ECE 52 Protocomputer

Component Walkthrough
ALU, Registers, Control; Memory next

Memory Operations
Since this machine only has one general purpose register, most operands and most results must be migrated to and from the main memory. MAR, MBR
Basic Instructions

READ
ADD
AND
OR
WRITE
NEG
JNZ
HLT
(leave JSR, RET, SHIFTs for later)

Simple Program Walkthrough

Simple program for the Protocomputer: takes the absolute value (in 2's complement sense) of number stored at memory location 275\text{10} and stores it in memory location 276\text{10}.

Assumptions:

- Program and associated data have already been loaded into memory at the locations specified
- Specifically, a value of interest has already been placed in location 275, perhaps by an external sensor or I/O device, for instance
- Program Counter (PC) is initialized to 0 at power-up so the program can safely begin by having its first instruction at location 0.

<table>
<thead>
<tr>
<th>LOC</th>
<th>LABEL</th>
<th>INSTRUCTION</th>
<th>BINARY</th>
<th>HEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>START</td>
<td>READ 275</td>
<td>001000100010011</td>
<td>1113</td>
</tr>
<tr>
<td>1</td>
<td>WRITE</td>
<td>276</td>
<td>001000100010100</td>
<td>2114</td>
</tr>
<tr>
<td>2</td>
<td>AND</td>
<td>MASK</td>
<td>0100000000001011</td>
<td>400A</td>
</tr>
<tr>
<td>3</td>
<td>JNZ</td>
<td>INVERT</td>
<td>0110000000001011</td>
<td>6005</td>
</tr>
<tr>
<td>4</td>
<td>HLT</td>
<td></td>
<td>1111000000000000</td>
<td>F000</td>
</tr>
<tr>
<td>5</td>
<td>INVERT</td>
<td>READ 275</td>
<td>0001000100010011</td>
<td>1113</td>
</tr>
<tr>
<td>6</td>
<td>NEG</td>
<td></td>
<td>0011000000000000</td>
<td>3000</td>
</tr>
<tr>
<td>7</td>
<td>ADD</td>
<td>ONE</td>
<td>0000000000001011</td>
<td>000B</td>
</tr>
<tr>
<td>8</td>
<td>WRITE</td>
<td>276</td>
<td>0010000100010100</td>
<td>2114</td>
</tr>
<tr>
<td>9</td>
<td>HLT</td>
<td></td>
<td>1111000000000000</td>
<td>F000</td>
</tr>
<tr>
<td>10</td>
<td>MASK</td>
<td>DC 8000H</td>
<td>1000000000000000</td>
<td>8000</td>
</tr>
<tr>
<td>11</td>
<td>ONE</td>
<td>DC 0001H</td>
<td>0000000000000001</td>
<td>0001</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NB I could have named 275 as “X”, and 276 as “Y” and then written instructions like READ X, WRITE Y.
Protocomputer Programming (continued)

Protocomputer Operation

All instructions will take a number of primitive steps to process. All instructions go through three distinct phases:

1. Fetch – get the next instruction, whatever it is, out of memory and (eventually) into the IR where it can be processed further; the Program Counter tells us where to get it. Fetch is common to all instructions!

2. Decode – trivial on the Protocomputer but a more complex operation on more sophisticated machines: determine which of our 12 instructions the current one is. A 4-to-16 decoder will do the job.

3. Execute – specific to each of our 12 instructions; at this point we have it (fetch) and know what it is (decode) so a specialized set of operations is ok.

The Fetch Cycle

1. MAR ← PC ; move address of next instr to MAR
2. MBR ← MEM[MAR]; i.e. READ!
3. IR ← MBR ; move instruction to IR
4. PC ← PC + 1 ; to avoid infinite loop!

• Common to all instructions
• Neither knows nor cares what instruction it has read; that is DECODE and EXECUTE’s job
• PC ← PC + 1 takes care of default case; JNZ, JSR can override

The Decode Phase

Trivial on the Protocomputer; simply examine 4 MSB of IR with 4-to-16 decoder to determine which instruction type this is; does not consume a “cycle”. More complex machines will beef this up.

The Execute Cycle
**READ:**
1. \( \text{MAR} \leftarrow \text{IR}[4..15] \); move only 12 LSB of IR to MAR
2. \( \text{MBR} \leftarrow \text{MEM}[\text{MAR}] \); read number from memory
3. \( \text{AC} \leftarrow \text{MBR} \); move number where we need it
4. <blank for now>

**WRITE:**
1. \( \text{MAR} \leftarrow \text{IR}[4..15] \); move only 12 LSB of IR to MAR
2. \( \text{MBR} \leftarrow \text{AC} \); arbitrary order with first
3. \( \text{MEM}[\text{MAR}] \leftarrow \text{MBR} \); write!
4. <blank for now>

**AND, ADD, OR:**
1. \( \text{MAR} \leftarrow \text{IR}[4..15] \)
2. \( \text{MBR} \leftarrow \text{MEM}[\text{MAR}] \); left operand now available
3. \( \text{AC} \leftarrow \text{AC} \land \text{MBR} \); both connected to ALU
4. <blank for now>

For ADD, OR, simply substitute the following in the above code:
1. \( \text{AC} \leftarrow \text{AC} + \text{MBR} \); ADD
   or
1. \( \text{AC} \leftarrow \text{AC} \lor \text{MBR} \); OR

**JNZ:**
1. IF (AC != 0) PC \( \leftarrow \text{IR}[4..15] \)

**NOT, RSHIFT, LSHIFT (Unary):**
1. \( \text{AC} \leftarrow \sim\text{AC} \)
   or
1. \( \text{AC} \leftarrow \text{LeftShift}(\text{AC}) \)
   or
1. \( \text{AC} \leftarrow \text{RightShift}(\text{AC}) \); Arithmetic or logical?

**HLT:**
1. <STOP THE MACHINE!>

We are leaving JSR, RET out for now.

**ALU Implementation**
Points:

- ALU can perform 7 functions, including passthrough; 3 are binary (in operand sense), 4 are unary
- Right operand for all binary ops comes from AC via special path in original Protocomputer diagram
- All unary ops operate on Right input from AC
- All 7 functions are computed all the time! About 7 cycles out of 8, we don’t want any of the outputs; 1 cycle out of 8 we want one of the 7 possible outputs.
- Thus inputs to units are garbage often, but MUX keeps unneeded and indeed illegal values out of the system
- We don’t wear out an adder by overusing it!
Control Unit Implementation

Figure 3: ECE 52 Protocomputer Control Unit

Points:
• Each Register is modeled after design in upper left; note that MAR and PC could be limited to 12 bits if desired; IR, PC, SP, MBR all must be 16 bits.
• 2 flip-flops establish three high-level states; FETCH, EXECUTE; HALT

We now can express the state changes from FETCH to EXECUTE; EXECUTE to FETCH, and EXECUTE to HALT in terms of our flip flops with signals S0_SET S0_RESET S1_SET S1_RESET

The Fetch Phase

1. MAR ← PC ; move address of next instr to MAR
2. MBR ← MEM[MAR]; i.e. READ!
3. IR ← MBR ; move instruction to IR
4. PC ← PC + 1; S1_SET; S0_RESET ; to avoid infinite loop
Typical Execute Phase

**READ**
1. MAR ← IR[4..15] ; move only 12 LSB of IR to MAR
2. MBR ← MEM[MAR]; read number from memory
3. AC ← MBR ; move number where we need it
4. S1_RESET; S0_SET

Wiring the Fetch Phase
<exercise>

Wiring Execute Phases

**READ**
<exercise>

**ADD**
<exercise>
Note that ADD requires MBR_OUT_ENA, ADD_ENA, and AC_LATCH to implement AC<-AC+MBR

**JNZ**
<exercise>

Final Connections

We see that, from our sea of AND gates, some signals are more popular than others: MBR_IN_ENA, for instance, is triggered by many sources. Note that, electrically, the outputs of all these AND gates cannot just be simply wired together to the Input Enable of the MAR; we need to connect them via an OR gate. Some specific technologies permit “wired-OR” operation, but in many others it
causes a short circuit, so we will draw the OR gates explicitly.

**JSR/RET RTL for protocomputer**

**JSR**
1. MAR = SP
2. MBR = PC
3. MEM[MAR] = MBR
4. SP--; PC = IR[4..15]; <fetch state>

**RET**
1. SP++
2. MAR = SP
3. MBR = MEM[MAR]
4. PC = MBR; <fetch state>

**Improving the performance of the protocomputer**

JSR above illustrates a “trick” we can apply to the protocomputer to speed it up significantly. JSR really requires 6 steps, but we note that three of them can be done concurrently on the last cycle because they are independent and non-interfering. In general, we can turn on any combination of control signals in a single clock cycle that are mutually independent and accomplish more than one task at a time.

We can apply this to the FETCH phase: Recall we had
1. MAR = PC; move address of next instr to MAR
2. MBR = MEM[MAR]; i.e. READ!
3. IR = MBR; move instruction to IR
4. PC = PC + 1; S1_SET; S0_RESET; to avoid infinite loop

and advance to execute phase. But note that we’re done with the program counter in the first cycle, so there’s no reason to wait until the fourth cycle to increment it:

1. MAR = PC; move address of next instr to MAR
2. MBR = MEM[MAR]; i.e. READ!
3. IR = MBR; PC = PC + 1
4. S1_SET; S0_RESET

But now we’re doing nothing “useful” in the fourth phase; we can move the “go to execute” commands to cycle 3 if we also invoke one additional signal:
1. MAR ← PC; move address of next instr to MAR
2. MBR ← MEM[MAR]; i.e. READ!
3. IR ← MBR; PC ← PC + 1; S1_SET; S0_RESET; COUNTER_RESET

This one simple change will speed up the protocomputer by 12.5%
as we have eliminated 1 in 8 cycles!

The same optimizations can be applied to most of the execute
phases as well; take for example NEG: it can be rewritten as a
single cycle

**NEG**

1. AC ← ~AC; S1_RESET; S0_SET; COUNTER_RESET

Applying these optimizations will speed up the protocomputer by
about 50%; instructions will now take somewhere between 4 and 7
cycles to complete (3 to fetch, and between 1 and 4 to execute).