Designing a Computer from Ground-up

Specification/design document
N. Narasimhamurthi

1. Register Organization

1.1 External Registers

These are the registers available to the programmer. The machine we are designing will have four registers numbered R0 to R3. All registers are treated the same, except R3 is the program counter. Thus programmers would not explicitly modify R3 except for branching instructions. The machine will not have any explicit conditional/unconditional branch instructions.

1.2 Internal Registers

In addition to the external registers, the machine will have the following internal registers. These registers are not available to the programmer or the programmer will have only limited access to these.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC</td>
<td>Sequence Counter</td>
<td>Generates the timing signals T0, T1, T2, ...</td>
</tr>
<tr>
<td>AR</td>
<td>Address Register</td>
<td>Holds the address for memory access</td>
</tr>
<tr>
<td>IR</td>
<td>Instruction Register</td>
<td>Holds the current instruction being evaluated</td>
</tr>
<tr>
<td>SP</td>
<td>Stack Pointer</td>
<td>Holds the address of the last used address.</td>
</tr>
<tr>
<td>TR</td>
<td>Temporary register</td>
<td>Temporary register for internal use</td>
</tr>
<tr>
<td>INPR</td>
<td>Input char</td>
<td>Holds input character</td>
</tr>
<tr>
<td>OUTR</td>
<td>Output char</td>
<td>Holds output character</td>
</tr>
</tbody>
</table>

Table 1 Internal Registers

Note: The stack will grow up the memory, i.e. start at a high address and move towards a lower address as items are pushed to the stack.

1.3 Flip flops

The computer will have the following flip-flops

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Carry Flag</td>
<td>Part of ALU.</td>
</tr>
<tr>
<td>N</td>
<td>Negative Flag</td>
<td>Part of ALU. Is the Bit15 of the last ALU operation</td>
</tr>
<tr>
<td>V</td>
<td>Overflow Flag</td>
<td>Part of ALU. Is set if the result requires 17 bits</td>
</tr>
</tbody>
</table>

2. Memory Organization

Because of the limitations on the number of bits in an instruction, the machine will have 4K memory. Memory addresses will be 12 bits long.

3. Instruction Format

3.1 Instructions involving a register and memory address

These instructions have the bit 15 of the instruction cleared. There are two instructions, LD and ST for load and store. The bit 14 is zero for LD and 1 for ST. Bits 12 and 13 specify the register. Bits 0-11 the address.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>R</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>ST</td>
<td>0</td>
<td>1</td>
<td>R</td>
<td>R</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

Table 3: Register/Memory Instructions

3.2 Instructions involving two registers

These instructions are identified by a zero in bits 7 and 14. Bits 12 and 13 specify the destination (D) register and bits 0 1 the source (S) register. There are four modes of transfer: normal, indirect, indirect and decrement, and decrement and indirect. These correspond to S, *S, *S++, *--S in C program. The mode is specified in bits 2 and 3 of the instruction. The general format is as follows:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>D</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
</tr>
</tbody>
</table>

Table 4: Register/Register instructions

In the table xxxx specifies the operation and MM denotes the addressing mode. The table below defines the modes.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>The value in the register is the source</td>
</tr>
<tr>
<td>0 1</td>
<td>The value in the register is the address of the source</td>
</tr>
</tbody>
</table>
### Specifications for a simple computer

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>Decrement the register first. The value in the register is the address of the source.</td>
</tr>
<tr>
<td>1 1</td>
<td>The value in the register is the address of the source. After fetching the source, increment the register.</td>
</tr>
</tbody>
</table>

**Table 5: Addressing modes**

The operations specified by \texttt{xxxx} are as shown below.

<table>
<thead>
<tr>
<th>\texttt{xxxx}</th>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>MOV</td>
<td>Transfer</td>
</tr>
<tr>
<td>0001</td>
<td>CMP</td>
<td>Compare</td>
</tr>
<tr>
<td>0010</td>
<td>ADD</td>
<td>Add w/o carry</td>
</tr>
<tr>
<td>0011</td>
<td>ADC</td>
<td>Add with carry</td>
</tr>
<tr>
<td>0100</td>
<td>SUB</td>
<td>Subtract w/o carry (borrow)</td>
</tr>
<tr>
<td>0101</td>
<td>SBC</td>
<td>Subtract with carry</td>
</tr>
<tr>
<td>0110</td>
<td>AND</td>
<td>And</td>
</tr>
<tr>
<td>0111</td>
<td>OR</td>
<td>Inclusive or</td>
</tr>
<tr>
<td>1000</td>
<td>EOR</td>
<td>Exclusive or</td>
</tr>
</tbody>
</table>

**Table 6: Op codes for R/R instructions**

Note we have room for 7 more instructions!

### 3.3 Single Register Instructions

These instructions are identified by a one in bit 7 and zero in bit 14. Bits 12 and 13 specify the destination (D) register.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>D</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 7: Register/Register instructions**

The operations specified by \texttt{xxxx} are as shown below. Note that we have space for one more instruction.
3.4 Single Register Offset Instructions

These instructions are identified by a one in bit 14 and a zero in bit 11. Bits 12 and 13 specify the destination (D) register.

<table>
<thead>
<tr>
<th>XXXX</th>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>CLR</td>
<td>Clear</td>
</tr>
<tr>
<td>0001</td>
<td>TST</td>
<td>Test for Zero</td>
</tr>
<tr>
<td>0010</td>
<td>INC</td>
<td>Increment</td>
</tr>
<tr>
<td>0011</td>
<td>DEC</td>
<td>Decrement</td>
</tr>
<tr>
<td>0100</td>
<td>ROL</td>
<td>Rotate Left</td>
</tr>
<tr>
<td>0101</td>
<td>ROR</td>
<td>Rotate Right</td>
</tr>
<tr>
<td>0110</td>
<td>LSL</td>
<td>Logical Shift Left</td>
</tr>
<tr>
<td>0111</td>
<td>LSR</td>
<td>Logical Shift Right</td>
</tr>
<tr>
<td>1000</td>
<td>ASR</td>
<td>Arithmetic Shift Right</td>
</tr>
<tr>
<td>1001</td>
<td>NEG</td>
<td>2s Complement</td>
</tr>
<tr>
<td>1010</td>
<td>COM</td>
<td>1s Complement</td>
</tr>
<tr>
<td>1011</td>
<td>PUSH</td>
<td>Push register</td>
</tr>
<tr>
<td>1100</td>
<td>PULL</td>
<td>Pull register</td>
</tr>
<tr>
<td>1101</td>
<td>TOSP</td>
<td>Copy to Stack pointer</td>
</tr>
<tr>
<td>1110</td>
<td>FRMSP</td>
<td>Copy from Stack pointer</td>
</tr>
<tr>
<td>1111</td>
<td>Unused</td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Op codes for single register operations

In these operations the 8 bit (signed) number specified by YYYYYYYY is added to the register DD depending on the condition specified in XXX shown in the table. Not much room left for more instructions!

3.5 Other Instructions

These instructions are identified by a one in bit 14 and a one in bit 11 and zero in bit 13.
Specifications for a simple computer

**Table 12: Op codes for other instructions**

<table>
<thead>
<tr>
<th>XXX</th>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>CLC</td>
<td>Clear Carry</td>
</tr>
<tr>
<td>001</td>
<td>SEC</td>
<td>Set Carry</td>
</tr>
<tr>
<td>010</td>
<td>CLV</td>
<td>Clear Overflow</td>
</tr>
<tr>
<td>011</td>
<td>SEV</td>
<td>Set Overflow</td>
</tr>
<tr>
<td>100</td>
<td>RTS</td>
<td>Return from subroutine</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 11: Other instructions**

<table>
<thead>
<tr>
<th>XXX</th>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>CLC</td>
<td>Clear Carry</td>
</tr>
<tr>
<td>001</td>
<td>SEC</td>
<td>Set Carry</td>
</tr>
<tr>
<td>010</td>
<td>CLV</td>
<td>Clear Overflow</td>
</tr>
<tr>
<td>011</td>
<td>SEV</td>
<td>Set Overflow</td>
</tr>
<tr>
<td>100</td>
<td>RTS</td>
<td>Return from subroutine</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.6 **I/O Instructions**

These instructions are identified by a one in bit 14 and a one in bits 11 and 13. Bit 12 specifies the destination register.

**Table 13: I/O instructions**

<table>
<thead>
<tr>
<th>XXX</th>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>INP</td>
<td>Move input char to register</td>
</tr>
<tr>
<td>001</td>
<td>OUT</td>
<td>Move register to output</td>
</tr>
<tr>
<td>010</td>
<td>HLT</td>
<td>Halt the computer</td>
</tr>
<tr>
<td>011</td>
<td>IFLG</td>
<td>Copy IFLG to V Flag</td>
</tr>
<tr>
<td>100</td>
<td>OFLG</td>
<td>Copy OFLG to V Flag</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.7 **OOPS we goofed instructions**

If we go through the instruction set, we have no space for JSR (jump to subroutine). Since we rarely explicitly save the program counter, (R3), we shall forego the SV R3 instruction and replace it with jump to subroutine instruction. Thus we modify the register memory table as follows:

**Table 15: Register/Memory Instructions. In ST command RR can not be 11**

<table>
<thead>
<tr>
<th>LD</th>
<th>0 0</th>
<th>R</th>
<th>R</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>0 1</td>
<td>R</td>
<td>R</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>JSR</td>
<td>0 1</td>
<td>1</td>
<td>1</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>
4. The instruction register organization

The instruction register is the brain inside the CPU that determines what actions take place. The instruction register would be connected to a number of decoders as shown in Figure 1. Note the various decoder lines that come from the IR register.

![Figure 1: Instruction Register and Decoders](image)

5. ALU organization

We can design the machine so that each register has its own ALU or they all share a common ALU. To keep the organization simple, we shall have a single ALU shared by all the registers. The ALU is organized as shown in figure 2. This is organization is similar to what is in the text book except, that the X input to the ALU is multiplexed. Note that the output of ALU is connected to all the registers. The output will be clocked into a register if its LD line is activated.

Notes on ALU: The ALU has internal encoders/multiplexers that will select the desired function. Also, the transfer command will transfer the Y input to the output.
6. Bus Architecture

All inter register and register memory transfers are performed through the bus with the exception of R0-R3. Input to these registers come from the ALU. The IR is connected to the bus at two addresses. At the first address, all the 16 bits are connected to the bus. At the second address, only bits 0-7 are connected to the bus. Bit 7 is also connected to the bits 8-15 of the bus. Thus, we can either load the entire IR or only the lower byte of IR to the bus.

7. Instruction Cycle

First two steps in instruction cycle are the same for all instructions:

- **T0:** \( AR \leftarrow PC \) (only 12 bits transferred)
- **T1:** \( IR \leftarrow M[AR], PC\leftarrow PC+1 \)

Subsequent cycles depend on the instruction that has to be executed. We will consider each of them separately without assigning a specific clock cycle.
7.1 Instructions involving a register and memory address

7.1.1 Load and store instructions

For both these instructions, we need to perform the following

- \( AR \leftarrow IR \)
- \( BUS \leftarrow \text{desired data (Either from register (ST) or from memory (LD))} \)
- \( M[AR]\leftarrow BUS \) (ST) or \( Rk \leftarrow BUS \) (LD)

7.1.2 JSR instruction

- \( SP\leftarrow SP-1 \)
- \( AR\leftarrow SP; \)
- \( M[AR]\leftarrow PC; \)
- \( PC\leftarrow IR \) (note only 12 bits will be transferred)

7.2 Instructions involving two registers

There are two steps involved. First, the data has to show up on the bus. Next, the appropriate action has to take place.

7.2.1 Mode 0

- \( BUS = SRC \)

7.2.2 Mode 1

- \( AR \leftarrow SRC \)
- \( BUS = M[AR] \)

7.2.3 Mode 2

- \( SRC \leftarrow SRC - 1 \)
- \( AR \leftarrow SRC \)
- \( BUS = M[AR] \)

7.2.4 Mode 3

- \( AR \leftarrow SRC \)
- \( BUS = M[AR] \)
- \( SRC\leftarrow SRC+1 \)

Depending on the opcode the following ALU select lines are set. In what follows Cx is the output of the opcode decoder (see figure 1).

<table>
<thead>
<tr>
<th>C0:</th>
<th>TR=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1:</td>
<td>SUB=1</td>
</tr>
<tr>
<td>C2:</td>
<td>ADD=1</td>
</tr>
<tr>
<td>C3:</td>
<td>ADC=1</td>
</tr>
<tr>
<td>C4:</td>
<td>SUB=1</td>
</tr>
<tr>
<td>C5:</td>
<td>SBC=1</td>
</tr>
<tr>
<td>C6:</td>
<td>AND=1</td>
</tr>
<tr>
<td>C7:</td>
<td>OR=1</td>
</tr>
<tr>
<td>C8:</td>
<td>EOR=1</td>
</tr>
</tbody>
</table>

7.3 Instructions involving a single register

Most of the work will be performed either internally in the register (CLR/INC/DEC) or through the ALU. Note TST command is same as TR
command except the output of the ALU is not clocked into any register but only the condition flags are set. Only instructions that require some explanation are the PUSH and PULL instructions.

```
PUSH:    SP<=SP-1
         AR<=SP
         BUS = REG
         M[AR]<=BUS

PULL:    AR<=SP
          BUS=M[AR]
          REG<=BUS
          SP=SP+1
```

7.4 Single Register Offset instructions

If the required condition is met, then this instruction consists of two parts

```
BUS = IR(0-7)
DST <= DST + BUS
```

Note: The condition flags should NOT be set after performing the addition!

7.5 Other instruction

Most of the instructions are obvious except for RTS. This has to undo the JSR command as follows:

```
AR<=SP
PC<=M[AR]
SP<=SP+1
```

8. Control and Decoder lines

The working of the computer is controlled by turning on/off the appropriate control lines at the appropriate moment in the instruction cycle. The following table lists all the control lines for the machine.

8.1 Control Lines for bus control

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS_IR</td>
<td>Connects IR to Bus</td>
<td>BUS_R0</td>
<td>Connects R0 to Bus</td>
</tr>
<tr>
<td>BUS_AR</td>
<td>Connects AR to Bus</td>
<td>BUS_R1</td>
<td>Connects R1 to Bus</td>
</tr>
<tr>
<td>BUS_M</td>
<td>Connects memory to Bus</td>
<td>BUS_R2</td>
<td>Connects R2 to Bus</td>
</tr>
<tr>
<td>BUS_SP</td>
<td>Connects SP to Bus</td>
<td>BUS_R3</td>
<td>Connects R3 to Bus</td>
</tr>
<tr>
<td>BUS_INP</td>
<td>Connects INPR to Bus</td>
<td>BUS_TR</td>
<td>Connects TR to Bus</td>
</tr>
</tbody>
</table>

Table 16: Control lines for bus control

8.2 Control Lines for ALU Multiplexor

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR0</td>
<td>Multiplex R0 to ALU-X</td>
<td>SR2</td>
<td>Multiplex R2 to ALU-X</td>
</tr>
<tr>
<td>SR1</td>
<td>Multiplex R1 to ALU-X</td>
<td>SR3</td>
<td>Multiplex R3 to ALU-X</td>
</tr>
<tr>
<td>SSP</td>
<td>Multiplex SP to ALU-X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 17: Control lines for ALU Multiplexer.
8.3 Control Lines for ALU control

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU_ADD</td>
<td>Select ADD functionality</td>
<td>ALU_ADD</td>
<td>Select ADC functionality</td>
</tr>
<tr>
<td>ALU_SUB</td>
<td>Select ADD functionality</td>
<td>ALU_SBC</td>
<td>Select ADC functionality</td>
</tr>
<tr>
<td>CLOCK_CSNZ</td>
<td>Enable Clocking result of operations to Flags (C, S, N, Z)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table 18: Control lines for ALU control. Only some are indicated. The others are similar*

8.4 Control Lines for registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD(R0)</td>
<td>LD line or R0</td>
<td>CLR(R0)</td>
<td>Clear line for R0</td>
</tr>
<tr>
<td>INC(R0)</td>
<td>Increment line for R0</td>
<td>DEC(R0)</td>
<td>Decrement line for R0</td>
</tr>
</tbody>
</table>

*Table 19: Control lines for registers. Only R0 is shown. Other registers each have four such lines*

8.5 Decoder Lines

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>I15</td>
<td>Bit 15 of IR</td>
<td>I14</td>
<td>Bit 14 of IR</td>
</tr>
<tr>
<td>I7</td>
<td>Bit 7 of IR</td>
<td>D0,D1,D2,D3</td>
<td>Decode of bits 12-13 of IR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S0,S1,S2,S3</td>
<td>Decode of bits 0-1 of IR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M0,M1,M2,M3</td>
<td>Decode of bits 2-3 of IR</td>
</tr>
</tbody>
</table>

*Table 20: Decoder lines*

9. Timing

Timing of most events are obvious. The following is a partially complete list of timing. Note that clock cycles T0 and T1 are used to fetch the instruction from memory.

9.1 Instructions involving a register and memory address

We can use the following Boolean expressions to identify the commands:

- LDINST = NOT(I15) && NOT(I14)
- JSRINST = NOT(I15) && I14 && I13 && I12
- STINST = NOT(I15) && I14 && NOT(JSRINST)
The accompanying table provides event timing for these instructions. Each event has the RTL statement at the top followed by the logic events that have to occur. From the logic events, we can deduce the gates that or needed to implement the control. For example, LD(AR) can be written as

$$LD(AR) = (T3&&LDINST) || (T3&&STINST) || (T4&&JSRINST) \ldots$$

Here the three dots represent other LD(AR) instructions. Thus we can collect all the events in the table and arrive at the following set of logic implementations

$$LD(AR) = (T3&&LDINST) || (T3&&STINST) || (T4&&JSRINST) \ldots$$
$$BUS_{IR} = (T3&&LDINST) || (T3 && STINST) || (T6&&JSRINST) || \ldots$$
$$BUS_{M} = (T4&&LDINST) || \ldots$$
$$BUS_{SP} = (T4 && JSRINST) || \ldots$$
$$Read = (T4&&LDINST) || \ldots$$
$$LD(R0) = (T4&&LDINST && & D0) || \ldots$$
$$LD(R1) = (T4&&LDINST && & D1) || \ldots$$
$$LD(R2) = (T4&&LDINST && & D2) || \ldots$$
$$LD(R3) = (T4&&LDINST && & D3) || (T6 && JSRINST)|| \ldots$$
$$Write = (T4 & STINST) || (T5 & & JSRINST) || \ldots$$
$$BUS_{R0} = (T4 & STINST && & D0) || \ldots$$
$$BUS_{R1} = (T4 & STINST && & D1) || \ldots$$
$$BUS_{R2} = (T4 & STINST && & D2) || \ldots$$
$$BUS_{R3} = (T4 & STINST && & D3)|| (T5 & & JSRINST) || \ldots$$
$$DEC(SP) = (T3 & & JSRINST)$$
$$CLR(SC) = (T4 && LDINST) || (T4 && STINST) || (T6 & & JSRINST) ||\ldots$$

<table>
<thead>
<tr>
<th>Cycle</th>
<th>LD</th>
<th>ST</th>
<th>JSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>AR&lt;=IR</td>
<td>AR&lt;=IR</td>
<td>SP&lt;=SP-1</td>
</tr>
<tr>
<td></td>
<td>BUS=IR</td>
<td>BUS=IR</td>
<td>DEC(SP)=1</td>
</tr>
<tr>
<td></td>
<td>LD(AR)=1</td>
<td>LD(AR)=1</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>Ri&lt;=M[AR]</td>
<td>M[AR]&lt;=Ri</td>
<td>AR&lt;=SP</td>
</tr>
<tr>
<td></td>
<td>SC&lt;=0</td>
<td>SC&lt;=0</td>
<td>BUS=SP</td>
</tr>
<tr>
<td></td>
<td>Bus=Mem</td>
<td>Bus=R(Di)</td>
<td>BUS=SP</td>
</tr>
<tr>
<td></td>
<td>Read=1</td>
<td>Write=1</td>
<td>LD(AR)=1</td>
</tr>
<tr>
<td></td>
<td>LD(Di)=1</td>
<td>CLR(SC)=1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLR(SC)=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>M[AR]&lt;=R3</td>
<td>BUS=R3</td>
<td>R3&lt;=IR</td>
</tr>
<tr>
<td></td>
<td>BUS=Mem</td>
<td>Write=1</td>
<td>SC&lt;=0</td>
</tr>
<tr>
<td></td>
<td>Read=1</td>
<td></td>
<td>BUS=IR</td>
</tr>
<tr>
<td></td>
<td>LD(R3)=1</td>
<td></td>
<td>LD(R3)=1</td>
</tr>
<tr>
<td></td>
<td>CLR(SC)=1</td>
<td></td>
<td>CLR(SC)=1</td>
</tr>
</tbody>
</table>

Table 21: Event Timing for Register/memory instructions
9.1.1 Exercise

Using the above translation as a model, determine the logic for implementing

T0: \( \text{AR} \leftarrow \text{PC} \) (only 12 bits transferred)
T1: \( \text{IR} \leftarrow \text{M}[\text{AR}], \text{PC} \leftarrow \text{PC}+1 \)

9.2 Instructions involving two registers

Depending on the addressing modes we have the following timing sequences. However, you should note the following:

The CMP command is same as SUB command except that the result is not loaded to any register. Mode1 and Mode3 are the same except that in mode 3, we increment the source register at the end. Mode2 is same as mode 3 except we decrement the source first.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Mode 0</th>
<th>Mode 1</th>
<th>Mode 2</th>
<th>Mode 3</th>
</tr>
</thead>
</table>
| T2    | \( D \leftarrow D \text{ op } S \)  
SC\leftarrow 0  
Bus = S  
LD_D=1  
ALU_OP=1  
CLOCK_CSNZ=1  
CLR_SC=1 | AR\leftarrow S  
BUS=S  
LD_AR=1 | S\leftarrow S-1  
DEC(S)=1 | AR\leftarrow S  
BUS=S  
LD_AR=1 |
| T3    | D\leftarrow D \text{ op } MEM  
Bus = MEM  
LD_D=1  
ALU_OP=1  
CLOCK_CSNZ=1  
CLR_SC=1 | AR\leftarrow S  
BUS=S  
LD_AR=1 | D\leftarrow D \text{ op } MEM  
IS\leftarrow S+1  
Bus = MEM  
LD_D=1  
ALU_OP=1  
CLOCK_CSNZ=1  
CLR_SC=1  
INC(SC)=1 |
| T4    | D\leftarrow D \text{ op } MEM  
Bus = MEM  
LD_D=1  
ALU_OP=1  
CLOCK_CSNZ=1  
CLR_SC=1 | D\leftarrow D \text{ op } MEM  
Bus = MEM  
LD_D=1  
ALU_OP=1  
CLOCK_CSNZ=1  
CLR_SC=1 |

Table 22: Timing for REG-REG operations. See notes

10. Simulating the computer

10.1 Overview

In this section, we study what it takes to simulate the system. The simulation is a while loop, which terminates only when the ONFLG becomes zero (due to a HLT) instruction. To accurately simulate the machine, define a variable (type unsigned int)
for each control line and for each decode line. Define a variable (type unsigned int) for each register, and a variable for the intermediate values, BUS, ALU_X and ALU_Y. Also define a variable (type char) for each 8 bit quantity (last 8 bits of IR, INPR and OUTR). For each flop-flop, define a variable (type int) and define two input lines (J and K). For example, a partial declaration for the machine would be:

```c
unsigned int R0, LD_R0, CLR_R0, INC_R0, DEC_R0;
unsigned int R1, LD_R1, CLR_R1, INC_R1, DEC_R1;
unsigned int R2, LD_R2, CLR_R2, INC_R2, DEC_R2;
unsigned int R3, LD_R3, CLR_R3, INC_R3, DEC_R3;
unsigned int SP, LD_SP, CLR_SP, INC_SP, DEC_SP;
unsigned int BUS_R0, BUS_R1, BUS_R2, BUS_R3, BUS_SP;
```

Note that there is a pattern to these declarations. Unless you are an excellent typist, you are better off using the C pre-processor to do the typing for you. It will be consistent, and hopefully error free. So somewhere at the top of your program (preferably in an include file) you may define the macro

```c
#define REG(X) unsigned int X, LD_##X, CLR_##X, INC_##X, DEC_##X, BUS_##X, S##X
```

Now to declare registers R0, R1, R2, R3, SP, TR, IR, SC, AR all you need is the following line

```c
REG(R0); REG(R1); REG(R2); REG(R3); REG(SP); REG(TR); REG(IR); REG(SC); REG(AR);
```

Similarly, you can use the macro FLIPFLOP(X) to declare flip-flops as:

```c
#define FLIPFLOP(X) int X, J_##X, K_##X
FLIPFLOP(C); FLIPFLOP(N); FLIPFLOP(V); FLIPFLOP(Z); FLIPFLOP(ONFLG);
FLIPFLOP(IFLG); FLIPFLOP(OFLG);
```

Next declare the decoder lines:

```c
int I15, I14, I7, D0, D1, D2, D3, S0, S1, S2, S3, M0, M1, M2, M3;
int C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13, C14, C15;
```

It will also be convenient to define a set of macros to define the various bit positions as

```c
#define BIT0 1
#define BIT1 (1<<1)
#define BIT2 (1<<2)
```

etc. Also, it may be convenient define a few intermediate variables to replace repeated logical expressions.

The main simulation code, except for declarations will be as follows:

```c
while(ONFLG) {
    /* Code to compute all the decoder lines */
    /* Code to compute all the control signal lines */
    /* Code to compute the bus value */
    /* Code to compute the inputs to ALU and output of ALU (ALU_X and ALU_Y) */
    /* Code to compute control signals for ALU flip-flops*/
```
We will take each of these in turn.

10.2 Code to compute all the decoder lines

To compute the decoder lines, we have to break up the IR into its component parts and extract the terms that are connected to decoders. This is accomplished by the following C statements:

\[
\begin{align*}
SS &= IR \& (BIT0\|BIT1) /* extract bits 0\&1*/ \\
MM &= (IR >> 2) \& (BIT0\|BIT1); /* Shift IR right twice and extract bits 0\&1*/ \\
CC &= (IR >>8) \& (BIT0\|BIT1\|BIT2\|BIT3); /* Shift IR right 8 times and extract bits 0-3*/ \\
DD &= (IR >>12) \& (BIT0 \| BIT1); /* Shift IR right 12 times and extract bits 0\&1*/
\end{align*}
\]

**Warning:** We are working at the bit level and should use \& and |, not && and ||.

Now that we have extracted the parts of IR, we can write the code for the decoder lines as follows:

\[
\begin{align*}
I15 &= IR \& BIT15; I14=IR\&BIT14; I7=IR\&BIT7; \\
D0=(DD==0); D1 = (DD==1); D2=(DD==2); D3 = (DD==3); \\
M0=(MM==0); M1 = (MM==1); M2=(MM==2); M3 = (MM==3); \\
S0=(SS==0); S1 = (SS==1); S2=(SS==2); S3 = (SS==3); \\
C0=(CC==0); C1 = (CC==1); C2=(CC==2); C3 = (CC==3); \\
C4=(CC==4); C5 = (CC==5); C6=(CC==6); C7 = (CC==7); \\
C8=(CC==8); C9 = (CC==9); C10=(CC==10); C11 = (CC==11); \\
C12=(CC==12); C13 = (CC==13); C14=(CC==14); C15 = (CC==15);
\end{align*}
\]

10.3 Code to compute all the control lines

This is the most tedious, error prone and down right boring part of building the simulator. A typical line of code will look something like and you have to make sure that there are absolutely no mistakes what-so-ever!

\[
\begin{align*}
xxx &= (aaa \&\& bbb) || (ccc \&\& ddd) || (eee \&\& fff) || ...
\end{align*}
\]

To make matters worse, each part of the terms between || will come from different parts of your computer specifications! If you value your sanity, you should probably write this code as

\[
\begin{align*}
xxx &= 0; \\
xxx &= xxx || (aaa \&\& bbb); \\
xxx &= xxx || (ccc \&\& ddd); \\
xxx &= xxx || (eee \&\& fff); \\
xxx &= xxx || ...
\end{align*}
\]

**Warning:** do not forget the first \texttt{xxx = 0}; statement! Again, it is better to define two macros to do most of the work for you:
10.4 Code to compute the bus value

The code to compute the bus value is a big if-else if statement. Only thing we have to take care of is the case when only the last 8 bits of IR is to be transferred. We declare a variable called IR8 (type char) and a bus control line BUS_IR8. Typical code would be:

```
IR8 = IR & 0XFF; /* Peel off last 8 bits of IR */

if (BUS_R0) bus = R0;
else if (BUS_R1) bus = R1;
else if (BUS_R2) bus = R2;
.....
else if (BUS_IR8) bus = IR8;
else bus = 0xFFFF; /* No one gets the bus */
```

10.5 Code to compute the ALU output

ALU is also implemented as a big if-else-if statement. Only thing we have to make sure is that the inputs to ALU are computed after the value of the bus has been determined. Secondly, we need to do arithmetic using 17 bits (so we can set the carry and overflow flags) which is an awkward number of bits. So we use 32 bit arithmetic and then truncate the answer.

```
if (SR0) ALU_X = R0;
else if (SR1) ALU_X=R1;
else if (SR2) ALU_X = R2;
else if (SR3) ALU_X = R3;
else if (SSP) ALU_X = SP;
else ALU_X = 0xFFFF; /* No one connected to ALU_X */
ALU_Y = bus;

if (ALU_ADD) {
  /* Need to do 32
  long int x;
  long int y;
  long int z;
  int zhigh;
  z = x+y;
  zhigh = (z >> 16) & 0xFFFF; /* Shift out the lower 16 bits */
  ALU_OUT = z & 0xFFFF; /* Keep only 16 bits */
  carry_out = (z>>1) & BIT15;
  neg_out = ALU_OUT & BIT15;
  zero_out = (ALU_OUT == 0);
  ov_out = NOT((zhigh == 0 ) || (zhigh == 0XFFFF));
}
else if (ALU_SUB) {
  /* Same code as ALU_ADD except use '-' instead of '+'*/
}
else if (ALU_TR) {
  ALU_OUT = ALU_Y & 0xFFFF;
  carry_out = 0
  neg_out = ALU_OUT & BIT15;
```
10.6 Code to compute control signals for ALU flip-flops

The code to clock ALU flip-flops are all the similar. We just make sure that the J/K lines are set only if \textsc{CLOCK\_CSNZ} is true.

\begin{center}
\begin{tabular}{l}
\textsc{J\_C} = \textsc{CLOCK\_CSNZ} && \textsc{carry\_out} ;& \textsc{K\_C} = \textsc{CLOCK\_CSNZ} \& \& \textsc{NOT(carry\_out)}; \\
\textsc{J\_Z} = \textsc{CLOCK\_CSNZ} \& \& \textsc{zero\_out} ;& \textsc{K\_Z} = \textsc{CLOCK\_CSNZ} \& \& \textsc{NOT(zero\_out)}; \\
\textsc{J\_N} = \textsc{CLOCK\_CSNZ} \& \& \textsc{neg\_out} ;& \textsc{K\_N} = \textsc{CLOCK\_CSNZ} \& \& \textsc{NOT(neg\_out)}; \\
\textsc{J\_V} = \textsc{CLOCK\_CSNZ} \& \& \textsc{ov\_out} ;& \textsc{K\_V} = \textsc{CLOCK\_CSNZ} \& \& \textsc{NOT(ov\_out)}; \\
\end{tabular}
\end{center}

10.7 Code to clock the registers

The code to clock the registers are all the same. The code would be something like:

\begin{center}
\begin{tabular}{l}
if (\textsc{LD\_R0}) \textsc{R0} = \textsc{ALU\_OUT} ; \\
else if (\textsc{CLR\_R0}) \textsc{R0} = 0 ; \\
else if (\textsc{INC\_R0}) \textsc{R0} = \textsc{R0}+1 ; \\
else if (\textsc{DEC\_R0}) \textsc{R0}=\textsc{R0}-1 ; \\

if (\textsc{LD\_R1}) \textsc{R1} = \textsc{ALU\_OUT} ; \\
else if (\textsc{CLR\_R1}) \textsc{R1} = 0 ; \\
else if (\textsc{INC\_R1}) \textsc{R1} = \textsc{R1}+1 ; \\
else if (\textsc{DEC\_R1}) \textsc{R1}=\textsc{R1}-1 ; \\

/* And so on for R2, R3 */ \\
if (\textsc{LD\_AR}) \textsc{AR} = \textsc{BUS} ; \\
else if (\textsc{CLR\_AR}) \textsc{AR} = 0 ; \\
else if (\textsc{INC\_AR}) \textsc{AR} = \textsc{AR}+1 ; \\
else if (\textsc{DEC\_AR}) \textsc{AR} = \textsc{AR}-1 ; \\

/* and so on for other registers connected to the bus ... */
\end{tabular}
\end{center}

The above code is again a candidate for the use of a macro. The following macro will make life easier all around! The macro does not fit in one line, so we use \ to continue on to the next line.

\begin{center}
\texttt{#define CLOCK(dst, src)  \\
    if (\textsc{LD\_##dst}) \texttt{dst} = src; \ \texttt{\} \\
    else if (\textsc{CLR\_##dst}) \texttt{dst} = 0; \ \\
\end{center}
10.8 Code to clock the flip-flops

The code to clock the flip-flops is similar to that of the registers. Typical code would be

```c
if (J_C && K_C) C = NOT(C); else if (J_C) C = 1; else if (K_C) C = 0;
if (J_N && K_N) N = NOT(N); else if (J_N) N = 1; else if (K_N) N = 0;
/* and so on. */
```

Again a macro will make the code less error prone!

11. A minimal simulation code

This is a minimal code to test the computer and the simulator. THIS CODE IS NOT COMPLETE!

```c
Error! Cannot open file.
```