Reading Assignment

• Course Policies and Procedures (web page)
• Lab Policies and Procedures (web page)
• Textbook: Chapter 1, “See a Program Running”, pages 1 – 26
Simple computer from ECE 270

- PC (Program Counter)
- ALU
- Memory (16 words)
- Control logic
- State machine
- Address Bus (4 bits)
- Data Bus (7)
- IR (Instruction Reg)
- Operation Code (3)
- A (Accumulator)
- NZCV flags
Instructions:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>xxxx HLT</td>
<td>Halt</td>
</tr>
<tr>
<td>001</td>
<td>xxxx LDA</td>
<td>Load from mem loc xxxx to A</td>
</tr>
<tr>
<td>010</td>
<td>xxxx ADD</td>
<td>Add mem loc xxxx to A</td>
</tr>
<tr>
<td>011</td>
<td>xxxx SUB</td>
<td>Subtract mem loc xxxx from A</td>
</tr>
<tr>
<td>100</td>
<td>xxxx AND</td>
<td>AND mem loc xxxx with A</td>
</tr>
<tr>
<td>101</td>
<td>xxxx STA</td>
<td>Store A to mem loc xxxx</td>
</tr>
<tr>
<td>110</td>
<td>xxxx INA</td>
<td>Load A from input</td>
</tr>
<tr>
<td>111</td>
<td>xxxx OUT</td>
<td>Display A</td>
</tr>
</tbody>
</table>

Example program:

<table>
<thead>
<tr>
<th>Addr.</th>
<th>Value</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>001</td>
<td>1011</td>
<td>LDA 1011 ; Load A from memory 1011</td>
</tr>
<tr>
<td>0001</td>
<td>010</td>
<td>1010</td>
<td>ADD 1010 ; Add memory location 1010 to A</td>
</tr>
<tr>
<td>0010</td>
<td>010</td>
<td>1001</td>
<td>ADD 1001 ; Add memory location 1001 to A</td>
</tr>
<tr>
<td>0011</td>
<td>010</td>
<td>1000</td>
<td>STA 1000 ; Store A to memory location 1000</td>
</tr>
<tr>
<td>0100</td>
<td>000</td>
<td>0000</td>
<td>HLT ; Halt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>What is the final result stored in 1000?</td>
</tr>
<tr>
<td>1000</td>
<td>000</td>
<td>0000</td>
<td>HLT</td>
</tr>
<tr>
<td>1001</td>
<td>000</td>
<td>0001</td>
<td>HLT</td>
</tr>
<tr>
<td>1010</td>
<td>000</td>
<td>0010</td>
<td>HLT</td>
</tr>
<tr>
<td>1011</td>
<td>000</td>
<td>0100</td>
<td>HLT</td>
</tr>
</tbody>
</table>
Assembler

- Turns symbolic expressions into raw bits.
  - LDA 1011  ==> 001 1011

- Abstracts memory locations into symbols
  - These are called **labels**
  - If we created a label called “var” for the location 1011, we could say: LDA var ...and it would turn into... 001 1011
  - A label is a **memory location**, AKA an **address**

- Does the same thing with numerical constants.
What if we had a C compiler?
(And an assembler.)

C code:

```c
int x = 2;
int y = 6;
int z = 3;

void main() {
    x = y - z;
}
```

Assembly code:

```
.text
main:
    LDA y
    SUB z
    STA x
.data
x:
    .word 2
y:
    .word 6
z:
    .word 3
```

Instructions follow
Label: assign an address
Data follows
Alloc space
And initialize

The assembler assigns the following addresses:
main: 0000
x: 1000
y: 1001
z: 1010

Machine code:

```
0000  LDA 1001 ; main
0001  SUB 1010
0010  STA 1000
0011  HLT 0000
... 1000  000 0010 ; x
1001  000 0110 ; y
1010  000 0011 ; z
```
How about loops?

```c
int x = 7;
int y = 6;
int z = 1;

void main() {
    x = z;
    do {
        x = x + y;
        y = y - z;
    } while (y != 0);
}
```

The simple computer was extended with the JNE instruction for this purpose! (Few people understood why.) JNE tests the Z flag and Jumps if it is Not Equal to zero.

```
.text
main:  lda z   ; x = z;
       sta x
L0:    lda x   ; x = x + y
       add y
       sta x
       lda y   ; y = y - z
       sub z
       sta y
       lda y   ; while y != 0
       jne L0
       hlt

.data
x: .word 7
y: .word 6
z: .word 1
```
How about function calls?

- We need something to "remember" where we left off when we called a 2\textsuperscript{nd} function.

- We might want to call a 3\textsuperscript{rd} function inside the 2\textsuperscript{nd} function.
  - We might even want to call a function from itself. (Recursion)
  - We need an "unlimited" number of places to remember where we left off.

```c
int x = 7;
int y = 6;
int z = 1;

void main() {
    x = z;
    do {
        more();
        y = y - z;
    } while (y != 0);
}

void more() {
    x = x + z;
    return;
}
```
Added a "stack pointer"

• SP starts at high address.

• A "JSR xxxx" instruction does the following:
  - Decrement SP.
  - Put the PC value in the memory location pointed to by the new SP.
  - Jump to the address xxxx.

• This is called "Jump to Subroutine"
Added a "stack pointer"

- An "RTS xxxx" instruction does the following:
  - Get the value of the memory location pointed to by SP and place it in PC.
  - Increment SP.
  - (xxxx is ignored).

- RTS “undoes” everything JSR does.
How about function calls?

```c
int x = 7;
int y = 6;
int z = 1;

void main() {
    x = z;
    do {
        more();
        y = y - z;
    } while (y != 0);
}

void more() {
    x = x + z;
    return;
}
```

```assembly
.text
main:  lda z      ; x = z;
       sta x
L0:   jsr more    ; more()
       lda y      ; y = y - z
       sub z
       sta y
       lda y      ; while y != 0
       jne L0
       hlt
more: lda x      ; x = x + z
       add z
       sta x
       rts          ; return
.data
x:  .word 7
y:  .word 6
z:  .word 1
```

How many memory words am I using with this program?

If SP starts at the top of memory, do we even have any memory to use for JSR?
Other stack operations

- Simple computer was also extended to support push (PSH) and pop (POP).
- It’s too hard to demonstrate those with C.
  - And we’re already out of memory.
Why look at the simple computer?

- Once you understand assembly language for one machine, you have no trouble with any other.

```
data
x: .long 7
y: .long 6
z: .long 1
.text
more:
    movl x, %ecx
    movl z, %edx
    addl %ecx, %edx
    movl %edx, x
    ret
main:
    movl z, %eax
    movl %eax, z
.L4:
call more
    movl y, %edx
    movl z, %eax
    subl %eax, %edx
    movl %edx, y
    movl y, %eax
    testl %eax, %eax
    jne .L4
ret
```

This is the 32-bit x86 architecture.
Let’s move on to ARM Cortex-M0
Development system for ECE 362

- STM32F0-Discovery board
- Contains STM32F051R8T6.
- Integrated programmer.
- Headers for LQFP64 I/O pins.
- 2 CPU-controllable LEDs.
- 2 push buttons
What is a development board for?

• It’s usually too hard to use a bare microcontroller
  – Solder the LQFP64 package
  – Add all the support electronics

• This provides an easier start.

• Finished product will use the raw chip.
  – Your mini-projects can use the development board.

• There are two qualities of an “embedded” CPU.
Embedded CPU has lots of gadgets

- 32-bit ARM Cortex-M0 CPU.
- 8MHz clock multiplied x6 by a PLL.
- 64kB flash. 8KB SRAM.
- 32 maskable interrupt channels w/ 4 priority levels
- 12-bit ADC (multiple channels), 12-bit DAC
- 2 SPI, 1 I²S, 2 I²C, 2 USARTs
- Several timers with integrated PWM
- So much more:
  - https://engineering.purdue.edu/ece362.refs/
Embedded CPU Is Low Power

- Maximum power dissipation: 444mW
- No heatsink needed. Easy to “embed.”
- Sometimes people try to embed a conventional high-performance CPU.
  - When the fan stops working, it burns their customer’s house down.
- By comparison, this laptop has an Intel Core i7-3720QM. 45W TDP. Not good for embedded use.
Architecture

- Little Endian. Just like Intel x86.
- 16 registers. Looks simple.
  - R15 is the PC.
  - R13 is the SP.
  - R14 is the "Link Register" (LR)
  - Most instructions can only use R0 – R7.
- Program Status Register.
  - Upper four bits are the flags.
### Memory Model

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFFFFF</td>
<td>Private peripherals including NVIC and debug components</td>
</tr>
<tr>
<td>0xE0000000</td>
<td>Mainly used for external peripherals.</td>
</tr>
<tr>
<td>0xE0000000</td>
<td>Mainly used for external memory.</td>
</tr>
<tr>
<td>0x60000000</td>
<td>Mainly used for peripherals.</td>
</tr>
<tr>
<td>0x30000000</td>
<td>Mainly used for data memory (e.g. static RAM)</td>
</tr>
<tr>
<td>0x00000000</td>
<td>Mainly used for program code. Also used for default exception vector table</td>
</tr>
</tbody>
</table>

#### Diagram

- **System**
- **Private Peripheral Bus**
- **System Control Space (SCS)**
- **Private Peripheral Bus (PPB)**

- **External Device**: 1GB
- **External RAM**: 1GB
- **Peripherals**: 0.5GB
- **SRAM**: 0.5GB
- **CODE**: 0.5GB

Peripherals are “memory mapped” starting at 0x4000 0000. You can read and write them as though they were memory.

- `.data` starts at 0x2000 0000
- `.text` starts at 0x0800 0000
Next topic

• The ARM Cortex-M0 Instruction Set