers?

• Example w/ registers: \( F = (X+Y) - (X\times Y) \)
  - Load \( X \) and \( Y \) into registers
  - ADD: \( R0 + R1 \) and store result in \( R2 \)
  - MUL: \( R0 \times R1 \) and store result in \( R3 \)
  - SUB: \( R2 - R3 \) and store result in \( R4 \)
  - Store \( R4 \) back to memory
  - 3 total memory access
Other Registers

• Some bookkeeping information is needed to make the processor operate correctly

• Example: Program Counter (PC)
  – Recall that the processor must fetch instructions from memory before decoding and executing them
  – PC register holds the address of the currently executing instruction
Fetching an Instruction

- To fetch an instruction
  - PC contains the address of the instruction
  - The value in the PC is placed on the address bus and the memory is told to read
  - The PC is incremented, and the process is repeated for the next instruction
Fetching an Instruction

- To fetch an instruction
  - PC contains the address of the instruction
  - The value in the PC is placed on the address bus and the memory is told to read
  - The PC is incremented, and the process is repeated for the next instruction
Control Circuitry

- Control circuitry is used to decode the instruction and then generate the necessary signals to complete its execution.
- Controls the ALU.
- Selects registers to be used as source and destination locations (using muxes).

![Diagram showing the relationship between processor, control, ALU, and memory.]

- Processor:
  - ALU: ADD, SUB, AND, OR
  - Control:
    - PC: 0
  - Out: op.
  - In1: in1
  - In2: in2
  - R0-R15

- Memory:
  - Addr: inst. 1, inst. 2, inst. 3, inst. 4, inst. 5, ...
  - Data
  - Control
  - FF
Control Circuitry

- Assume 0x0201 is machine code for an ADD instruction of $R2 = R0 + R1$
- Control Logic will...
  - select the registers (R0 and R1)
  - tell the ALU to add
  - select the destination register (R2)
INSTRUCTION SETS
INSTRUCTION SET OVERVIEW
Instruction Sets

• 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.)
Instruction Set Architecture (ISA)

• 2 approaches
  – CISC = Complex instruction set computer
    • Large, rich vocabulary
    • More work per instruction, slower clock cycle
  – RISC = Reduced instruction set computer
    • Small, basic, but sufficient vocabulary
    • Less work per instruction, faster clock cycle
    • Usually a simple and small set of instructions with regular format facilitates building faster processors
### 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>MOVE ( F,X )</td>
<td>LOAD ( X )</td>
<td>PUSH Z</td>
</tr>
<tr>
<td>SUB ( F,F,Z )</td>
<td>ADD ( F,Y )</td>
<td>ADD ( Y )</td>
<td>PUSH Y</td>
</tr>
<tr>
<td>ADD ( G,A,B )</td>
<td>SUB ( F,Z )</td>
<td>SUB ( Z )</td>
<td>SUB</td>
</tr>
<tr>
<td>MOVE ( G,A )</td>
<td>ADD ( G,B )</td>
<td>STORE ( F )</td>
<td>PUSH X</td>
</tr>
<tr>
<td>ADD ( A )</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
  – 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.
  – Memory/Memory Architecture (DEC VAX)
    • Operands of ALU instruc. Can be in memory
    • ADD addrDst, addSrcl, addrSrc2

Load/Store Architecture

1.) Load operands to proc. registers
2.) Proc. Performs operation using register values
3.) Store results back to memory
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: \( r8 = r8 + A[i] \) \( (A[i] \text{ is in mem.}) \):
  - Intel instructions would allow: ADD r8, A[i]
    - A[i] is read from memory AND added to r8 in a single instruction
  - Other processors require 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, r9 \( (r8 = r8 + r9) \)
Load/Store Addressing

• When we load or store from/to memory how do we specify the address to use?
  – Note: Everything is a pointer at the instruction level

• Option 1: Direct Addressing
  – Address must be a constant: LOAD r8, (0xa140)
    • 0xa140 is just a made up address where we will assume A[0] lives
  – Insufficient!
  – Would have to translate to:
    • LOAD r8, (0xa140)
    • LOAD r9, (0xa144)
    • LOAD r10, (0xa144)
    • ...
Load/Store Addressing

• Option 2: Indirect 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
  – loop:
    LOAD r8, (r9)
    ADD r9, r9, 4
    repeat
  – Sufficient!

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

<table>
<thead>
<tr>
<th></th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>00</td>
</tr>
<tr>
<td>A[1]</td>
<td>00</td>
</tr>
</tbody>
</table>
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
PicoBlaze Processor Bus Topology

- PicoblaZe has a separate:
  - Instruction memory / bus
  - Data memory / bus
  - I/O bus
PICOBLAZE INSTRUCTION SET
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  // sx = sx + const.
  – add sx, sy       // 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: and s3, 01
  – Performs register s3 = s3 & 1

• Example: and s3, sb
  – Performs register 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
  – \texttt{load s3, constant}  // sx = \texttt{const}.

• Example: \texttt{load s3, 05}
  – Performs register \texttt{s3 = 05}

Derived from the KCPSM6 Manual
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 \textit{s3}, const\_addr
  – store \textit{s3}, (sy)

• Example: \texttt{store s3, 20}
  – Stores data from s3 to memory address 20

• Example: \texttt{store s3, (sf)}
  – Stores data in s3 using the value in reg. \textit{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 // output_port = sx
     // port_id = const_addr

  – output sx, (sy) // output_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
  - \texttt{input sx, const_addr}  
    \[
    \begin{array}{l}
    \text{sx} = \text{in_port} = \text{sx} \\
    \text{port_id} = \text{const_addr}
    \end{array}
    \]
  - \texttt{input sx, (sy)}  
    \[
    \begin{array}{l}
    \text{sx} = \text{in_port} \\
    \text{port_id} = \text{sy}
    \end{array}
    \]

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

- **Example:** \texttt{input s3, (sf)}
  - Uses the contents of sf as the I/O port address and reads the data into processor reg. \texttt{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. \( \text{btn} \Rightarrow 0 \)) and off when not pressed (i.e. \( \text{btn} \Rightarrow 1 \))

```assembly
L1: input s1, 04 // read button
    compare s1, 01 // btn == 1
    jump s1, 01 // 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

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

- HW/SW connections for a Button and LED

12 hex = 0001 0010 bin.

04 hex = 0000 0100 bin.
• 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

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

![Diagram showing connections between processor, data memory, and A-to-D converter](image)