Design of the WashU-2 Processor – Part 1

- Basic operation
- Instruction set
- Programming and simulating the processor

Jon Turner
WashU-2 Block Diagram

- Memory stores programs and data
  - organized as set of numbered memory words
  - memory word can hold a 16 bit value
  - processor can read from or write to any word
- Fetch & execute cycle
  - read word whose address is in Program Counter (PC) and store it in Instruction Register (IR); then increment PC
  - perform instruction using Accumulator (ACC), Arithmetic & Logic Unit (ALU) and possibly Indirect Address Register (IAR)
- IREG, PC, IAR and ACC are all 16 bit registers
- Controller coordinates actions of other components
- Data & Address Buses carry data between components
Sign Extension

- Sign extension is used to convert a number to a format with more bits, but the same sign and magnitude
  - for positive values (sign bit=0) pad to the left with 0s
  - for negative values (sign bit=1) pad to the left with 1s
  - so, 0110 becomes 00110 and 1010 becomes 11010

- To see why this works, consider converting from 4 bits to 5 bits

![Diagram showing sign extension from 4 bit to 5 bit format]
WashU-2 Instruction Set

0000  *halt* – halt execution
0001  *negate* – ACC := –ACC
01xx  *branch* – PC := PC + ssxx (sign-extended addition)
02xx  *branch if zero* – if ACC = 0 then PC := PC + ssxx
03xx  *branch if positive* – if ACC > 0 then PC := PC + ssxx
04xx  *branch if negative* – if ACC < 0 then PC := PC + ssxx
05xx  *indirect branch* – PC := M[PC+ssxx]
1xxx  *constant load* – ACC := ssxx
2xxx  *direct load* – ACC := M[pxxx] (p=high digit of PC)
3xxx  *indirect load* – IAR := M[pxxx]; ACC := M[IAR]
5xxx  *direct store* – M[pxxx] := ACC
6xxx  *indirect store* – IAR := M[pxxx]; M[IAR] := ACC
8xxx  *add* – ACC := ACC + M[pxxx]
cxxx  *and* – ACC := ACC and M[pxxx]
# Simple Program

- Repeat: wait for new input to be present;
  read input and add it to running sum

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1000 (ACC = 0)</td>
<td>-- sum = 0;</td>
</tr>
<tr>
<td>0001</td>
<td>5011 (M[0011] = ACC)</td>
<td></td>
</tr>
</tbody>
</table>
| 0002 (loop) | 1FFF (ACC= -1) | -- while (true) {
| 0003    | 5010 (M[0010] = ACC) | -- inVal = -1; |
| 0004    | 1001 (ACC = 1) | -- while (inVal == -1) {} |
| 0005    | 8010 (ACC = ACC+M[0010]) | |
| 0006    | 02FE (if ACC=0 go back 2) | |
| 0007    | 2010 (ACC = M[0010]) | -- if (inVal == 0) break; |
| 0008    | 0204 (if ACC=0 go ahead 4) | |
| 0009    | 8011 (ACC = ACC + M[0011]) | -- sum += inVal; |
| 000A    | 5011 (M[0011] = ACC) | |
| 000B    | 01F7 (go back 9) | -- } |
| 000C (end) | 0000 (halt) | -- halt |

...  

0010  1234  -- input value  
0011  5678  -- sum of input values
## Assembly Language Version

<table>
<thead>
<tr>
<th>label</th>
<th>Instruction [arg]</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>location 0</td>
<td></td>
<td>-- sum = 0;</td>
</tr>
<tr>
<td>cLoad 0</td>
<td></td>
<td>--</td>
</tr>
<tr>
<td>store sum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop:</td>
<td>cLoad -1</td>
<td>-- while (true) {}</td>
</tr>
<tr>
<td></td>
<td>store inValue</td>
<td>-- inVal = -1;</td>
</tr>
<tr>
<td></td>
<td>cLoad 1</td>
<td>-- while (inVal == -1) {}</td>
</tr>
<tr>
<td></td>
<td>add inValue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>brZero -2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dLoad inValue</td>
<td>-- if (inVal == 0) break;</td>
</tr>
<tr>
<td></td>
<td>brZero end</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add sum</td>
<td>-- sum += inVal;</td>
</tr>
<tr>
<td></td>
<td>dStore sum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>branch loop</td>
<td>-- }</td>
</tr>
<tr>
<td>end:</td>
<td>halt</td>
<td>-- halt;</td>
</tr>
<tr>
<td></td>
<td>location 0010</td>
<td></td>
</tr>
<tr>
<td>inVal:</td>
<td>01234</td>
<td>-- input value</td>
</tr>
<tr>
<td>sum:</td>
<td>05678</td>
<td>-- sum of input values</td>
</tr>
</tbody>
</table>
Using the Assembler

- Compile and run by typing
  - `javadoc Assembler.java`
  - `java Assembler sourceFile`

- Paste output from assembler into VHDL source for memory – in memory initialization section

```
16#0000# => x"1000",        -- sum = 0;
16#0001# => x"5011",        
16#0002# => x"1fff",        -- while (true) {
16#0003# => x"5010",        -- inVal = -1;
16#0004# => x"1001",        -- while (inVal = -1) {} 
16#0005# => x"8010",        
16#0006# => x"02fe",        
16#0007# => x"2010",        -- if (inVal == 0) break;
16#0008# => x"0204",        
16#0009# => x"8011",        -- sum += inVal
16#000a# => x"5011",        
16#000b# => x"01f7",        -- }
16#000c# => x"0000",        -- halt
16#0010# => x"1234",        -- input value
16#0011# => x"5678",        -- sum of input values
```
Simulation – Global View

1 loaded into M[10];
program then adds it to M[11]

special monitoring registers for M[10], M[11]
Exercises

1. Rewrite the following machine code fragment in assembly language.

```
1234  101a
1235  8236
1236  0001
1237  03fe
```

1. The diagram below shows the processor registers for the Washu-2 processor and the contents of several memory locations. In the space to the right of the registers, write the values the registers would have after the next two instructions are executed. Similarly, show updated values of any memory locations that are changed, to the right of the memory diagram. (Assume that the "current instruction" has not yet been fetched from memory.)

```
<table>
<thead>
<tr>
<th>Processor</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>IREG</td>
<td>63AA</td>
</tr>
<tr>
<td>PC</td>
<td>9001</td>
</tr>
<tr>
<td>ACC</td>
<td>7A02</td>
</tr>
<tr>
<td>IAR</td>
<td>1234</td>
</tr>
<tr>
<td>8FFC</td>
<td>20AB</td>
</tr>
<tr>
<td>8FFD</td>
<td>8FFC</td>
</tr>
<tr>
<td>8FFE</td>
<td>0123</td>
</tr>
<tr>
<td>8FFF</td>
<td>83A1</td>
</tr>
<tr>
<td>9000</td>
<td>63AA</td>
</tr>
<tr>
<td>9001</td>
<td>03FC</td>
</tr>
<tr>
<td>9002</td>
<td>3003</td>
</tr>
<tr>
<td>9003</td>
<td>8FFC</td>
</tr>
</tbody>
</table>
```
Exercises

1. The Washu-2 processor simulation at right has several places that have been replaced with blank spaces labeled with letters. In the spaces below, fill in the values that should appear where the blanks are.
   
   A.
   B.
   C.
   D.
   E.
Using the WashU-2 Processor

- Reset by setting pressing “north” button
  - this restarts the processor, but does not re-initialize memory
  - use program button to re-initialize memory

- Using single step mode
  - press west button to enter single-step mode, press again to execute one instruction
  - press south button to exit single step mode

- Using display
  - display shows
    - IREG  ACC  SnoopAdr
    - PC  IAR  SnoopData
  - when switch(3)=0, knob controls Snoop Address register
  - when switch(3)=1, knob controls Snoop Data register
    - press east button to load SnoopData value into M[SnoopAdr]
  - press down on knob to change value by larger increments
WashU-2 Memory Map

- WashU-2 is designed to address up to $2^{16} = 65,536$ words
- FPGA used to implement it only has enough memory for 20,480
- Convenient to organize memory into “pages” of 4,096 words
  - enough memory for 5 pages
  - first 4 for programs and data
  - last one for display buffer
- Display buffer holds pixel data
  - 120 rows of 160 pixels each
    - top-left to bottom-right
  - 5 pixels per 16 bit word (3 bits each)
  - 0=black, 1=blue, 2=green, 3=cyan
    - 4=red, 5=magenta, 6=yellow, 7=white
Using Subprograms

<table>
<thead>
<tr>
<th>Location 0100</th>
<th>int mult(int a, int b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult_a: 0</td>
<td>first argument</td>
</tr>
<tr>
<td>mult_b: 0</td>
<td>second argument</td>
</tr>
<tr>
<td>mult_prod: 0</td>
<td>return value</td>
</tr>
<tr>
<td>mult_ret: 0</td>
<td>return address</td>
</tr>
<tr>
<td>mult:</td>
<td>prod = 0</td>
</tr>
<tr>
<td></td>
<td>mask = 1</td>
</tr>
<tr>
<td>mult_loop:</td>
<td>while (mask != 0) {</td>
</tr>
<tr>
<td></td>
<td>if ((b &amp; mask) != 0)</td>
</tr>
<tr>
<td></td>
<td>prod += a;</td>
</tr>
<tr>
<td></td>
<td>a &lt;&lt;= 1;</td>
</tr>
<tr>
<td></td>
<td>mask &lt;&lt;= 1;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>mult_end:</td>
<td>branch mult_loop</td>
</tr>
<tr>
<td></td>
<td>return prod;</td>
</tr>
<tr>
<td>mult_mask: 0</td>
<td>mask</td>
</tr>
</tbody>
</table>

14
Convert ASCII String to Binary

```c
location 0200  -- int asc2bin(char* p, int n) {
    a2b_p: 0  -- first argument
    a2b_n: 0  -- second argument
    a2b_num: 0  -- return value
    a2b_ret: 0
    asc2bin: cLoad 0  -- num = 0;
    dStore a2b_num
    a2b_loop: dLoad a2b_n  -- while (n != 0) {
        brZero a2b_end
        cLoad 10  -- num *= 10;
        dStore mult_a
        dLoad mult_num
        dStore mult_b
        cLoad a2b_retLoc
        dStore mult_ret
        iBranch 1
        mult
        a2b_retLoc: dStore a2b_num  -- num += (*p - '0');
        cLoad -30
        dStore a2b_temp
        iLoad a2b_p
        add a2b_temp
        add a2b_num
        dStore a2b_num
    }
```
cLoad 1 -- p++;
add a2b_p
dStore a2b_p
cload -1 -- n--;
add a2b_n;
dStore a2b_n;
branch a2b_loop --
} a2b_end:
ibBranch a2b_ret -- return num
a2b_stop: 0 -- temporary storage
Exercises

1. Which of the registers in the WashU-2 processor can be loaded from memory? For which registers can their values be stored in memory? Which instructions change the value of the accumulator? Which can change the value in a memory location?

2. Write a program in assembly language for the WashU-2 processor that makes all the pixels on the fifth line of the VGA display blue.

3. Write a subprogram `toLowerCase` in assembly language for the WashU-2 processor that takes two argument: a pointer to a memory location `p` and a count `n`. It examines the `n` memory locations starting at location `p` and interprets the value in each location as an ASCII character code. For each character code that corresponds to an upper case letter, it replaces the character with the code for the corresponding lower-case letter (So, 'H' becomes 'h' and so forth).