Design Notes of Microprocessor $\mu311.1$

LECTURE NOTES - CENG311 COMPUTER ARCHITECTURE

Tolga Ayav
tolgaayav@iyte.edu.tr

October 1, 2018

Technical Report
Department of Computer Engineering
İzmir Institute of Technology

35430 Urla İzmir, Turkey. Web: http://compeng.iyte.edu.tr

ISSN:

All rights, including translation into other languages are reserved by the authors. No part of this report may be reproduced or used in any form or by any means - graphically or mechanically, including photocopying, recording, taping or information and retrieval systems - without written permission from the authors.
Contents

1 External Parts 3
   1.1 Reset Circuit ................................................. 3
   1.2 Clock Circuit ............................................... 3
   1.3 1024x16-bit ROM ............................................ 4
   1.4 1024x16-bit RAM ............................................. 6

2 Microprocessor \( \mu 311.1 \) 7
   2.1 Instruction Set .............................................. 8
   2.2 Datapath ..................................................... 15
      2.2.1 Registers ................................................ 16
      2.2.2 Program Counter ....................................... 16
      2.2.3 Instruction Register, Stack Pointer .................. 18
      2.2.4 Register File .......................................... 18
      2.2.5 Multiplexers ........................................... 20
      2.2.6 Buffers .................................................. 21
      2.2.7 ALU and Shifter ........................................ 22
      2.2.8 The processor ......................................... 32
   2.3 Stack ....................................................... 33
   2.4 Control Unit ............................................... 33
      2.4.1 Bus Cycles ............................................. 35

3 Testbench 47

4 Address Decoding and I/O Communication 48

5 Interrupts 49

6 Additional Instructions and Units 50
   6.1 Watchdog Timer ............................................. 50
   6.2 Base Pointer Register ..................................... 50

7 Programming 52
   7.1 High-Level Programming .................................. 52
   7.2 Assembly and Linking ..................................... 53
   7.3 Sample Programs .......................................... 54
   7.4 Assemblers ................................................. 55
   7.5 Linking ..................................................... 56

8 Instruction Pipelining 63

A Simulation in ModelSim PE Student Edition 67

B BNF Syntax for VHDL 68

C Implementation Hierarchy 76
D  *as311* Assembler

E  Multitasking in \( \mu 311.1 \)
   E.1  16-bit timer .......................................................... 80

F  \( \mu 311.1 \) Internal Schematic 83
Preface

This handbook includes a part of the lecture notes of CENG 311 Computer Architecture course given in the undergraduate program of the Department of Computer Engineering at Izmir Institute of Technology.

One aim of this course is to introduce the preliminaries of a general purpose microprocessor design. To this end, I aim to teach a very simple microprocessor which we call $\mu$311.1, an 16-bit processor with only 25 instructions.

This document is intended to help the students with their laboratory works. In the experimental part of the course, students are expected to implement this or another similar processor using VHDL in order to attain a sufficient knowledge and intuition about “What is really happening inside a computer system?”.

In other words, starting from typing `printf("value:%d",*p);` they must understand compiling, assembling, linking, loading the machine code and how processors execute this code. This document aims to give a very short and abstract answer to the above question.

Students may find many parts missing, too short or incomplete. Nonetheless, I hope that this will be a good starting point for their deeper research as well as their study of computer architecture.
Figure 1: \( \mu311.1 \) internal diagram
1 External Parts

1.1 Reset Circuit

μ311.1 needs an external reset circuit as given in Figure 2. The reset signal is used to restart the microprocessor properly. Depending on the design, for a proper reset, this signal must be given to the processor for a certain period of time.

![Reset circuit diagram]

**Figure 2: Reset circuit.**

```vhdl
-- reset.vhd: Reset circuit
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity rst_gen is port (reset : out std_logic);
end rst_gen;

architecture Behavioral of rst_gen is
constant rst_period : time := 100 us;
reset <= '1' after 0 us, '0' after rst_period;
end Behavioral;
```

1.2 Clock Circuit

```vhdl
-- clock.vhd: Clock signal generator
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```
Figure 3: Clock generator using digital inverter. For $C_1 = 1nF$ and $R_1 = 1k\Omega$, $f = 1MHz$.

```vhdl
entity clk_gen is port (clk : out std_logic);
end clk_gen;

architecture Behavioral of clk_gen is
constant clk_period : time := 1 us;
clk_process : process
begin
    clk <= '0';
    wait for clk_period/2;  --for 0.5 us signal is '0'.
    clk <= '1';
    wait for clk_period/2;  --for next 0.5 us signal is '1'.
end process;
end Behavioral;
```

**Question 1** Discuss about “Synthesizable VHDL code”. Are “clock.vhd” and “reset.vhd” synthesizable codes?

### 1.3 1024x16-bit ROM

```vhdl
-- rom1024.vhd: 1024x16bit ROM
library ieee;
library work;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use work.u311.all;
use work.opcodes.all;

entity rom1024 is port(
cs : in std_logic;
oe : in std_logic;
addr : in std_logic_vector (9 downto 0);
data : out std_logic_vector (15 downto 0))
```
architecture imp of rom1024 is

subtype cell is std_logic_vector(15 downto 0);
type rom_type is array(0 to 24) of cell;

-- Our program stored in the memory
constant ROM : rom_type :=(
    X"b0ff", -- movi a stack
    X"b800", -- mov sp a
    X"136c", -- sub d d d
    X"0460", -- mov e d
    X"b208", -- movi c size
    X"580f", -- jmp _main
    X"0000",
    X"0000",
    X"0000",
    X"0000",
    X"0000",
    X"0000",
    X"0000",
    X"0000",
    X"a070", -- L write d e
    X"3360", -- inc d
    X"3a40", -- dec c
    X"6c10", -- jnz L
    X"7800", -- ret
    X"0760", -- _main mov h d
    X"b6aa", -- movi g OzAA
    X"a0bc", -- write g h
    X"5c03", -- jmp _main
    X"8800" -- halt
);

begin
    process(cs, oe, addr)
    begin
        if (cs='0' and oe='1') then
            data <= ROM(conv_integer(addr));
        else data <= (others=>'Z');
        end if;
    end process;
end imp;
We have an assembler, namely \texttt{as311}, to translate the assembly programs to the machine code of $\mu311.1$. The assembler generates a special output file with .vhdl_hex extension. It can be copied and pasted to the appropriate place in the rom1024.vhd.

1.4 1024x16-bit RAM

Stack operations require a volatile memory. An implementation of 1024x16 bit RAM is as follows:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity ram1024 is port(
    rst: in std_logic;
    cs: in std_logic; --chip select
    wr: in std_logic; --write enable
    rd: in std_logic;--read enable
    addr: in std_logic_vector(9 downto 0);
    data: inout std_logic_vector(15 downto 0));
end ram1024;

architecture imp of ram1024 is
    subtype cell is std_logic_vector(15 downto 0);
    type ram_type is array(0 to 1023) of cell;
    signal RAM: ram_type;
begin
    process(cs,wr,rd,addr)
    begin
        if ( cs='0' and rd='1') then
            data <= RAM(conv_integer(addr));
        elsif( cs='0' and wr='1') then
            RAM(conv_integer(addr)) <= data after 10ns ;
        else data <= (others=>'Z');
        end if;
    end process;
end imp;
```
2 Microprocessor \( \mu 311.1 \)

The general specifications of \( \mu 311.1 \) are:

- 16-bit processor
- 39 pins
- Addresses up to 64k locations
- No internal program memory
- 8x16-bit general purpose registers
- interrupt mechanism (supports 8 external interrupts)
- 4 cycles: opcode fetch, read memory-I/O, write memory-I/O and interrupt cycles.
- 25 single-word instructions with single cycle operation.

Figure 4 shows a general diagram of \( \mu 311.1 \). \( \mu 311.1 \) is a simple 16-bit processor. It has the following inputs/outputs:

- \textbf{clk} is clock signal that is needed by the microprocessor.
- \textbf{reset} restarts the microprocessor.
- \textbf{int} is the hardware interrupt signal that is used for event triggering.
- \textbf{inta} is the acknowledge of \( \mu 311.1 \) as a response to the interrupt request of an external device.
- \textbf{address bus} is an 16-bit bus that is used for the communication with external memory and I/O devices. It can address up to 64k locations.
- \textbf{data bus} is an 16-bit bus that is used for the data transfer between external devices and \( \mu 311.1 \).
wr indicates a write cycle.

rd indicates a read cycle.

opfetch indicates an opcode fetch cycle.

All control signals of $\mu 311.1$ (wr, rd, reset, int, inta, opfetch) are active high. This means that wr=1 indicates a write cycle and the microprocessor is reset when reset=1.

![Figure 5: The two 64k memory maps of $\mu 311.1$.](image)

**Question 2** Write a simulator in Java for $\mu 311.1$. Your simulator should take an assembly program as input and execute it. During the simulation, registers and other critical values will be shown on the screen.

**2.1 Instruction Set**

$\mu 311.1$’s limited instruction set has only 25 instructions. These commands are given in Table 1. To encode 25 instructions, the operation code (opcode) requires 5 bits, giving us 32 different combinations. As shown in the encoding column, the five most significant bits represent the opcode of the instructions. For example, the opcode for **mov** is 00000 and the opcode for **movi** is 10111 and so on.
Table 1: Instruction set of \( \mu 311.1 \). Each instruction is 16-bit long.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Encoding</th>
<th>Operation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>mov R1, R2</td>
<td>00000 ( r_1 r_1 r_2 x x uu )</td>
<td>R1 ( \leftarrow ) R2</td>
<td>move register</td>
</tr>
<tr>
<td>00001</td>
<td>add R1, R2, R3</td>
<td>00001 ( r_1 r_1 r_1 r_2 r_2 r_3 x x uu )</td>
<td>R1 ( \leftarrow ) R2 + R3</td>
<td>addition</td>
</tr>
<tr>
<td>00010</td>
<td>sub R1, R2, R3</td>
<td>00010 ( r_1 r_1 r_1 r_2 r_2 r_3 x x uu )</td>
<td>R1 ( \leftarrow ) R2 - R3</td>
<td>subtraction</td>
</tr>
<tr>
<td>00011</td>
<td>and R1, R2, R3</td>
<td>00011 ( r_1 r_1 r_1 r_2 r_2 r_3 x x uu )</td>
<td>R1 ( \leftarrow ) R2 &amp; R3</td>
<td>logical and</td>
</tr>
<tr>
<td>00100</td>
<td>or R1, R2, R3</td>
<td>00100 ( r_1 r_1 r_1 r_2 r_2 r_3 x x uu )</td>
<td>R1 ( \leftarrow ) R2 | R3</td>
<td>logical or</td>
</tr>
<tr>
<td>00101</td>
<td>not R</td>
<td>00101 ( rrr rrr x x x uu )</td>
<td>R ( \leftarrow ) not R</td>
<td>logical not</td>
</tr>
<tr>
<td>00110</td>
<td>inc R</td>
<td>00110 ( rrr rrr x x x uu )</td>
<td>R ( \leftarrow ) R+1</td>
<td>increment</td>
</tr>
<tr>
<td>00111</td>
<td>dec R</td>
<td>00111 ( rrr rrr x x x uu )</td>
<td>R ( \leftarrow ) R-1</td>
<td>decrement</td>
</tr>
<tr>
<td>01000</td>
<td>sr R</td>
<td>01000 ( rrr rrr x x x uu )</td>
<td>R ( \leftarrow ) R &gt;&gt; 1</td>
<td>shift right</td>
</tr>
<tr>
<td>01001</td>
<td>sl R</td>
<td>01001 ( rrr rrr x x x uu )</td>
<td>R ( \leftarrow ) R &lt;&lt; 1</td>
<td>shift left</td>
</tr>
<tr>
<td>01010</td>
<td>rr R</td>
<td>01010 ( rrr rrr x x x uu )</td>
<td>R_{15} ( \leftarrow ) R_{6}; R ( \leftarrow ) R &gt;&gt; 1</td>
<td>shift right</td>
</tr>
<tr>
<td>01011</td>
<td>jump add11</td>
<td>01011 ( aaaaaaaaaa )</td>
<td>PC ( \leftarrow ) PC + add11</td>
<td>jump</td>
</tr>
<tr>
<td>01100</td>
<td>jz add11</td>
<td>01100 ( aaaaaaaaaaaaa )</td>
<td>if(zero) PC ( \leftarrow ) PC + add11</td>
<td>jump if zero</td>
</tr>
<tr>
<td>01101</td>
<td>jnz add11</td>
<td>01101 ( aaaaaaaaaaaaa )</td>
<td>if(!zero) PC ( \leftarrow ) PC + add11</td>
<td>jump if not zero</td>
</tr>
<tr>
<td>01110</td>
<td>call add11</td>
<td>01110 ( aaaaaaaaaaa )</td>
<td>push PC; PC ( \leftarrow ) PC + add11</td>
<td>call function</td>
</tr>
<tr>
<td>01111</td>
<td>ret</td>
<td>01111 ( uuuuuuuuuu )</td>
<td>SP ( \leftarrow ) SP + 1; PC ( \leftarrow ) mem[SP]</td>
<td>return</td>
</tr>
<tr>
<td>10000</td>
<td>nop</td>
<td>10000 ( uuuuuuuuuu )</td>
<td>-</td>
<td>no operation</td>
</tr>
<tr>
<td>10001</td>
<td>halt</td>
<td>10001 ( uuuuuuuuuu )</td>
<td>-</td>
<td>halt processor</td>
</tr>
<tr>
<td>10010</td>
<td>push R</td>
<td>10010 ( xxx xxx xxx uu )</td>
<td>mem[SP] ( \leftarrow ) R; SP ( \leftarrow ) SP-1</td>
<td>push R onto stack</td>
</tr>
<tr>
<td>10011</td>
<td>pop R</td>
<td>10011 ( xxx xxx xxx uu )</td>
<td>SP ( \leftarrow ) SP+1; R ( \leftarrow ) mem[SP]</td>
<td>pop R from stack</td>
</tr>
<tr>
<td>10100</td>
<td>write @R1, R2</td>
<td>10100 ( x x r_1 r_1 r_1 r_2 r_2 uu )</td>
<td>mem[R1] ( \leftarrow ) R2</td>
<td>write to memory</td>
</tr>
<tr>
<td>10101</td>
<td>read R1, @R2</td>
<td>10101 ( r_1 r_1 r_1 r_2 r_2 x x uu )</td>
<td>R1 ( \leftarrow ) mem[R2]</td>
<td>read from memory</td>
</tr>
<tr>
<td>10110</td>
<td>movi R, imm8</td>
<td>10110 ( rrr rrr rrr rrr uu )</td>
<td>R ( \leftarrow ) imm8</td>
<td>move immediate</td>
</tr>
<tr>
<td>10111</td>
<td>mov SP, R</td>
<td>10111 ( xxx xxx xxx uu )</td>
<td>SP ( \leftarrow ) R</td>
<td>move to SP</td>
</tr>
<tr>
<td>11000</td>
<td>mov R, SP</td>
<td>11000 ( xxx xxx xxx uu )</td>
<td>R ( \leftarrow ) SP</td>
<td>move from SP</td>
</tr>
</tbody>
</table>

\( r, r_1, r_2 \) = 16-bit register  
mem[65536] = 64 kW memory  
add11 = 11-bit signed integer  
imm8 = 8-bit immediate value  
PC = program counter register  
SP = stack pointer register  
zero = zero flag  
x, u = “don’t care” and undefined bits.
**mov R1, R2**

Meaning: \( R1 = R2 \)

This command copies the content of register \( R2 \) to register \( R1 \). Note that this is not a move operation since the source register is not altered. An example command and its equivalent machine codes is:

\[
\text{mov H, A} \quad (00000 111 000 000 00)
\]

**add R1, R2, R3**

Meaning: \( R1 = R2 + R3 \)

This command calculates the sum of \( R2 \) and \( R3 \). The result is then placed into \( R1 \). An example command and its equivalent machine codes is:

\[
\text{add A, B, C} \quad (00001 000 001 010 00)
\]

**sub R1, R2, R3**

Meaning: \( R1 = R2 - R3 \)

This command subtracts \( R3 \) from \( R2 \). The result is then placed into \( R1 \). An example command and its equivalent machine codes is:

\[
\text{sub A, B, C} \quad (00010 000 001 010 00)
\]

**and R1, R2, R3**

Meaning: \( R1 = R2 \) and \( R3 \)
This is logical and operation. An example command and its equivalent machine codes is:
and A, B, C  (00011 000 001 010 00)

or R1, R2, R3

Meaning: R1=R2 or R3

not R

Meaning: R=not R

inc R

Meaning: R++

dec R

Meaning: R--
**sr R**

Meaning: R>> 1

```
  5  3  3  3  2
  opcode  R  R
```

opcode=0100

Shift right operation shifts the given register to the right. Same as dividing by 2. The rightmost bit is discarded. An example command and its equivalent machine codes is:

sr c  (01000 010 010 000 00)

---

**sl R**

Meaning: R<< 1

```
  5  3  3  3  2
  opcode  R  R
```

opcode=0100

Shift left operation shifts the given register to the left. Same as multiplying by 2. The leftmost bit is discarded. An example command and its equivalent machine codes is:

sl c  (01001 010 010 000 00)

---

**rr R**

Meaning: t=R.0; R>> 1; R.15=t;

```
  5  3  3  3  2
  opcode  R  R
```

opcode=01010

Rotate right operation shifts the given register to the right. The rightmost bit is moved to the leftmost bit. An example command and its equivalent machine codes is:

rr c  (01010 010 010 000 00)

---

**jmp add11**

Meaning: PC=PC± add11

```
  5  3  3  2
  opcode  add11
```

opcode=01011

This jumps the execution to another location. The address of the new location will be PC± add11 (add11 is a signed integer) An example command and its equivalent machine codes is:

jmp 03H  (01100 00000000011)

---

**jz add11**

Meaning: if(zero) PC=PC± add11
This jumps the execution to another location if zero flag is set. The address of the new location will be $PC \pm \text{add11}$ (add11 is a signed integer) An example command and its equivalent machine codes is:
\[
jz \ 03H \quad (01101\ 00000000011)
\]

\[\text{jnz add11}\]

Meaning: if(!zero) $PC=PC \pm \text{add11}$

This jumps the execution to another location if zero flag is not set. The address of the new location will be $PC \pm \text{add11}$ (add11 is a signed integer) An example command and its equivalent machine codes is:
\[
jnz \ 03H \quad (01110\ 00000000011)
\]

\[\text{call add11}\]

Meaning: push $PC; \ PC=PC \pm \text{add11}$

This command calls a procedure. The starting address is $PC \pm \text{add11}$. It is similar to jmp command. The only difference is that the return address is pushed onto the stack a priori. An example command and its equivalent machine codes is:
\[
call \ 03H \quad (01111\ 00000000011)
\]

\[\text{ret}\]

Meaning: pop $PC \equiv PC=\text{mem[SP++]}$

This command returns from procedure. The memory address that will be returned to is popped from the stack. An example command and its equivalent machine codes is:
\[
\text{ret} \quad (10000\ 00000000000)
\]

\[\text{nop}\]

Meaning: -

This is no operation (Discuss: When do we need this command?).
nop  (10001 00000000000)

**halt**

**Meaning:** Halting

```
  5  3  3  3  2
  opcode
  opcode=10010
```

This command halts the processor. In other words, execution is stopped (Discuss: When can we need this command? Why?). An example command and its equivalent machine codes is:

**halt**  (10010 00000000000)

**push R**

**Meaning:** mem[SP]=R; SP–;

```
  5  3  3  3  2
  opcode
  opcode=10011
```

Pushes register R onto the stack memory. An example command and its equivalent machine codes is:

**push B**  (10010 000 000 001 00)

**pop R**

**Meaning:** SP++; R=mem[SP]

```
  5  3  3  3  2
  opcode
  opcode=10100
```

Popes register R from the stack memory. An example command and its equivalent machine codes is:

**pop B**  (10100 001 000 000 00)

**write @R1, R2**

**Meaning:** mem[R1]=R2

```
  5  3  3  3  2
  opcode
  opcode=10101
```

Writes the content of R2 into the memory location that is is pointed by R1. An example command and its equivalent machine codes is:

**write @D, B**  (10101 000 011 001 00)

**read R1, @R2**

**Meaning:** R1=mem[R2]
Reads from memory. An example command and its equivalent machine codes is:

```plaintext
read B, @D    (10101 001 011 000 00)
```

**movi R, imm8**

Meaning:

Places 8-bit immediate value imm8 into R’s less significant 8-bit portion. An example command and its equivalent machine codes is:

```plaintext
movi B, 05H   (10111 001 00000101)
```

**mov SP, R**

Meaning:

Copies register R to SP. An example command and its equivalent machine codes is:

```plaintext
mov SP, B     (10111 000 001 000 00)
```

**mov R, SP**

Meaning:

Copies SP to register R. An example command and its equivalent machine codes is:

```plaintext
mov B, SP    (11000 001 000 000 00)
```

### 2.2 Datapath

The datapath is responsible for manipulating data. It includes (1) functional units such as adders, shifters, multipliers, ALUs, and comparators, (2) registers and other memory elements for the temporary storage of data, and (3) buses, multiplexers, and tri-state buffers for the transfer of data between the different components in the datapath, and the external world. External data enters the datapath through the data input lines. Results from the datapath operations are provided through the data output lines. These signals serve as the primary input/output data ports for the microprocessor. In the following subsections, we will see the components of the datapath in detail.
2.2.1 Registers

$\mu$311.1 has 8 general purpose registers and three special purpose registers that are program counter (PC), instruction register (IR) and stack pointer (SP). The following VHDL code is the description of a generic 16-bit register.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

library work;
use work.uP.all;

entity reg16 is port(
d: in std_logic_vector(15 downto 0);
ld: in std_logic; --load/enable.
clr: in std_logic; --async clear.
clk: in std_logic; --clock.
q: out std_logic_vector(15 downto 0) --output.
);
end reg16;

architecture description of reg16 is
begin
process(clk, clr)
begin
if clr = '1' then
q <= x"0000";
elsif rising_edge(clk) then
if ld = '1' then
q <= d;
end if;
end if;
end process;
end description;
```

In the architecture body of $\mu$311.1 implementation, special purpose registers can be implemented using register16.

2.2.2 Program Counter

Program counter (PC) contains the memory location of where the next instruction is stored. Each time an instruction is fetched from a memory location pointed to by the PC, normally the PC must be incremented to the next memory location for the next instruction. Alternatively, if the instruction is a jump instruction, the PC must be loaded with a new memory address instead.
There exists an addsub circuit in the program counter next logic circuit. This can be behaviorally implemented as follows:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

library work;
use work.uP.all;

entity addsub16 is port(
    sub: in std_logic;
    in1,in2: in std_logic_vector(15 downto 0);
    output: out std_logic_vector(15 downto 0));
end addsub16;

architecture imp of addsub16 is
begin
    with sub select output <=
        in1-in2 when '1',
        in1+in2 when '0',
        (others =>'Z') when others;
end imp;
```

Figure 6: Program Counter (PC) register and PC Next Logic.
2.2.3 Instruction Register, Stack Pointer

Instruction register (IR) stores the instruction being fetched from the program memory. PC, IR and SP can be implemented in the datapath using the 16-bit register as seen below:

1. **PCx**: `reg16 port map(P_out,PCload,reset,clk,PC_out);
2. **IRx**: `reg16 port map(RB,IRload,reset,clk,IR_out);
3. **SPx**: `reg16 port map(S_out,SPload,reset,clk,SP_out);

2.2.4 Register File

Register file contains 32 registers. The block diagram of the register file is seen in Figure 7.

![Figure 7: ALU and register file](image)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

library work;
use work.uP.all;

entity regfile is port(
  clk: in std_logic;
  reset: in std_logic;
  we: in std_logic;

```
WA: in std_logic_vector(2 downto 0);
D: in std_logic_vector(15 downto 0);
rbe: in std_logic;
rae: in std_logic;
RAA: in std_logic_vector(2 downto 0);
RBA: in std_logic_vector(2 downto 0);
portA: out std_logic_vector(15 downto 0);
portB: out std_logic_vector(15 downto 0));
end regfile;

architecture imp of regfile is
  subtype reg is std_logic_vector(15 downto 0);
type regArray is array(0 to 7) of reg;
signal RF: regArray;
begin

WritePort: Process(clk,reset)
begin
  if(reset='1') then
    for I in 0 to 7 loop
      RF(I) <= (others => '0');
    end loop;
  elsif(we='1') then
    RF(conv_integer(WA)) <= D;
  end if;
end process;

ReadPortA: Process(rae,RAA)
begin
  if(rae='1') then
    PortA <= RF(conv_integer(RAA));
  else
    PortA <= (others => 'Z');
  end if;
end process;

ReadPortB: Process(rbe,RBA)
begin
  if(rbe='1') then
    PortB <= RF(conv_integer(RBA));
  else
    PortB <= (others => 'Z');
  end if;
end process;
end imp;
Question 3 Implement program counter (PC), instruction register (IR) and output register in VHDL (See [Hwa04] for VHDL). Make a simulation in Modelsim to make sure that they run properly.

2.2.5 Multiplexers

In $\mu$311, we use 2 and 4-channel multiplexers.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
library work;
use work.uP.all;

entity mux2 is port(
  s: in std_logic;
  x0,x1: in std_logic_vector(15 downto 0);
  y: out std_logic_vector(15 downto 0));
end mux2;

Architecture behavioral of mux2 is
begin
  Process(s,x0,x1)
  begin
    case s is
      when '0' => y <= x0;
      when '1' => y <= x1;
    when others => y <= "XXXXXXXXXXXXXXXX";
    end case;
  end Process;
end behavioral;
```

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
library work;
use work.uP.all;

entity mux4 is port(
  S: in std_logic_vector(1 downto 0);
  x0,x1,x2,x3: in std_logic_vector(15 downto 0);
  y: out std_logic_vector(15 downto 0));
end mux4;

architecture imp of mux4 is
begin
```
process(S, x0, x1, x2, x3)
begin
  case S is
    when "00" => y <= x0;
    when "01" => y <= x1;
    when "10" => y <= x2;
    when "11" => y <= x3;
    when others => y <= "XXXXXXXXXXXXXXXX";
  end case;
end process;
end imp;

2.2.6 Buffers
Besides multiplexers, we need unidirectional and bidirectional buffers to produce address and databus signals.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
library work;
use work.uP.all;

entity buf is port(
  enable: in std_logic;
  input: in std_logic_vector(15 downto 0);
  output: out std_logic_vector(15 downto 0));
end buf;

architecture imp of buf is
begin
  with enable select
  output <= input when '1',
  (others =>'Z') when others;
end imp;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
library work;
use work.uP.all;

entity buf2 is port(
enable: in std_logic;
direction: in std_logic;
input: inout std_logic_vector(15 downto 0);
output: inout std_logic_vector(15 downto 0));
end buf2;

architecture imp of buf2 is
begin
Bproc: process(enable,direction,input,output)
begin
if(enable='1' and direction='1') then output <= input;
elsif(enable='1' and direction='0') then input <= output;
else input <= (others => 'Z');
output <= (others =>'Z');
end if;
end process;
end imp;

2.2.7 ALU and Shifter

Figure 8: ALU schematic.

The arithmetic logic unit (ALU) is one of the main components inside a microprocessor. It is responsible for performing arithmetic and logic operations, such as addition, subtraction, logical AND, and logical OR. μ311.1’s ALU performs only two actions: addition and subtraction. Our ALU has two input ports, A and B, one output port F and a selection input s, as seen in figure 8. We can define the function of ALU as:

\[
F = f(s, A, B) \tag{1}
\]

\[
F = s_2s_1s_0A + s_2s_1s_0(A\&B) + s_2s_1s_0(A|B) + s_2s_1s_0(A') \tag{2}
\]
\[
+ s_2s_1s_0(A + B) + s_2s_1s_0(A + B' + 1) \tag{3}
\]
\[
+ s_2s_1s_0(A + 1) + s_2s_1s_0(A - 1) \tag{4}
\]

To implement ALU we will use a generic circuit consisting of a set of full adders augmented with arithmetic and logic extenders as shown in Figure 9. The two combinational circuits in front of the full adder (FA) are labeled LE and AE. The logic extender (LE) is for manipulating all logical operations.
whereas the arithmetic extender (AE) is for manipulating all arithmetical operations. The LE performs logical operations on the two primary operands, \( a_i \) and \( b_i \), before passing the result to the first operand, \( x_i \), of the FA. On the other hand, the AE only modifies the second operand, \( b_i \), and passes it to the second operand, \( y_i \), of the FA where the actual arithmetical operation is performed. To perform additions and subtractions, we only need to modify \( y_i \) (the second operand to the FA) so that all operations can be done with additions. The combinational circuit labeled CE (for carry extender) is for modifying the primary carry-in signal, \( c_0 \), so that arithmetical operations are performed correctly.

**Question 4** Design the ALU using common digital design techniques that benefit from truth tables, karnaugh maps or other simplification methods. The function of ALU is given in table 2.

Below, you can find the necessary VHDL programs to implement the ALU. The first program describes the full adder circuit:

```
--FA.vhd: Full Adder

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity FA is port(
    carryIn: in std_logic;
    \dots
);
Figure 10: Implementation of ALU.

<table>
<thead>
<tr>
<th>s₂</th>
<th>s₁</th>
<th>s₀</th>
<th>Operation Name</th>
<th>Operation</th>
<th>x₁(LE)</th>
<th>y₁(AE)</th>
<th>c₀(CE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Pass</td>
<td>Pass A to output</td>
<td>a₀</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>AND</td>
<td>A AND B</td>
<td>a₁ AND b₁</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>OR</td>
<td>A OR B</td>
<td>a₁ OR b₁</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>NOT</td>
<td>A'</td>
<td>a₀'</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Addition</td>
<td>A + B</td>
<td>a₀ b₀</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Subtraction</td>
<td>A − B</td>
<td>a₀ b₀'</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Increment</td>
<td>A + 1</td>
<td>a₀</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Decrement</td>
<td>A − 1</td>
<td>a₀</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

```vhdl
9 carryOut:       out std_logic;
10 x, y : in std_logic;
11 s : out std_logic
12 );
13 end FA;

architecture imp of FA is
begin
  s <= x xor y xor carryIn;
  carryOut <= (x and y) or (carryIn and (x or y));
end imp;
```

The ALU will be developed using structural programming style. Therefore, all the modules of ALU are hierarchically connected to each other. Next, 16 FA are cascaded to form a 16-bit addsub circuit:

```vhdl
-- FA16.vhd: Array of 16 Full Adders

library ieee;
library work;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.u311.all;

entity FA16 is port(
A : in std_logic_vector(15 downto 0);
B : in std_logic_vector(15 downto 0);
```
F : out std_logic_vector(15 downto 0);
cIn: in std_logic;
unsigned_overflow: out std_logic;
signed_overflow: out std_logic
);
end FA16;

architecture imp of FA16 is
signal C: std_logic_vector(15 downto 1);
begin
U0: FA port map(cIn, C(1), A(0), B(0), F(0));
U1_14: for I in 1 to 14 generate
begin
U: FA port map(C(I), C(I+1), A(I), B(I), F(I));
end generate U1_14;
U15: FA port map(C(15), unsigned_overflow,A(15),B(15),F(15));
signed_overflow <= C(15) xor C(14) ;
end imp;

The following circuits describe the logical and arithmetical extension parts of the ALU:

-- LE.vhd: Logic Extender circuit
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
entity LE is port(
    S: in std_logic_vector(2 downto 0);
    a, b: in std_logic;
    x: out std_logic
);
end LE;

architecture imp of LE is
begin
process(S,a,b)
begin
    case S is
        when "000" => x <= a;
        when "001" => x <= a and b;
        when "010" => x <= a or b;
        when "011" => x <= not a;
        when others => x <= a;
    end case;
end process;
library ieee;
library work;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use work.u311.LE;

entity LE16 is port(
    S: in std_logic_vector(2 downto 0);
    A, B: in std_logic_vector(15 downto 0);
    x: out std_logic_vector(15 downto 0)
);
end LE16;

architecture imp of LE16 is
begin
    LE16X: for I in 0 to 15 generate
        LEX: LE port map(S, A(I), B(I), X(I));
    end generate LE16X;
end imp;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity AE is port(
    S: in std_logic_vector(2 downto 0);
    a, b: in std_logic;
    x: out std_logic
);
end AE;

architecture imp of AE is
begin
    process(S,b)
    begin
        case S is
            when "100" => x <= b;
    end process;
end imp;
when "101" => x <= not b;
when "110" => x <= '0';
when "111" => x <= '1';
when others => x <= '0';
end case;
end process;
end imp;

-- AE16.vhd: Array of 16 AE circuits
library ieee;
library work;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use work.u311.AE;
entity AE16 is port(
S: in std_logic_vector(2 downto 0);
A, B: in std_logic_vector(15 downto 0);
Y: out std_logic_vector(15 downto 0)
);
end AE16;
architecture imp of AE16 is
begin
AE16X: for I in 0 to 15 generate
AEX: AE port map(S, A(I), B(I), Y(I));
end generate AE16X;
end imp;

The last part of the ALU is the shifter. This allows shifting a given number one bit to the left or right. The shifter is composed of 16 multiplexers:

-- shifter.vhd: 16 bit shifter
library ieee;
library work;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use work.u311.all;
entity shifter16 is port(
S: in std_logic_vector(1 downto 0);
A: in std_logic_vector(15 downto 0);
Y: out std_logic_vector(15 downto 0);
carryOut: out std_logic;
zero: out std_logic);
end shifter16;
architecture imp of shifter16 is
begin
process(S)
begin
if(S="01") then
carryOut <= A(15);
elsif(S="10") then
carryOut <= A(0);
end if;
end process;
U0 : mux port map(S, A(0), '0', A(1), A(1), Y(0));
U1_14: for I in 1 to 14 generate
UX: mux port map(S, A(I), A(I-1), A(I+1), A(I+1), Y(I));
end generate U1_14;
U15 : mux port map(S, A(15), A(14), '0', A(0), Y(15));
process(A,S)
begin
if(S="00") then
if(A = "0000") then
zero <= '1';
else
zero <= '0';
end if;
end if;
end process;
end imp;
The last step is to bring all these parts together to constitute the ALU as follows:
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
library work;
use work.uP.all;
entity ALU is port(
  S: in std_logic_vector(4 downto 0);
  A, B: in std_logic_vector(15 downto 0);
  F: out std_logic_vector(15 downto 0);
  unsigned_overflow: out std_logic;
  signed_overflow: out std_logic;
  carry: out std_logic);
end ALU;

architecture imp of ALU is 
signal X, Y, ShiftInput: std_logic_vector(15 downto 0);
signal c0: std_logic;
begin
  CarryExtender_ALU: c0 <= (S(0) xor S(1)) and S(2);
  LogicExtender16_ALU: LE16 port map(S(2 downto 0), A, B, X);
  ArithmeticExtender16_ALU: AE16 port map(S(2 downto 0), A, B, Y);
  FA16_ALU: FA16 port map(X, Y, ShiftInput, c0, unsigned_overflow, signed_overflow, carry);
end imp;

Despite its less resource consumption, the structural implementation is really cumbersome. The behavioural implementation of the ALU, indeed, would be as easy as follows:

-- alu2.vhd: Alternative implementation of ALU

library ieee;
library work;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use work.u311.all;

entity ALU_Behavioral is port (
  S: in std_logic_vector(4 downto 0);
  A, B: in std_logic_vector(15 downto 0);
  F: out std_logic_vector(15 downto 0);
  zero: out std_logic
);
end ALU_Behavioral;

architecture imp of ALU_Behavioral is 
signal X, Y, ShiftInput: std_logic_vector(15 downto 0);
signal c0: std_logic;
begin
  ALU: process(S, A, B)
begin
  case S is
    when "00000" => F <= A;
    when "00100" => F <= A and B;
    when "01000" => F <= A or B;
    when "01100" => F <= not A;
    when "10000" => F <= A + B;
    when "10100" => F <= A - B;
    when "11000" => F <= A + 1;
    when "11100" => F <= A - 1;
    when "00001" => F <= to_stdlogicvector(to_bitvector(A) sll 1);
    when "00010" => F <= to_stdlogicvector(to_bitvector(A) srl 1);
    when "00011" => F(15) <= A(0);
    when others => F <= "ZZZZZZZZZZZZZZZZZ";
  end case;
end process;
end imp;

The entire datapath can then be constructed as follows:

library ieee;
library work;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.uP.all;

entity datapath is port(
  clk: in std_logic;
  reset : in std_logic;
  pcen, den, dir, aen: in std_logic;
  SPlode, PClode, IRload: in std_logic;
  Psel, Ssel, Rsel, Osel : in std_logic_vector(1 downto 0);
  sub2: in std_logic;
  jmpMux : in std_logic;
  IR : out std_logic_vector (4 downto 0);
  zero: out std_logic;
  ALUsel : in std_logic_vector (4 downto 0);
  we, rae, rbe : in std_logic;
  Buf2_out: out std_logic_vector(15 downto 0);
  Buf3_out: inout std_logic_vector(15 downto 0)
);
end dataPath;

architecture imp of datapath is
-- Signals
signal ALU_out, PC_out, IR_out, SP_out, Pm_out: std_logic_vector(15 downto 0);
signal P_out, Add1_out, Add2_out, O_out, S_out, R_out: std_logic_vector(15 downto 0);
signal RA, RB: std_logic_vector(15 downto 0);
signal int_in, pc_in: std_logic_vector(15 downto 0);
signal sub1: std_logic;
signal Pm_in: std_logic_vector(15 downto 0);

begin

int_in <= "000000000" & IR_out(2 downto 0) & "1111";
pc_in <= X"00" & IR_out(7 downto 0);
IR <= IR_out(15 downto 11);

-- Special registers
PCx: reg16 port map(P_out,PCload,reset,clk,PC_out);
IRx: reg16 port map(RB,IRload,reset,clk,IR_out);
SPx: reg16f port map(S_out,SPload,reset,clk,SP_out);

--- Multiplexers
Pmux4: mux4 port map(Psel,int_in,int_in,RB,Add1_out,P_out);
Rmux4: mux4 port map(Rsel,RA,SP_out,RB,pc_in,R_out);
Smux4: mux4 port map(Ssel,X"0000",X"0000",RA,Add2_out,S_out);
Omux4: mux4 port map(Osel,PC_out,SP_out,X"0000",RA,O_out);
Pm_in <= "000000" & IR_out(9 downto 0);
Pmux2: mux2 port map(jmpMux,X"0001",Pm_in,Pm_out);

--- ALU and Regfile
Regf: regfile port map(clk,reset,we,IR_out(10 downto 8),ALU_out,rbe,rae,IR_out(7 downto 5),IR_out(4 downto 2),RA,RB);
--ALUx: alu2 port map(ALUsel,R_out,RB,ALU_out,open,open,open);
ALUx: ALU_Behavioral port map(ALUsel,R_out,RB,ALU_out,open);

-- zero flag !!
process(ALU_out)
begin
if(IR_out(15 downto 11) = "00001" or IR_out(15 downto 11) = "00111" or IR_out(15 downto 11) = "00010") then
    if (ALU_out = "0000") then
        zero <= '1';
    else
        zero <= '0';
    end if;
end if;
end process;
--- Buffers -------------------------------------------------------
 Buf1x: buf port map(pcen,PC_out,Buf3_out);
 Buf2x: buf port map(aen,O_out,Buf2_out);
 Buf3x: buf2 port map(den,dir,RB,Buf3_out);

--- Addsub circuits -----------------------------------------------
 sub1 <= IR_out(10) and jmpMux;
 Addsub1: addsub16 port map(sub1,PC_out,Pm_out,Add1_out);
 Addsub2: addsub16 port map(sub2,SP_out,X"0001",Add2_out);

end imp;

2.2.8 The processor

library ieee;
library work;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use work.uP.all;

entity u311_1 is port(
  clk: in std_logic;
  reset: in std_logic;
  opfetch: out std_logic;
  INT: in std_logic;
  INTA: out std_logic;
  WR: out std_logic;
  RD: out std_logic;
  A: out std_logic_vector(15 downto 0);
  D: inout std_logic_vector(15 downto 0));
end u311_1;

architecture imp of u311_1 is

signal pcen,aen,den,dir: std_logic;
signal SPload,IRload,PCload: std_logic;
signal Psel,Ssel,Rsel,Osel : std_logic_vector(1 downto 0);
signal IR: std_logic_vector(4 downto 0);
signal we,rae,rbe: std_logic;
signal ALUsel: std_logic_vector(4 downto 0);
signal zero: std_logic;
signal sub2: std_logic;
signal jmpMux: std_logic;

begin
CU: controller port map(clk,reset,pcen,den,dir,aen,SPload,PCload,IRload,Psel,Ssel,Rsel,Osel,sub2,jmpMux,opfetch,IR,zero,ALUsel,we,rae,rbe,int,inta,wr,rd);
DP: datapath port map(clk,reset,pcen,den,dir,aen,SPload,PCload,IRload,Psel,Ssel,Rsel,Osel,sub2,jmpMux,IR,zero,ALUsel,we,rae,rbe,A,D);
end imp;

2.3 Stack

Stack region can be defined in the external memory. Stack pointer register SP must be initialized for this. Recall that SP holds 0x0000 after a reset. An appropriate value could be for example 0x00FF. This initialization can be done with the following code:

```plaintext
1 mov a, ffh
2 mov c, 08h
3 L: sl a
4 dec c
5 jnz L
6 mob b, ffh
7 add a, a, b
8 mov sp, a
```

**Question 5** Assume that the first instruction of the program is “pop a”. In this case, what would register A hold after this command?

2.4 Control Unit

The control unit inside the microprocessor is a finite state machine. By stepping through a sequence of states, the control unit controls the operations of the datapath. For each state that the control unit is in, the output logic that is inside the control unit will generate all of the appropriate control signals for the datapath to perform one data operation. These data operations are referred to as register-transfer operations. Each register-transfer operation consists of reading a value from a register, modifying the value by one or more functional units, and finally, writing the modified value back into the same or a different register.

The block diagram of our control unit is given in figure 11. Figure 12 shows the FSM of µ311.1.
Figure 11: Control Unit.
2.4.1 Bus Cycles

μ311.1 has 4 cycles: opcode fetch, read, write and interrupt. The timing diagram for each cycle is given below.
1. Opcode Fetch Cycle

![Figure 13: Opcode fetch cycle]

2. Memory/IO Read Cycle

![Figure 14: Memory - I/O read cycle]
3. Memory/IO Write Cycle

![Figure 15: Memory - I/O write cycle](image)

4. Interrupt Cycle

![Figure 16: Interrupt cycle](image)
Question 6  Complete the next-state diagram of the control unit given in the table 3 and design the control unit using J-K flip-flops.

```vhdl
-- controller.vhd: control unit
library ieee;
library work;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_textio.all;
use IEEE.std_logic_arith.all;
use IEEE.numeric_bit.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_signed.all;
use IEEE.std_logic_unsigned.all;
use IEEE.math_real.all;
use IEEE.math_complex.all;
use work.uP.all;

entity controller is port(
    clk: in std_logic;
    reset : in std_logic;
    pcen, den, dir, aen: out std_logic;
    SPload, PClode, IRload: out std_logic;
    Psel, Ssel, Rsel, Osel : out std_logic_vector(1 downto 0);
    sub2: out std_logic;
    jmpMux : out std_logic;
    opfetch : out std_logic;
    IR : in std_logic_vector (4 downto 0);
    zero: in std_logic;
    ALUsel : out std_logic_vector (4 downto 0);
    we, rae, rbe : out std_logic;
    int: in std_logic;
    inta, wr, rd: out std_logic);
end controller;

architecture imp of controller is

-- controller.vhd: control unit
```
s_inc,
s_dec,
s_sr,
s_sl,
s_rr,
s_clr,
s_jmp,
s_call,
s_ret,
s_nop,
s_halt,
s_psh,
s_psh2,
s_pop,
s_pop2,
s_wrt,
s_read,
s_movi,
s_mvspr,
s_mvrsp,
s_r_c1,
s_r_c2,
s_r_c3,
s_w_c1,
s_w_c2,
s_w_c3,
s_int_c1,
s_int_c2,
s_int_c3);

    signal state: state_type := s_strt;
    signal zero_flag: std_logic;

begin

NEXT_STATE_LOGIC: process(clk, reset)
    variable int_occr : boolean := false;
begin

if(reset = '1') then
    state <= s_strt;
elsif (int = '1' and int_occr=false) then
    int_occr := true;
elsif(clk'event and clk='1') then

    case state is
when s_strt => state <= s_ftch after 1ns;
when s_ftch => state <= s_dcd after 1ns;
when s_dcd2 =>

case IR is
  when "00000" => state <= s_mov after 1ns;
  when "00001" => state <= s_add after 1ns;
  when "00010" => state <= s_sub after 1ns;
  when "00011" => state <= s_and after 1ns;
  when "00100" => state <= s_or after 1ns;
  when "00101" => state <= s_not after 1ns;
  when "00110" => state <= s_inc after 1ns;
  when "00111" => state <= s_dec after 1ns;
  when "01000" => state <= s_sr after 1ns;
  when "01001" => state <= s_sl after 1ns;
  when "01010" => state <= s_rr after 1ns;
  when "01011" => state <= s_jmp after 1ns;
  when "01100" => if(zero_flag = '1') then state <= s_jmp after 1ns;
                  elsif(zero_flag = '0') then state <= s_nop after 1ns;
                  end if;
  when "01101" => if(zero_flag = '0') then state <= s_jmp after 1ns;
                  elsif(zero_flag = '1') then state <= s_nop after 1ns;
                  end if;
  when "01110" => state <= s_call after 1ns;
  when "01111" => state <= s_ret after 1ns;
  when "10000" => state <= s_nop after 1ns;
  when "10001" => state <= s_halt after 1ns;
  when "10010" => state <= s_psh after 1ns;
  when "10011" => state <= s_pop after 1ns;
  when "10100" => state <= s_wrt after 1ns;
  when "10101" => state <= s_read after 1ns;
  when "10110" => state <= s_movi after 1ns;
  when "10111" => state <= s_mvspr after 1ns;
  when "11000" => state <= s_mvrsp after 1ns;
  when others =>
    state <= s_strt after 1us;
end case;

when s_halt => state <= s_halt after 1ns;
when s_wrt => state <= s_w_c1 after 1ns;
when s_read => state <= s_r_c1 after 1ns;
when s_w_c1 => state <= s_w_c2 after 1ns;
when s_r_c1 => state <= s_r_c2 after 1ns;
when s_w_c2 => state <= s_w_c3 after 1ns;
when s_r_c2 => state <= s_r_c3 after 1ns;
when s_int_c1 => state <= s_int_c2 after 1ns;
when s_int_c2 => state <= s_int_c3 after 1ns;

when others =>
  if(int_occr = true) then
    state <= s_int_c1 after 1ns;
    inta <= '1' after 1ns;
    int_occr := false;

  elsif(int_occr = false) then
    state <= s_ftch after 1ns;
  end if;
end case;
end if;

if(clk'event and clk='0') then
  case state is
    when s_psh => state <= s_psh2 after 1ns;
    when s_pop => state <= s_pop2 after 1ns;
    when s_dcd => state <= s_dcd2 after 1ns;
    when others =>
      end case;
  end if;
end process;

OUTPUT_LOGIC: process(state)
begin
  case state is
    when s_strt =>
      inta <= 'Z';
      WR <= 'Z';
      RD <= 'Z';
      opfetch <= 'Z';
      pcen <= '0';
      den <= '0';
      dir <= '0';
      aen <= '0';
      SPload <= '0';
      PClode <= '0';
      IRload <= '0';
      Psel <= "XX";
      Ssel <= "XX";
      Osel <= "XX";
  end case;
end case;
end if;
end process;
ALUsel <= "XXXXX";
Rsel <= "XX";
sub2 <= 'X';
jmpMux <= 'X';
we <= '0';
rbe <= '0';
rae <= '0';
when s_ftch =>
case IR is
  when "00001" =>
    if(zero='1') then zero_flag <= '1';
    else zero_flag <= '0';
    end if;
  when "00111" =>
    if(zero='1') then zero_flag <= '1';
    else zero_flag <= '0';
    end if;
  when "00010" =>
    if(zero='1') then zero_flag <= '1';
    else zero_flag <= '0';
    end if;
  when "00110" =>
    if(zero='1') then zero_flag <= '1';
    else zero_flag <= '0';
    end if;
  when others =>
end case;
inta <= 'Z';
WR <= 'Z';
RD <= 'Z';
opfetch <= '1' after 2ns;
pcen <= '0';
den <= '1';
dir <= '0';
aen <= '1';
SPload <= '0';
PCload <= '1';
IRload <= '1';
Psel <= "11";
Ssel <= "00";
Osel <= "00";
ALUsel <= "XXXXX";
Rsel <= "XX";
sub2 <= 'X';
jmpMux <= '0';
we <= '0';
rbe <= '0';
rae <= '0';

when s_dcd =>
 inta <= 'Z';
WR <= 'Z';
RD <= 'Z';
opfetch <= '0';
pcen <= '0';
den <= '0';
dir <= '0';
aen <= '0';

case IR is
  when "10011" => SPload <= '1'; -- for pop inst.
    sub2 <= '0';
  when "01111" => SPload <= '1';-- for ret inst.
    sub2 <= '0';
  when others => SPload <= '0';
    sub2 <= 'X';
end case;

PCload <= '0';
IRload <= '0';
Psel <= "XX";
Ssel <= "11";
Osel <= "XX";
ALUsel <= "XXXXX";
Rsel <= "XX";
sub2 <= '0';
jmpMux <= '0';
we <= '0';
rbe <= '0';
rae <= '0';

when s_dcd2 =>
  inta <= 'Z';
WR <= 'Z';
RD <= 'Z';
opfetch <= '0';
pcen <= '0';
den <= '0';
dir <= '0';
aen <= '0';
SPload <= '0';
PCload <= '0';
IRload <= '0';
Psel <= "XX";
Ssel <= "XX";
Osel <= "XX";
ALUsel <= "XXXX";
Rsel <= "XX";
sub2 <= 'X';
jmpMux <= '0';
we <= '0';
rbe <= '0';
rae <= '0';

when s_mov =>
when s_add =>
when s_sub =>
when s_and =>
when s_or =>
when s_not =>
when s_inc =>
when s_dec =>
when s_sr =>
when s_sl =>
when s_rr =>
when s_jmp =>
when s_call =>
when s_ret =>
when s_nop =>
when s_halt =>
when s_psh =>
when s_psh2 =>
when s_pop =>
when s_pop2 =>
when s_wrt =>
when s_read =>
when s_movi =>
when s_mvspr =>
when s_mvrsp =>
when s_r_c1 =>
when s_r_c2 =>
when s_r_c3 =>
when s_w_c1 =>
when s_w_c2 =>
when s_w_c3 =>
when s_int_c1 =>
when s_int_c2 =>
when s_int_c3 =>
when others => inta <= 'X';
end case;
end process;
end imp;
Table 3: Control Unit Next-State Table (Complete the table!)

| Nr | \( J_{15-11} \) | state | ASel | Ssel | Psel | Osel | IRload | PCload | SPload | jmpMux | sub2 | we | rae | rbe | dir | den | pcdn | aen | wr | rd | opfch |
|----|----------------|-------|------|------|------|------|--------|--------|--------|--------|------|----|----|----|----|----|-----|-----|----|----|----|-------|
| 0  | xxxx           | start | xx   | xx   | xx   | 0    | 0      | 0      | 0      | 0      | 0    | 0  | 0  | 0  | 0  | 0  | 0   | x   | x  | x  | x  | xxxx              |
| 1  | xxxx           | fetch |      |      |      | 0    | 0      | 0      | 0      | 0      | 0    | 0  | 0  | 0  | 0  | 0  | 0   | z   | z  | z  | z  | xxxx              |
| 2  | xxxx           | decode|      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 3  | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 4  | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 5  | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 6  | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 7  | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 8  | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 9  | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 10 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 11 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 12 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 13 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 14 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 15 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 16 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 17 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 18 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 19 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 20 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 21 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 22 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 23 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 24 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 25 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 26 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 27 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 28 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
| 29 | xxxx           |       |      |      |      |      |        |        |        |        |      |    |   |   |   |   |     |     |   |   |   | xxxx              |
3 Testbench

The minimum configuration for $\mu311.1$ to run should include these units:

- $\mu311.1$
- clock circuit
- reset circuit
- program memory

The circuit diagram is shown in Figure 17.

![Figure 17: Testbench with minimum configuration.](image)

Therefore, the following testbench can be used to test the microprocessor. Note that this testbench is for simulation only and it cannot be synthesized due to some non-synthesizable parts in it. On the other hand, “u311_1.vhd” is fully synthesizable and it can be realized on an FPGA.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

library work;
use work.uP.all;

entity testbench is
   --no ports.
end testbench;
architecture imp of testbench is
   signal clk: std_logic;
```

Tolga Ayav
signal reset: std_logic;
signal opfetch: std_logic;
signal wr, rd : std_logic;
signal addressbus, databus: std_logic_vector(15 downto 0);

begin
  --minimum configuration
  clock_gen: clk_gen port map(clk);
  reset_gen: rst_gen port map(reset);
  processor: u311_1 port map(clk, reset, opfetch, '0', open, wr, rd, addressbus, databus);
  rom: rom1024 port map('0', opfetch, addressbus(9 downto 0), databus);
  ram: ram1024 port map(reset, '0', wr, rd, addressbus(9 downto 0), databus);
end imp;

4 Address Decoding and I/O Communication

Figure 18: Connecting program memory, 1K RAM and a 8255 to \( \mu311.1 \).
5 Interrupts

µ311.1’s interrupt cycle consists of the following steps:

1. Interrupt signal is asserted by an external device (INT pin of the µ311.1).
2. µ311.1 produces an acknowledge signal at INTA pin.
3. 3-bit interrupt number is fetched from the data bus ($D_{2-0}$ pins)
4. The content of PC is pushed onto the stack.
5. Execution jumps to the address of “00000000 & D2-0 & 1111” where the related interrupt service routine (ISR) is located.
6. ret command returns from the ISR.

Each ISR has a predefined location in the memory. The ISR addresses are given in Table 4.

<table>
<thead>
<tr>
<th>Int. No</th>
<th>$D_{2-0}$</th>
<th>ISR address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>0000000000 000 1111</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>0000000000 001 1111</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>0000000000 010 1111</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>0000000000 011 1111</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0000000000 100 1111</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>0000000000 101 1111</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>0000000000 110 1111</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>0000000000 111 1111</td>
</tr>
</tbody>
</table>
Figure 20 shows an example application that uses three interrupts. These external interrupt signals are ORed and connected to the INT pin of the processor. Note that the same interrupt signals are also connected to the data bus, which constitutes the necessary interrupt number so that the processor can jump to the related ISR. Hereby, INTA is used as the output enable signal for 74'244 buffer. Figure 21 shows the signal waveform of an interrupt cycle and the related interrupts used by this application.

Figure 20: Example application: Connecting 3 different interrupt sources to $\mu$311.1.

<table>
<thead>
<tr>
<th>Int. No</th>
<th>$D_{2-0}$</th>
<th>ISR address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>001</td>
<td>000000000 001 1111</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>000000000 010 1111</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>000000000 100 1111</td>
</tr>
</tbody>
</table>

Figure 21: Related ISRs.

6 Additional Instructions and Units

6.1 Watchdog Timer

Question 7 Add the circuit given in Figure 22 to $\mu$311.1. Discuss the function of this circuit. What additional instruction(s) do you suggest to use this circuit?

6.2 Base Pointer Register

Question 8 Add two additional instructions like “mov bp,sp” and “mov sp,bp”. Make the necessary modifications in $\mu$311.1 such as adding a new base pointer register. Discuss the function and benefits of
Question 9 In ALU-related instructions, the least significant two bits are not used. Can you suggest a modification such that when these bits are used, the destination register will become SP or BP so that all arithmetic and logic operations can also be performed on these special registers. These two bits can be used as suggested in the table:

00 - BP, SP are not used
01 - Destination is SP
10 - Destination is BP
11 - Reserved for future use
7 Programming

7.1 High-Level Programming

Let’s define the following high-level imperative programming language $C_{311.1}$ for $\mu_{311.1}$:

\[
S ::= \begin{cases} 
    x = A & \text{assignment} \\
    p = A | *p = A & \text{pointer assignment} \\
    x++ | x-- & \text{increment, decrement} \\
    \text{nop} & \text{no operation} \\
    S_1;S_2 & \text{sequencing} \\
    \text{if } B \text{ then } S_1 \text{ else } S_2 & \text{conditional} \\
    \text{while } B \text{ do } S & \text{iteration} \\
    \text{func}(\text{vars}) & \text{function call} \\
    \text{isr0..7()} & \text{interrupt service routine} \\
    \text{uint16 } x, *p & \text{variable definition} \\
    \text{register } x & \text{variable definition} \\
\end{cases}
\]

where

\[
B ::= \text{true} | \text{false} | x_1 \square x_2 | \text{not } x
\]

$\square ::= \{\text{and, or}\}$

Compilation is another topic and beyond the scope of this document. Here, we will assume that a compiler can generate the intermediary assembly code given in figure 29.

**Question 10** Note that $C_{abs}$ is a very limited language such that it is well suited to the hardware. For example, it supports 4 mathematical operations and only 8-bit constant values. Discuss if we could use multiplication, i.e., $op \in \{+, -, \times\}$. How can the compiler translate the following line to the assembly of $\mu_{311.1}$?

\[
t := t \times n;
\]

Could we also generalize 8-bit constants to 16-bit? If so, how would you translate the following line to the assembly of $\mu_{311.1}$?

\[
\text{if } t! = 1024 \text{ then } t := t + 1;
\]

**Question 11** Try to develop a compiler for our $C_{abs}$ language using lex and yacc tools.
7.2 Assembly and Linking

Assembly and linking are the last steps in the compilation process - they turn a list of instructions into an image of the program’s bits in memory. Figure 28 highlights the role of assemblers and linkers in the compilation process. This process is often hidden from us by compilation commands that do everything required to generate an executable program. As the figure shows, most compilers do not directly generate machine code, but instead create the instruction-level program in the form of human-readable assembly language. Generating assembly language rather than binary instructions frees the compiler writer from details extraneous to the compilation process, which include the instruction format as well as the exact addresses of instructions and data. The assembler’s job is to translate symbolic assembly language statements into bit-level representations of instructions known as object code. The assembler takes care of instruction formats and does part of the job of translating labels into addresses. However, since the program may be built from many files, the final steps in determining the addresses of instructions and data are performed by the linker, which produces an executable binary file. That file may not necessarily be located in the CPU’s memory, however, unless the linker happens to create the executable directly in RAM. The program that brings the program into memory for execution is called a loader.

Since we do not have any compiler to compile the high-level source code to the assembly format of μ311.1, we will do it by hand. The assembly output is seen in figure 29.

The simplest form of the assembler assumes that the starting address of the assembly language program has been specified by the programmer. The addresses in such a program are known as absolute addresses. However, in many cases, particularly when we are creating an executable out of several component files, we do not want to specify the starting addresses for all the modules before assembly. If we did, we would have to determine before assembly not only the length of each program in memory but also the order in
which they would be linked into the program. Most assemblers therefore allow us to use relative addresses by specifying at the start of the file that the origin of the assembly language module is to be computed later. Addresses within the module are then computed relative to the start of the module. The linker is then responsible for translating relative addresses into absolute addresses.

7.3 Sample Programs

EXAMPLE1: Assume that \( \mu_311.1 \) is attached to 1k program memory (starting from address 0000h) and 64k RAM (starting from address 0000h). Find the multiplication of two numbers stored in register A and B. The result will be placed in register C.

The assembly program is shown in Figure 30.

EXAMPLE2: Find the first 30 Fibonacci numbers and place them into the RAM starting from address 0000h.

The assembly program is shown in Figure 31.

EXAMPLE3: Convert the C program given in Figure 32 to assembly program.

The resulting assembly program is shown in Figure 33.

EXAMPLE 4: \( \mu_311.1 \) is attached to a 256 word RAM placed at the beginning of the memory map and
a communication device that has 2 registers placed at addresses 0140H and 0141H. I/O device sends an interrupt to CPU when it receives data from the external world and places it into its registers. Whenever ISR0 is invoked, the difference between those two registers should be computed and placed iteratively starting from the RAMs first location. Write a C and assembly program for \( \mu 311.1 \).

The resulting C program is given in Figure 34 and assembly program is given in Figure 35.

### 7.4 Assemblers

When translating assembly code into object code, the assembler must translate opcodes and format the bits in each instruction, and translate labels into addresses. In this section, we review the translation of assembly language into binary. Labels make the assembly process more complex, but they are the most important abstraction provided by the assembler. Labels let the programmer (a human programmer or a compiler generating assembly code) avoid worrying about the absolute locations of instructions and data. Label processing requires making two passes through the assembly source code as follows:

1. The first pass scans the code to determine the address of each label.
2. The second pass assembles the instructions using the label values computed in the first pass.

The name of each symbol and its address is stored in a symbol table that is built during the first pass. The symbol table is built by scanning from the first instruction to the last (For the moment, we assume that we know the absolute address of the first instruction in the program). During scanning, the current location in memory is kept in a program location counter (PLC). Despite the similarity in name to a program counter, the PLC is not used to execute the program, only to assign memory locations to labels. For example, the PLC always makes exactly one pass through the program, whereas the program counter makes many passes over code in a loop. Thus, at the start of the first pass, the PLC is set to the program’s starting address and the assembler looks at the first line. After examining the line, the assembler updates the PLC to the next location (since our architecture is one byte long, the PLC would
be incremented by one) and looks at the next instruction. If the instruction begins with a label, a new entry is made in the symbol table, which includes the label name and its value. The value of the label is equal to the current value of the PLC. At the end of the first pass, the assembler rewinds to the beginning of the assembly language file to make the second pass. During the second pass, when a label name is found, the label is looked up in the symbol table and its value substituted into the appropriate place in the instruction. In our program, the only label $L_1$ is replaced with “10111”.

7.5 Linking

Many assembly language programs are written as several smaller pieces rather than as a single large file. Breaking a large program into smaller files helps delineate program modularity. If the program uses library routines, those will already be preassembled, and assembly language source code for the libraries may not be available for purchase. A linker allows a program to be stitched together out of several smaller pieces. The linker operates on the object files created by the assembler and modifies the assembled code to make the necessary links between files. Some labels will be both defined and used in the same file. Other labels will be defined in a single file but used elsewhere. The place in the file where a label is defined is known as an entry point. The place in the file where the label is used is called an external reference. The main job of the loader is to resolve external references based on available entry points. As a result of the need to know how definitions and references connect, the assembler passes to the linker not only the object file but also the symbol table. Even if the entire symbol table is not kept for later debugging purposes, it must at least pass the entry points. External references are identified in the object code by their relative symbol identifiers.

The linker proceeds in two phases. First, it determines the absolute address of the start of each object file. The order in which object files are to be loaded is given by the user, either by specifying parameters when the loader is run or by creating a load map file that gives the order in which files are to be placed in memory. Given the order in which files are to be placed in memory and the length of each object file, it is easy to compute the absolute starting address of each file. At the start of the second phase, the loader merges all symbol tables from the object files into a single, large table. It then edits the object files to change relative addresses into absolute addresses. This is typically performed by having the assembler
write extra bits into the object file to identify the instructions and fields that refer to labels. If a label cannot be found in the merged symbol table, it is undefined and an error message is sent to the user.

**Question 12** VHDL synthesizer sometimes produces an error like “...all logic was removed from the design...”. What does it mean?

**Question 13** Simulate the example program in Modelsim. How many clock cycles does it take for $\mu311.1$ to execute this program?
**Figure 30:** Example assembly program 1 for $\mu311.1$.

```
.org 0x0000
sub c,c,c
L:  add c,c,a
dec b
jnz L
halt
```

**Figure 31:** Example assembly program 2 for $\mu311.1$.

```
.org 0x0000
movi a,0x00
movi b,0x01
movi c,0x02
movi e,0x1e
movi g,0xff
write @a,b
inc a
write @a,c
inc a
L:  add f,b,c
write @a,f
inc a
not g
jz L3
mov b,f
L2:  dec e
jnz L
halt
L3:  mov c,f
jmp L2
```
```c
uint16 clear(uint16 *x, uint16 size)
{
    register i=0;
    while(size--)*(x+i++)=0;
}

main()
{
    clear(0,10);
}
```

Figure 32: C program 3 for μ311.1.

```assembly
.org 0x0000
movi h,0xff
mov sp,h
movi a,0x00
push a
movi a,0x0A
push a
call clear
pop a
pop a
halt
clear: mov b,sp
    inc b
    inc b
    read c,@b
    inc b
    read d,@b
    movi e,0x00
L: write @c,e
    inc c
    dec d
    jnz L
ret
```

Figure 33: Example assembly program 3 for μ311.1.
```c
uint16 *ram_ptr=0x0000;
isr0()
{
    uint16 *io_ptr=0x0140;
    *ram_ptr++ = *(io_ptr+1)-(*io_ptr);
}
main()
{
    while(1);
}
```

Figure 34: C program 4 for µ311.1.

```assembly
.org 0x0000

; sp initialization. sp=0x00ff
movi a,ffh
mov sp,a

; initialize register b as a pointer to RAM. b=0x0000
sub b,b,b

; main function
main: jmp main

; interrupt service routine
.org 0x000f
movi d, 0xa0
sl d ; d=0x0140
read e,@d
inc d
read f,@d
sub a,f,e
write @b,a
inc b
ret
```

Figure 35: Example assembly program 4 for µ311.1.
; Example assembly program
.org 0x0000
.equ stack 0xff
.equ size 0x08

; boot code
movi a, stack
mov sp, a
sub d,d,d
mov e,d
movi c, size
jmp _main

.org 0x000F

; isr0 code
L: write @d,e
inc d
dec c
jnz L
ret

_main: mov h,d
movi g,0xAA
write @g,h
jmp _main
halt

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td>0xff</td>
</tr>
<tr>
<td>size</td>
<td>0x08</td>
</tr>
<tr>
<td>L</td>
<td>15</td>
</tr>
<tr>
<td>_main</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 36: Assembling (First Pass) Generating symbol table.
input line [address]:machine code
----------------------- -------------------------
; Example assembly program
.org 0x0000
.equ stack 0xff
.equ size 0x08
; boot code
movi a stack

[0]: 1011000011111111

mov sp a

[1]: 1011100000000000

sub d d d

[2]: 0001001101101100

mov e d

[3]: 0000010001100000

movi c size

[4]: 1011001000001000

jmp _main

[5]: 0101100000001110

.org 0x000F

; isr0 code

L write @d e

[15]: 1010000001110000

inc d

[16]: 0011001101100000

dec c

[17]: 0011101001000000

jnz L

[18]: 0110111000000100

ret

[19]: 0111100000000000

_main mov h d

[20]: 0000011101100000

movi g 0xAA

[21]: 1011011010101010

write @g h

[22]: 1010000010111100

jmp _main

[23]: 0011100000000000

halt

[24]: 1000100000000000

Figure 37: Assembling (Second Pass) Generating object code. The left box shows the output of the assembler. The right box shows the .hex output file that can be directly placed in the program memory.
8 Instruction Pipelining

Pipelining, a standard feature in RISC processors, is much like an assembly line. Because the processor works on different steps of the instruction at the same time, more instructions can be executed in a shorter period of time.

A useful method of demonstrating this is the laundry analogy. Let’s say that there are four loads of dirty laundry that need to be washed, dried, and folded. We could put the first load in the washer for 30 minutes, dry it for 40 minutes, and then take 20 minutes to fold the clothes. Then pick up the second load and wash, dry, and fold, and repeat for the third and fourth loads. Supposing we started at 6 PM and worked as efficiently as possible, we would still be doing laundry until midnight. However, a smarter approach to the problem would be to put the second load of dirty laundry into the washer after the first was already clean and whirling happily in the dryer. Then, while the first load was being folded, the second load would dry, and a third load could be added to the pipeline of laundry. Using this method, the laundry would be finished by 9:30.

μ311.1’s execution consists of 3 stages: fetch, decode and execution cycle. At first glance, a pipelining in μ311.1 would have this form: To apply pipelining to μ311.1, we may need additional registers.

<table>
<thead>
<tr>
<th>fetch</th>
<th>decode</th>
<th>execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch</td>
<td>decode</td>
<td>execute</td>
</tr>
<tr>
<td>fetch</td>
<td>decode</td>
<td>execute</td>
</tr>
</tbody>
</table>

Figure 38: Pipelining in μ311.1.

μ311.1 has single-cycle operations and this makes pipelining easier. The only command that may complicate pipelining is jnz. If a jump occurs during the execution of jnz, then pipelining mechanism must take into account this and start to fetch from the new location.

Question 14 Modify μ311.1 architecture to perform pipelining (we can call the modified microprocessor as μPabs). Reconstruct the next-state table of the control unit given in 3. Modify the VHDL codes and simulate μPabs in Modelsim. Please notice the execution time difference.

Question 15 For μ311.1, find out a formula to calculate the execution time of any given program.

Question 16 Is it possible to perform context switching in μ311.1?
References

Index

address decoding, 47
ALU, 24, 25
assembly, 52

bidirectional buffer, 23
BNF for VHDL, 62
buffer, 23
bus communication, 47
bus cycles, 36
clock, 3
control unit, 34
datapath, 17
hierarchical design, 70

I/O, 47
instruction register, 19
instruction set, 10
internal diagram of $\mu$311.1, 71
interrupt, 48
interrupt cycle, 38, 48

linking, 52

memory read cycle, 37
memory write cycle, 38
microprocessor, 34
multiplexer, 21
nonvolatile memory, 5

opcode, 10
opcode fetch cycle, 37
operation code, 10

program counter, 18
program memory, 5
programming, 51
RAM, 7
register, 18
register file, 20
reset, 3
ROM, 5

schematic, 71
stack, 34
stack pointer, 19
structural design, 70
testbench, 46
unidirectional buffer, 23
VHDL, 62
volatile memory, 7
A Simulation in ModelSim PE Student Edition
B  BNF Syntax for VHDL

abstract_literal ::= decimal_literal | based_literal
access_type_definition ::= access subtype_indication
actual_designator ::= expression | signal_name | variable_name | file_name | open
actual_parameter_part ::= parameter_association_list
actual_part ::= actual_designator | function_name ( actual_designator ) | type_mark ( actual_designator )
adding_operator ::= + | - | &
aggregate ::= ( element_association { , element_association } )
alias_declaration ::= alias alias_designator [ : subtype_indication ] is name
alias_designator ::= identifier | character_literal | operator_symbol
allocator ::= new subtype_indication | new qualified_expression
architecture_body ::= architecture identifier of entity_name is architecture_declarative_part begin architecture_statement_part end [ architecture ]
architecture_declarative_part ::= { block_declarative_item }
architecture_identifier of entity_name is architecture_declarative_part begin
architecture_statement_part end [ architecture ]
array_type_definition ::= unconstrained_array_definition | constrained_array_definition
association_element ::= association_element , association_element
attribute_declaration ::= attribute identifier : type_mark ;
attribute_designator ::= attribute_simple_name
attribute_name ::= prefix [ signature ] ' attribute_designator [ ( expression ) ]
attribute_specification ::= attribute attribute_designator of entity_specification is expression ;
base ::= integer
base_specifier ::= B | O | X
based_integer ::= extended_digit { [ underline ] extended_digit }
based_literal ::= base # based_integer [ . based_integer ] # [ exponent ]
basic_character ::= basic_graphic_chara...
<table>
<thead>
<tr>
<th>attribute_specification</th>
<th>configuration_specification</th>
<th>disconnection_specification</th>
<th>use_clause</th>
<th>group_template_declaration</th>
<th>group_declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>block_declarative_part :=</td>
<td>{ block_declarative_item }</td>
<td>block_header ::=</td>
<td>[ generic_clause</td>
<td>[ generic_map_aspect ; ]</td>
<td>[ port_map_aspect ; ]</td>
</tr>
<tr>
<td>instantiation_specification</td>
<td>instantiation_list : component_name</td>
<td>concurrent_statement ::=</td>
<td>block_statement</td>
<td>process_statement</td>
<td>concurrent_procedure_call_statement</td>
</tr>
<tr>
<td></td>
<td>block_declarative_part ::=</td>
<td>block_statement ::=</td>
<td>block_label :</td>
<td>[ guard_expression ] [ is ]</td>
<td>[ label : ] [ postponed ] procedure_call ;</td>
</tr>
<tr>
<td></td>
<td>[ block_declarative_item }</td>
<td>block_header ::=</td>
<td>[ generic_clause</td>
<td>[ generic_map_aspect ; ]</td>
<td>[ port_clause</td>
</tr>
<tr>
<td></td>
<td>component_configuration ::=</td>
<td>block_specification ::=</td>
<td>architecture_name</td>
<td>[ block_statement_label</td>
<td>[ block_statement_label [ ( index_specification ) ]</td>
</tr>
<tr>
<td></td>
<td>case_statement ::=</td>
<td>block_statement ::=</td>
<td>block_label :</td>
<td>block [ ( guard_expression ) ] [ is ]</td>
<td>block [ block_label ] ;</td>
</tr>
<tr>
<td></td>
<td>when choices =&gt;</td>
<td>case_statement ::=</td>
<td>[ case_label : ]</td>
<td>[ case_label ] ;</td>
<td>end block [ block_label ] ;</td>
</tr>
<tr>
<td></td>
<td>[ simple_expression</td>
<td>case_statement ::=</td>
<td>[ case_label ] ;</td>
<td>[ case_label ] ;</td>
<td>end case [ case_label ] ;</td>
</tr>
<tr>
<td></td>
<td>discrete_range</td>
<td>case_statement ::=</td>
<td>[ case_label ] ;</td>
<td>[ case_label ] ;</td>
<td>end case [ case_label ] ;</td>
</tr>
<tr>
<td></td>
<td>element_simple_name</td>
<td>character_literal ::=</td>
<td>' graphic_character '</td>
<td>component_configuration ::=</td>
<td>use_clause</td>
</tr>
<tr>
<td></td>
<td>others</td>
<td>character_literal ::=</td>
<td>' graphic_character '</td>
<td>component_configuration ::=</td>
<td>group_declaration</td>
</tr>
<tr>
<td></td>
<td>choices ::= choice {</td>
<td>choice</td>
<td>component_configuration ::=</td>
<td>use_clause</td>
<td></td>
</tr>
<tr>
<td></td>
<td>component_configuration ::=</td>
<td>component_configuration ::=</td>
<td>[ binding_indication ;</td>
<td>[ configuration_declarative_part</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for component_specification</td>
<td>component_configuration ::=</td>
<td>[ block_configuration ]</td>
<td>configuration_identifier of entity_name is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[ binding_indication ; ]</td>
<td>component_configuration ::=</td>
<td>end for ;</td>
<td>configuration_declarative_part</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[ block_configuration ]</td>
<td>component_configuration ::=</td>
<td>block_configuration</td>
<td>block_configuration</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end for ;</td>
<td>component_configuration ::=</td>
<td>[ configuration_declarative_item ]</td>
<td>configuration_specification ::=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>component_configuration ::=</td>
<td>component_configuration ::=</td>
<td>[ component_simple_name ]</td>
<td>for component_specification binding_indication ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>component identifier [ is ]</td>
<td>component_configuration ::=</td>
<td>[ local-generic_clause ]</td>
<td>configuration_specification ::=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[ local-port_clause ]</td>
<td>component_configuration ::=</td>
<td>[ local_port_clause ]</td>
<td>for component_specification binding_indication ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end component [ component_simple_name ] ;</td>
<td>component_configuration ::=</td>
<td>[ local-port_clause ]</td>
<td>for component_specification binding_indication ;</td>
<td></td>
</tr>
</tbody>
</table>
constant_declaration ::= constant identifier_list : subtype_indication [ := expression ] ;

constrained_array_definition ::= array index_constraint of element_subtype_indication

constraint ::= range_constraint | index_constraint

context_clause ::= { context_item }

decimal_literal ::= integer [ . integer ] [ exponent ]

declaration ::= type_declaration | subtype_declaration | object_declaration | interface_declaration | alias_declaration | attribute_declaration | component_declaration | group_template_declaration | group_declaration | entity_declaration | configuration_declaration | subprogram_declaration | package_declaration | delay_mechanism ::= transport [ reject time_expression ] inertial

design_file ::= design_unit { design_unit }

design_unit ::= context_clause library_unit

designator ::= identifier | operator_symbol
direction ::= to | downto
disconnection_specification ::= disconnect guarded_signal_specification after time_expression ;

discrete_range ::= discrete_subtype_indication | range

element_association ::= [ choices => ] expression

element_declaration ::= element_declaration_list : element_subtype_definition

element_subtype_definition ::= subtype_indication

entity_aspect ::= entity aspect

entity_access ::= entity_access_list [ ( architecture_identifier ) ]

entity_class ::= entity | architecture | configuration
type ::= type | subtype | constant

signal ::= signal | variable | component

label ::= label | literal | units
group ::= group | file
designer ::= entity_designator

designer ::= entity_designator
designer ::= others
designer ::= all

element_specification ::= element_specification

element_name ::= element_name

element_name ::= open

element_class ::= entity_class
enumeration_type_definition ::= ( enumeration_literal { , enumeration_literal } )

exit_statement ::= [ label : ] exit [ loop_label ] [ when condition ] ;

exponent ::= E [ + ] integer | E - integer

expression ::= relation { and relation } | relation { or relation } | relation { xor relation } | relation { nand relation } | relation { xnor relation }

extended_digit ::= digit | letter

extended_identifier ::= graphic_character { graphic_character } \ 
\ graphic_character { graphic_character } \\
\ factor ::= 
\ primary [ ** primary ] \\
\ abs primary \\
\ not primary

file_declaration ::= file_identifier_list : subtype_indication 
file_open_information ;

file_logical_name ::= string_expression

file_open_information ::= [ open file_open_kind_expression ] is 

file_logical_name

file_type_definition ::= file of type_mark

floating_type_definition ::= range_constraint

generic_designator ::= generic_name

| port_name

| parameter_name

generic_map_aspect ::= generic map ( generic_association_list )

group_constituent ::= name | character_literal

guarded_signal_specification ::= guarded_signal_list : type_mark

if_statement ::= [ if_label : ] if condition then sequence_of_statements 

{ elsif condition then sequence_of_statements }

[ else sequence_of_statements ]

end if [ if_label ] ;

incomplete_type_declaration ::= type identifier ;

index_constraint ::= ( discrete_range { , discrete_range } )

index_subtype_definition ::= type_mark range <>

instantiate_unit ::= [ component ] component_name

indexed_name ::= prefix ( expression { , expression } )

instantiated_unit ::= 

[ block_declarative_item ]

instantiation_list ::=
<table>
<thead>
<tr>
<th>type_declaration</th>
<th>subprogram_body</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype_declaration</td>
<td>type_declaration</td>
</tr>
<tr>
<td>constant_declaration</td>
<td>subtype_declaration</td>
</tr>
<tr>
<td>signal_declaration</td>
<td>constant_declaration</td>
</tr>
<tr>
<td>shared_variable_declaration</td>
<td>signal_declaration</td>
</tr>
<tr>
<td>file_declaration</td>
<td>shared_variable_declaration</td>
</tr>
<tr>
<td>alias_declaration</td>
<td>file_declaration</td>
</tr>
<tr>
<td>component_declaration</td>
<td>alias_declaration</td>
</tr>
<tr>
<td>attribute_declaration</td>
<td>component_declaration</td>
</tr>
<tr>
<td>attribute_specification</td>
<td>attribute_declaration</td>
</tr>
<tr>
<td>disconnection_specification</td>
<td>attribute_specification</td>
</tr>
<tr>
<td>use_clause</td>
<td>disconnection_specification</td>
</tr>
<tr>
<td>group_template_declaration</td>
<td>use_clause</td>
</tr>
<tr>
<td>group_declaration</td>
<td>group_template_declaration</td>
</tr>
<tr>
<td>package_declarative_part ::=</td>
<td>group_declaration</td>
</tr>
<tr>
<td>{ package_declarative_item }</td>
<td></td>
</tr>
<tr>
<td>parameter_specification ::=</td>
<td></td>
</tr>
<tr>
<td>identifier in discrete_range</td>
<td>parameter_specification</td>
</tr>
<tr>
<td>physical_literal ::= [ abstract_literal ] unit_name</td>
<td>physical_literal</td>
</tr>
<tr>
<td>physical_type_definition ::=</td>
<td>physical_type_definition</td>
</tr>
<tr>
<td>range_constraint</td>
<td>physical_type_definition</td>
</tr>
<tr>
<td>units</td>
<td>range_constraint</td>
</tr>
<tr>
<td>base_unit_declaration</td>
<td>units</td>
</tr>
<tr>
<td>{ secondary_unit_declaration }</td>
<td>base_unit_declaration</td>
</tr>
<tr>
<td>end units</td>
<td>{ secondary_unit_declaration }</td>
</tr>
<tr>
<td>physical_type_simple_name</td>
<td>end units</td>
</tr>
<tr>
<td>port_clause ::=</td>
<td>physical_type_simple_name</td>
</tr>
<tr>
<td>port ( port_list ) ;</td>
<td>port_clause</td>
</tr>
<tr>
<td>port_list ::= port_interface_list</td>
<td>port</td>
</tr>
<tr>
<td>port_map_aspect ::=</td>
<td>port_list</td>
</tr>
<tr>
<td>port_map ( port_association_list )</td>
<td>port_map_aspect</td>
</tr>
<tr>
<td>prefix ::=</td>
<td>port_map</td>
</tr>
<tr>
<td>name</td>
<td>prefix</td>
</tr>
<tr>
<td></td>
<td>name</td>
</tr>
<tr>
<td>function_call</td>
<td></td>
</tr>
<tr>
<td></td>
<td>function_call</td>
</tr>
<tr>
<td>qualified_expression</td>
<td></td>
</tr>
<tr>
<td>type_conversion</td>
<td></td>
</tr>
<tr>
<td></td>
<td>type_conversion</td>
</tr>
<tr>
<td>allocator</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>( expression )</td>
<td>allocator</td>
</tr>
<tr>
<td>record_type_definition ::= record</td>
<td>record</td>
</tr>
<tr>
<td></td>
<td>record</td>
</tr>
<tr>
<td>type_mark ' ( expression )</td>
<td>record_type_definition</td>
</tr>
<tr>
<td></td>
<td>type_mark '</td>
</tr>
<tr>
<td>aggregate</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>element_declaration</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>secondary_unit_declaration ::= identifier = physical_literal ;</td>
<td>element_declaration</td>
</tr>
<tr>
<td></td>
<td>secondary_unit_declaration</td>
</tr>
<tr>
<td>architecture_body</td>
<td></td>
</tr>
<tr>
<td></td>
<td>architecture_body</td>
</tr>
<tr>
<td>package_body</td>
<td></td>
</tr>
<tr>
<td></td>
<td>package_body</td>
</tr>
<tr>
<td>subprogram_declaration</td>
<td></td>
</tr>
<tr>
<td></td>
<td>subprogram_declaration</td>
</tr>
</tbody>
</table>

Tolga Ayav
selected_name ::= prefix . suffix

selected_signal_assignment ::= with expression select
  target <= options selected_waveforms ;

selected_waveforms ::= { waveform when choices , }
waveform when choices

sensitivity_clause ::= on sensitivity_list

sensitivity_list ::= signal_name { , signal_name }

sequence_of_statements ::= { sequential_statement }

sequential_statement ::= wait_statement
| assertion_statement
| report_statement
| signal_assignment_statement
| variable_assignment_statement
| procedure_call_statement
| if_statement
| case_statement
| loop_statement
| next_statement
| exit_statement
| return_statement
| null_statement

shift_expression ::= simple_expression [ shift_operator simple_expression ]

simple_expression ::= [ sign ] term { adding_operator term }

term ::= factor { multiplying_operator factor }

factor ::= ( type_conversion )
| timeout_clause ::= for time_expression

type_conversion ::= type_mark ( expression )

type_declaration ::= full_type_declaration
| incomplete_type_declaration

full_type_declaration ::=

incomplete_type_declaration ::= [ return type_mark ]

subprogram_body ::= subprogram_specification is
  subprogram_declarative_part
begin
  subprogram_statement_part
end [ subprogram_kind ] [ designator ] ;

subprogram_specification ::= procedure designator [ ( formal_parameter_list ) ]
return type_mark

subprogram_declarative_part ::= { subprogram_declarative_item }

subprogram_declarative_item ::= subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| use_clause
| group_template_declaration
| group_declaration
| null_statement

procedure_designator [ ( formal_parameter_list ) ]
return type_mark

subprogram_statement_part ::= { sequential_statement }

subprogram_kind ::= procedure | function

subprogram_specification ::=

subprogram_declarative_item ::= subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| use_clause
| group_template_declaration
| group_declaration
| null_statement

signal_assignment_statement ::= [ label : ] target <= [ delay_mechanism ] waveform ;

type_conversion ::= type_mark ( expression )

type_identifier is subtype_indication ;

subprogram_declarative_item ::= subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| use_clause
| group_template_declaration
| group_declaration
| null_statement

subprogram_declarative_part ::= { subprogram_declarative_item }

subprogram_kind ::= procedure | function

subprogram_specification ::=

subprogram_declarative_item ::= subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| use_clause
| group_template_declaration
| group_declaration
| null_statement

signal_identifier_list : subtype_indication

signal_kind ::= register | bus

signal_list ::= signal_name { , signal_name }

signal_name { , signal_name }

| others

| all

| aggregate

| character_literal

| operator_symbol

| all

| name

| aggregate

| term :

| factor { multiplying_operator factor }

| timeout_clause ::= for time_expression

| type_conversion ::= type_mark ( expression )

| incomplete_type_declaration
type_definition ::= 
  scalar_type_definition 
  | composite_type_definition 
  | access_type_definition 
  | file_type_definition 

unconstrained_array_definition ::= 
  array ( index_subtype_definition { , index_subtype_definition } ) 
  of element_subtype_indication 

use_clause ::= 
  use selected_name { , selected_name } 

variable_assignment_statement ::= 
  [ label : ] target := expression ;

variable_declaration ::= 
  [ shared ] variable identifier_list : 
  subtype_indication [ := expression ] ;

wait_statement ::= 
  [ label : ] wait [ sensitivity_clause ] 
  [ condition_clause ] [ timeout_clause ] ;

waveform ::= 
  waveform_element { , waveform_element } 
  | unaffected 

waveform_element ::= 
  value_expression [ after time_expression ] 
  | null [ after time_expression ]
C Implementation Hierarchy

![Diagram of implementation hierarchy]

TESTBENCH  
CLOCK  
RESET  

DATAPATH  
REGISTERFILE  
MUX  

CONTROLUNIT  
CONTROLLER  
BUFFER BUFFER2  
ADD SUB16  

RAM1024  
OPCODES  
U311.1  

SHIFTER ALU  
FA  
MUX  

LE  
AE  
LE  

AE  
FA  
LE  

ADD SUB16  
MUX2  
MUX4  
ALU  

as311 Assembler

as311 translates assembly files to \( \mu 311.1 \) machine code.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define KNRM "\x1B[0m"
#define KRED "\x1B[31m"

FILE *fp=NULL,*fp2=NULL;
char cmd[1023][5][30];
unsigned pc[1023], pcnt=0;
unsigned cnt=0,romcnt=0;
unsigned base=0;

enum opcodes{mov,add,sub,and,or,not,inc,dec,sr,sl,rr,jmp,jz,jnz,call,ret,nop,halt,push,pop,write,read,movi,movspr,movrsp};
enum registers{a,b,c,d,e,f,g,h};

union{

  unsigned short int instruction; /* 16-bit instruction */
  struct { /* R type instruction */
    int u: 2;
    int r3: 3;
    int r2: 3;
    int r1: 3;
    int opcode: 5;
  }r;

  struct { /* J type instruction */
    int add: 10;
    int sign: 1;
    int opcode: 5;
  }j;

  struct { /* I type instruction */
    int imm8: 8;
    int r1: 3;
    int opcode: 5;
  }i;

}x;

int find_label(char *s,unsigned current)
{
  int i;
  for(i=0; i<cnt; i++)
  {
    if(!strcmp(cmd[i][0],s))
    {
      return pc[i]-pc[current]-1;
    }
  }
  return 0;
}

int find_num(char *s)
{
  int i;
  for(i=0; i<cnt; i++)
  {
    if(!strcmp(cmd[i][1],".equ"))
    {
      if(!strcmp(cmd[i][2],s))
      {
        return strtol(cmd[i][3],NULL,16);
      }
    }
  }
  return strtol(s,NULL,16);
}
```

Tolga Ayav
void print_binary(unsigned short k, int end, int begin)
{
    int i;
    for(i=end-1;i>=begin;i--) printf("%u",(k>>i)&0x1);
}

unsigned int assign_reg(char *s)
{
    if(s[0]=='@'){s[0]=s[1]; s[1]=0;}
    if(s[0]=='s') return 0;
    else return s[0]-97;
}

int main(int argc, char **argv)
{
    char s[100],s2[100];
    char *ch=NULL;
    unsigned char sgn=0;
    int i,j,k,opr=0;
    memset(cmd,0,120000);
    memset(s,0,100);memset(s2,0,100);
    if(argc<2) {printf("Usage: as311 filename\n"); return 0;}
    if((fp=fopen(argv[1],"r"))==NULL) {printf("%s cannot be opened\n",argv[1]); return 0;}
    // FIRST PASS //
    while(fgets(s,100,fp)){
        i=j=opr=0;
        while(*(s+i)!=0){
            if(*(s+i)=='\n') continue;
            else if(cmd[cnt][0]=='.'||cmd[cnt][0]=='!') printf("%s\n",cmd[cnt]);
            else if(cmd[cnt][0]!=0 && j!=0 && *(s+i)==' '||*(s+i)==',||*(s+i)=='\n'||*(s+i)=='\t'))
            {*(s2+j)=0; j=0; strcpy(cmd[cnt][2+opr++],s2); if(opr==5)break; memset(s2,0,100);}
            else *(s2+j++)=*(s+i);
            i++;
        }
        if(s[0]=='\n') continue;
        if(cmd[cnt][1][0]=='.'||cmd[cnt][1][0]=='!') printf("%s\n",cmd[cnt][1]);
        if(!strcmp(cmd[cnt][1],".org")) {
            if(strtol(cmd[cnt][2],NULL,16)>=(base+pcnt)) base=strtol(cmd[cnt][2],NULL,16);
            else {printf("Error in .org directive.\n"); return -1;}
            pcnt=0;
        }
        npc=0;
        if(cmd[cnt][1][0]==0)cnt++;
        memset(s2,0,100);
    }
    fclose(fp); fp=fopen(strcat(strtok(argv[1],"."),".vhdl_hex"),"w");
    fp2=fopen(strcat(strtok(argv[1],"."),".hex"),"w");
    fprintf(fp,"constant ROM: rom_type := (\n"),
    fprintf(fp,"input line\t\t[address]:machine code\n-------------------\t-------------------------\n");
    // SECOND PASS //
    for(i=0; i<cnt; i++) {
        x.instruction=0;
        for(j=0; j<S; j++) if(cmd[i][j][0]==0) printf("%s\n",cmd[i][j]);
    }
}

Tolga Ayav 78 / 84
if(!strcmp(cmd[i][1],"mov") && cmd[i][2][0]=='s') {
    x.r.opcode=movspr; x.r.r1=assign_reg(cmd[i][2]); x.r.r2=assign_reg(cmd[i][3]);
} else if(!strcmp(cmd[i][1],"mov") && cmd[i][3][0]=='s') {
    x.r.opcode=movrsp; x.r.r1=assign_reg(cmd[i][2]); x.r.r2=assign_reg(cmd[i][3]);
} else if(!strcmp(cmd[i][1],"mov")) {
    x.r.opcode=mov; x.r.r1=assign_reg(cmd[i][2]); x.r.r2=assign_reg(cmd[i][3]);
} else if(!strcmp(cmd[i][1],"add")) {
    x.r.opcode=add; x.r.r1=assign_reg(cmd[i][2]); x.r.r2=assign_reg(cmd[i][3]); x.r.r3=assign_reg(cmd[i][4]);
} else if(!strcmp(cmd[i][1],"sub")) {
    x.r.opcode=sub; x.r.r1=assign_reg(cmd[i][2]); x.r.r2=assign_reg(cmd[i][3]); x.r.r3=assign_reg(cmd[i][4]);
} else if(!strcmp(cmd[i][1],"and")) {
    x.r.opcode=and; x.r.r1=assign_reg(cmd[i][2]); x.r.r2=assign_reg(cmd[i][3]); x.r.r3=assign_reg(cmd[i][4]);
} else if(!strcmp(cmd[i][1],"or")) {
    x.r.opcode=or; x.r.r1=assign_reg(cmd[i][2]); x.r.r2=assign_reg(cmd[i][3]);
} else if(!strcmp(cmd[i][1],"not")) {
    x.r.opcode=not; x.r.r1=assign_reg(cmd[i][2]); x.r.r2=assign_reg(cmd[i][3]);
} else if(!strcmp(cmd[i][1],"inc")) {
    x.r.opcode=inc; x.r.r1=assign_reg(cmd[i][2]); x.r.r2=assign_reg(cmd[i][3]);
} else if(!strcmp(cmd[i][1],"dec")) {
    x.r.opcode=dec; x.r.r1=assign_reg(cmd[i][2]); x.r.r2=assign_reg(cmd[i][3]);
} else if(!strcmp(cmd[i][1],"rr")) {
    x.r.opcode=rr; x.r.r1=assign_reg(cmd[i][2]); x.r.r2=assign_reg(cmd[i][3]);
} else if(!strcmp(cmd[i][1],"sl")) {
    x.r.opcode=sl; x.r.r1=assign_reg(cmd[i][2]); x.r.r2=assign_reg(cmd[i][3]);
} else if(!strcmp(cmd[i][1],"sr")) {
    x.r.opcode=sr; x.r.r1=assign_reg(cmd[i][2]); x.r.r2=assign_reg(cmd[i][3]);
} else if(!strcmp(cmd[i][1],"jmp")) {
    x.j.opcode=jmp; k=find_label(cmd[i][2],i); if(k<0) x.j.sign=1; x.j.add=abs(k);
} else if(!strcmp(cmd[i][1],"jz")) {
    x.j.opcode=jz; k=find_label(cmd[i][2],i); if(k<0) x.j.sign=1; x.j.add=abs(k);
} else if(!strcmp(cmd[i][1],"jnz")) {
    x.j.opcode=jnz; k=find_label(cmd[i][2],i); if(k<0) x.j.sign=1; x.j.add=abs(k);
} else if(!strcmp(cmd[i][1],"call")) {
    x.j.opcode=call; k=find_label(cmd[i][2],i); if(k<0) x.j.sign=1; x.j.add=abs(k);
} else if(!strcmp(cmd[i][1],"ret")) { x.j.opcode=ret; }
else if(!strcmp(cmd[i][1],"push")) { x.r.opcode=push; x.r.r3=assign_reg(cmd[i][2]);
} else if(!strcmp(cmd[i][1],"pop")) { x.r.opcode=pop; x.r.r1=assign_reg(cmd[i][2]);
} else if(!strcmp(cmd[i][1],"write")) {
    x.r.opcode=write; x.r.r2=assign_reg(cmd[i][2]); x.r.r3=assign_reg(cmd[i][3]);
} else if(!strcmp(cmd[i][1],"read")) {
    x.r.opcode=read; x.r.r1=assign_reg(cmd[i][2]); x.r.r2=assign_reg(cmd[i][3]);
} else if(!strcmp(cmd[i][1],"movi")) {
    x.i.opcode=movi; x.i.r1=assign_reg(cmd[i][2]); x.i.imm8=find_num(cmd[i][3]);
}

if(!strcmp(cmd[i][1][0]==',') && cmd[i][1][0]=='=') {
    print("X\n\n\n\n","pc[i];
    printf("%x",KRED); print_binary(x.instruction,16,11);
    printf("%x",KNRM); print_binary(x.instruction,11,0);
    if(pc[i]>romcnt) { for(j=0; j<(pc[i]-romcnt); j++) fprintf(fp,"X"\"%.4x\",\"n",0); fprintf(fp2,"%.4x\n",0); }
    romcnt=pc[i];
} else if(pc[i]==romcnt) { fprintf(fp,"X"\"%.4x\", -- "\",x.instruction); fprintf(fp2,"%.4x\n",x.instruction);
    for(j=0; j=5; j++) if(cmd[i][j][0]==0) fprintf(fp,"%s ",cmd[i][j]); fprintf(fp,"\n"); romcnt++;
    }
### E Multitasking in \( \mu 311.1 \)

![Connecting program memory, 1K RAM and an 16-bit timer to \( \mu 311.1 \).](image)

#### E.1 16-bit timer

The 16-bit timer is connected to \( \mu 311.1 \) as an I/O device. It has two registers R0 and R1 both of which are readable and writable. R1 is a down counter. In each clock cycle, it counts down by 1 and when it reaches to zero, R1 is reloaded with R0, generating and interrupt signal. Thus, the frequency of this timer interrupt is determined by R0 (For example, for \( T_{CLK} = 1\mu s \) and R0=10000, the interrupt period would be 10ms). As seen in Figure 39, R0 and R1 are accessed at the addresses 0400h and 0401h respectively.

```vhdl
-- timer16.vhd: 16-bit timer
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity timer16 is port(
    addr : in std_logic_vector(0 downto 0);
    data : inout std_logic_vector(15 downto 0);
    cs : in std_logic;
    wr : in std_logic;
    rd : in std_logic;
    clk : in std_logic; -- clock.
    int : out std_logic; -- interrupt
    inta : in std_logic);
end timer16;
```

Tolga Ayav
architecture description of timer16 is

subtype cell is std_logic_vector(15 downto 0);
type ram_type is array(0 to 1023) of cell;
signal RF: ram_type;

begin

process(cs,addr)
begin
    if (cs='0' and rd='1') then
        data <= RF(conv_integer(addr));
    elsif (cs='0' and wr='1') then
        RF(conv_integer(addr)) <= data;
    else
        data <= (others => 'Z');
    end if;
end process;

process(clk,inta)
begin
    if rising_edge(clk) then
        RF(1) <= RF(1) - 1;
        if RF(1) = X"0000" then
            RF(1) <= RF(0);
            int <= '1';
        end if;
    end if;
    if (inta='1') then
        int <= '0';
        data <= "ZZZZZZZZZZZZZZ00";
    else
        data <= (others => 'Z');
    end if;
end process;
end description;

; Scheduling two tasks
.org 0x0000
.equ stack 0xff
.equ size 0x08
; boot code
    movi a, stack
    mov sp, a
    movi d, 0x80
    sl d
sl d
sl d
; d=0x0400
movi e, 0xc8
; interrupt period 200us
write @d, e
jmp _main
.org 0x000F
; isr0 code
; context switching
push a
push b
push e
push h
movi e, 0xc8
mov h,sp
inc h
read a, @h
read b, @e
write @h, b
write @e,a
pop h
pop e
pop b
pop a
ret
_main: call L
L: pop a
movi b,0x6
add a,a,b
write @e,a
jmp _task2
_task1: sub a,a,a
L1: inc a
jmp L1
_task2: sub b,b,b
L2: dec b
jmp L2
F  \( \mu 311.1 \) Internal Schematic