Design Notes of Microprocessor

\( \mu \text{P}_{\text{abs}} \)

Student’s Handbook
(Ver. 08.a)

Asst. Prof. Dr. Tolga AYAV

Department of Computer Engineering
Izmir Institute of Technology

September 2008
Contents

1 A Simple Microprocessor: μP_abs
   1.1 Instruction Set ................................................. 4
   1.2 Datapath ....................................................... 5
      1.2.1 Registers ................................................ 5
      1.2.2 ALU ....................................................... 6
      1.2.3 Internal Program Memory ................................. 7
   1.3 Control Unit .................................................. 8

2 Software ...................................................... 11
   2.1 High-Level Programming ...................................... 11
   2.2 Assembly and Linking ....................................... 12

3 Instruction Pipelining .......................................... 17
   A Test Bench and Processor ...................................... 19
   B Datapath ................................................................ 21
   C ALU ..................................................................... 23
   D Registers ................................................................ 25
   E Internal Program Memory (ROM) ............................... 27
   F Multiplexers, Addsub circuitry, Full Adder ............... 29
   G Control Unit ....................................................... 31
   H Simulation ........................................................... 33
Preface

This handbook is intended to provide the students of Computer Engineering in Izmir Institute of Technology with understanding of the basic principles of digital logic design and assistance in the related courses in the curriculum:

1. CENG214, Logic Design
2. CENG311, Computer Architecture
3. CENG314, Embedded Computer Systems
4. CENG383, Real-Time Systems
5. CENG384, Microprocessors
6. CENG451, Advanced Digital System Design
7. CENG452, Building Software for Embedded Systems
8. CENG563, Real-Time and Embedded System Design

The main purpose of this document in fact is to give the students some intuition about the following question:

**Question 1** What is really happening inside a computer system?

In other words, starting from typing `printf("value:%d", *p);` (we know C programming very well), we must understand compiling, assembling, linking, loading the machine code, execution the machine code on the processor and how processors execute this code: from the functions of logic circuits to even the movement of electrons in the transistors of integrated circuits. This is a very long way to learn and these topics are covered in at least 8-10 undergraduate courses in computer and electronics programs.

This document aims to give a very short and abstract answer to the above question. It starts from designing a simple microprocessor called $\mu P_{abs}$. Design of a microprocessor can be seen as an ultimate in digital logic design efforts. VHDL codes of $\mu P_{abs}$ are also given in appendices so that students can better understand the functions of each part of the microprocessor and they can even realize the processor on a FPGA board. Compiling, assembling and linking are introduced shortly to give the complete understanding of the whole computer system.

Although it is not the aim of this book to cover the aforementioned topics completely, students may still find many things they are wondering are missing, too short or incomplete. Nonetheless, I hope this will be a good starting point for their deeper research as well as their study of computer architecture.

Tolga AYAV
03/09/2008
Chapter 1

A Simple Microprocessor: $\mu P_{\text{abs}}$

$\mu P_{\text{abs}}$ is a simple 8-bit processor. Since it has an internal program memory and I/O ports, we can call it microcontroller as well. The specifications of $\mu P_{\text{abs}}$ are:

- 18 pins: 16 I/O, Reset and Clock pins
- 8-bit internal address and data buses
- 256x8 bit internal program memory
- 32x8 bit register file
- 8-bit input and 8-bit output ports
- 8 instructions with single cycle operation

A general diagram of $\mu P_{\text{abs}}$ showing its input and output pins is given in figure 1.1.

Figure 1.2 demonstrates the behavior of $\mu P_{\text{abs}}$ using a GCL-like (Guarded Command Language) pseudo-code (See [NN92] for the description of GCL). In this code, $\text{ir}$, $\text{pc}$ and $\text{acc}$ denote 8-bit instruction register, program counter and accumulator respectively. $\text{imem}[\alpha]$ is the 8-bit value stored in “$\alpha + 1$”th location of the program memory where $\alpha$ is 8-bit address value. $\text{reg}[^\beta]$ denotes “$\beta + 1$”th register where $\beta$ is 5-bit register address. Please note that $\text{acc}$ is a special register and physically same with $\text{reg}[0]$. $\text{ir}$ consists of two parts such that $\text{ir.o}$ holds the most significant 3 bits as opcode and $\text{ir.a}$ holds the least significant 5 bits as an operand such as register address, memory address or immediate depending on the corresponding opcode. $\text{ir.a.s}$, however represents the most significant bit of $\text{ir.a}$ as the sign bit.

$[g_1 \rightarrow a_1 \parallel g_2 \rightarrow a_2 \cdot \cdot \cdot \parallel g_n \rightarrow a_n]$ is a selective construct such that one action among $a_1 \cdot \cdot \cdot a_n$ whose guard is true will be selected and executed. $\ast[\ldots]$ depicts a repetitive construct.
Question 2 Write a simulator for $\mu$P$_{abs}$ in C language (see the pseudo-code given in figure 1.2). 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.

1.1 Instruction Set

$\mu$P$_{abs}$'s limited instruction set has only eight instructions. These commands are given in table 1.1. To encode eight instructions the operation code (opcode) requires 3 bits, giving us eight different combinations. As shown in the encoding column, the three most significant bits represent the opcode of the instructions. For example, the opcode for $\text{sta}$ is 000 and the opcode for $\text{outp}$ is 110 and so on. [PH05]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Operation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sta}$ reg</td>
<td>000rrrr</td>
<td>$F[\text{reg}] \leftarrow \text{Acc}$</td>
<td>store accumulator</td>
</tr>
<tr>
<td>$\text{lda}$ reg</td>
<td>001rrrr</td>
<td>$\text{Acc} \leftarrow F[\text{reg}]$</td>
<td>load accumulator</td>
</tr>
<tr>
<td>movi imm5</td>
<td>010iii</td>
<td>$\text{Acc} \leftarrow \text{imm5}$</td>
<td>move immediate</td>
</tr>
<tr>
<td>inp reg</td>
<td>011rrrr</td>
<td>$F[\text{reg}] \leftarrow \text{input}$</td>
<td>read from input port</td>
</tr>
<tr>
<td>outp reg</td>
<td>100rrrr</td>
<td>$\text{output} \leftarrow F[\text{reg}]$</td>
<td>write to output port</td>
</tr>
<tr>
<td>jnz add5</td>
<td>101saaa</td>
<td>if($A!=0$ and $s=0$) then $\text{PC} \leftarrow \text{PC} + \text{aaaa}$ &lt;br&gt; if($A!=0$ and $s=1$) then $\text{PC} \leftarrow \text{PC} - \text{aaaa}$</td>
<td>jump if $\text{Acc}$ is not zero</td>
</tr>
<tr>
<td>adda reg</td>
<td>110rrrr</td>
<td>$\text{Acc} \leftarrow \text{Acc} + F[\text{reg}]$</td>
<td>summation</td>
</tr>
<tr>
<td>suba reg</td>
<td>111rrrr</td>
<td>$\text{Acc} \leftarrow \text{Acc} - F[\text{reg}]$</td>
<td>subtraction</td>
</tr>
</tbody>
</table>

Notations:
Acc = Accumulator, i.e. F[0]
F[0-31] = 32x8 bits register file
PC = Program counter register
add5 = 5 bits for specifying a memory address
imm5 = 5 bit immediate value
reg = 5 bits for specifying a register address

1.2 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.

1.2.1 Registers

µP_{abs} has three separate registers, program counter (PC), instruction register (IR), output
register (OR) and a register file consisting of 32 general purpose registers.

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.

![Figure 1.3: Program Counter (PC) register and PC Next Logic.](image)

Instruction Register and Output Register

Instruction register (IR) stores the instruction being fetched from the program memory. Output
register holds the value driven from the output port. The structure of these registers are entirely
identical to PC register.
Register File

Register file contains 32 registers. The block diagram of the register file is seen in figure 1.5. For further detail on its working, please see the code given in figure D.2 of appendix D.

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.

1.2.2 ALU

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. $\mu$Pabs’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 1.5. We can define the function of ALU as:

$$F = f(s, A, B)$$

$$F = s'_1s'_0A + s'_1s_0(A + B) + s_1s_0(A + B' + 1)$$

To implement ALU we will use a generic circuit consisting of a set of full adders augmented with arithmetic and logic extenders. As we can see from the figure 1.4, 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 (note that $\mu$Pabs does not have logical operations in fact); whereas, the arithmetic extender (AE) is for manipulating all arithmetic operations. The LE performs the actual 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 arithmetic 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

<table>
<thead>
<tr>
<th>$s_1$</th>
<th>$s_0$</th>
<th>Operation Name</th>
<th>Operation</th>
<th>$x_i$ (LE)</th>
<th>$y_i$ (AE)</th>
<th>$c_0$ (CE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Pass</td>
<td>Pass $A$ to output</td>
<td>$a_i$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Addition</td>
<td>$A + B$</td>
<td>$a_i$</td>
<td>$b_i$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Subtraction</td>
<td>$A - B$</td>
<td>$a_i$</td>
<td>$b'_i$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

![Figure 1.4: Implementation of ALU.](image-url)
circuit labeled CE (for carry extender) is for modifying the primary carry-in signal, \( c_0 \), so that arithmetic operations are performed correctly. Therefore, we can find out:

\[
\begin{align*}
  c_0 &= s_1 \\
  y_i &= s_1 \oplus (s_0 \land b_i) \\
  x_i &= a_i
\end{align*}
\] (1.1)

**Question 4** Find out the formulae given with (1.1), using common digital design techniques such as truth tables, karnaugh maps or other simplification methods. The function of ALU is given in table 1.2.

The implementation of ALU in VHDL can be seen in appendix C.

### 1.2.3 Internal Program Memory

It is the memory where the program code to be executed is stored. In each fetch cycle, one code is fetched from the memory and placed into the instruction register. The VHDL coding of the memory is given in appendix E.
1.3 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 1.7. Figure 1.8 shows the FSM of $\mu P_{ABS}$.

![Figure 1.7: Control Unit.](image)

![Figure 1.8: FSM diagram for the control unit.](image)

**Question 5** Please complete the next-state diagram of the control unit given in the table 1.3 and design the control unit using J-K flip-flops.
### Table 1.3: Next-State Diagram for the Control Unit (incomplete)

<table>
<thead>
<tr>
<th>clk</th>
<th>reset</th>
<th>Acq</th>
<th>IRZ</th>
<th>state</th>
<th>ALUSel</th>
<th>Asel</th>
<th>IRload</th>
<th>Pload</th>
<th>memMap</th>
<th>pipfetch</th>
<th>we</th>
<th>writeAcc</th>
<th>rbe</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note: The table is incomplete and contains placeholders for values.*
Figure 1.9: $\mu$P_{abs}.
Chapter 2

Software

2.1 High-Level Programming

Let’s define the following high-level programming language $C_{abs}$ for $\mu P_{abs}$:

$$S ::= \begin{array}{l}
\text{var } x : \text{int8} \\
\text{var definition} \\
\hline
x := E \\
\text{assignment} \\
\hline
\text{skip} \\
\text{no operation} \\
\hline
\text{read}(x) \\
\text{input read} \\
\hline
\text{write}(x) \\
\text{output write} \\
\hline
S_1; S_2 \\
\text{sequencing} \\
\hline
\text{if } B \text{ then } S_1 \text{ else } S_2 \\
\text{conditional} \\
\hline
\text{while } B \text{ do } S \\
\text{iteration}
\end{array}$$

where

$$E ::= E_1 \text{ op } E_2 \mid x \mid C \text{ op } \in \{+,-\}$$

$$B ::= (x \neq C)$$

int8 depicts the type of 8-bit integer

$C$ is 5-bit constant value.

Let’s write a program for our processor such that we read input port ($n$), find out $\sum_{i=1}^{n} i$ and write the result to the output port at the end. The following would be our program:

```plaintext
var t, n : int8;
t := 0;
read (n);
while n!=0 do
  t := t + n;
n := n - 1;
write (t);
```

Figure 2.1: Example algorithm written in the abstract language

Compilation is another topic and beyond the scope of this document. Please see figure 2.3 for detail. Here, we will assume that a compiler can generate the intermediary assembly code given in figure 2.4.
Question 6 Please note that $C_{abs}$ is very limited language such that it is well suited to the hardware. For example, it supports two mathematical operations and only 5-bit constant values. Discuss if we could use multiplication, i.e., $op \in \{+, -, \ast\}$. How can the compiler translate the following line to the assembly of $\mu P_{abs}$?

$$t := t \ast n;$$

Could we also generalize 5-bit constants to 8-bit? If so, how would you translate the following line to the assembly of $\mu P_{abs}$?

$$\text{if } t! = 231 \text{ then } t := t + 1;$$

Question 7 You can see the basic steps of compilation process in figure 2.3. Try to develop a compiler for our $C_{abs}$ language using lex and yacc tools.

2.2 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 2.2 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 $\mu P_{abs}$, we will do it by hand. The assembly output is seen in figure 2.4.

Question 8 Read the assembly code given in figure 2.4 carefully. What can you say about the maximum value of $n$ in the algorithm? Where does the limitation come from?
2.2. ASSEMBLY AND LINKING

Question 9 Re-write the assembly code given with figure 2.4 in MIPS assembly format.

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.

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.
CHAPTER 2. SOFTWARE

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 \( L1 \) is replaced with “10111”.

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.

After assembling and linking the program, we have the following machine code:

```
011 00001
010 00001
000 00100
010 00000
000 00011
001 00011
110 00001
000 00011
001 00001
111 00010
000 00001
101 10111
100 00011
```

Figure 2.5: Machine code of our example assembly program for \( \mu \text{P}_{\text{abs}} \).

**Question 10** VHDL synthesizer sometimes produces an error like “...all logic was removed from the design...”. What does it mean?

**Question 11** Simulate the example program in Modelsim. How many clock cycles does it take for \( \mu \text{P}_{\text{abs}} \) to execute this program?
Chapter 3

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.

µPabs’s execution consists of 3 stages: fetch, decode and execution cycle. At first glance, a pipelining in µPabs would have this form: To apply pipelining to µPabs, we may need additional registers. µPabs 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 12 Modify µPabs architecture to perform pipelining (we can call the modified microprocessor as µPabs). Reconstruct the next-state table of the control unit given in 1.3. Modify the VHDL codes and simulate µPabs) in Modelsim. Please notice the execution time difference.

Question 13 For both µPabs and µPabs, find out a formula to calculate the execution time of any given program.
Appendix A

Test Bench and Processor

```vhdl
31 entity test is port(
32 --input testbench: in std_logic_vector(7 downto 0);
33 output testbench: out std_logic_vector(7 downto 0));
34 end entity;
35
36 architecture imp of test is
37
38 signal input_signals: std_logic_vector(7 downto 0) := "00000100";
39 signal output_signals: std_logic_vector(7 downto 0);
40 signal clk: std_logic := '0';
41 signal reset: std_logic := '1';
42
43 begin
44
45 process(clk)
46 begin
47 if rising_edge(clk) then
48 output_signals <= input_signals after 5 ns;
49 end process;
50
51 process(reset)
52 begin
53 reset <= '0' after 30 ns;
54 end process;
55
56 process: entity work.uP_abs port map(clk, reset, input_signals, output_signals);
57 end proc;
58
Figure A.1: VHDL code for the test bench.
```

```vhdl
25 library ieee;
26 use ieee.std_logic_1164.all;
27 use ieee.std_logic_unsigned.all;
28 use ieee.numeric_std.all;
29
30 entity uP_abs is port(
31 clk: in std_logic;
32 reset: in std_logic;
33 input: in std_logic_vector(7 downto 0);
34 output: out std_logic_vector(7 downto 0));
35 end uP_abs;
36
37 architecture imp of uP_abs is
38
39 signal d: std_logic_vector(7 downto 0);
40 signal ALORem: std_logic_vector(1 downto 0);
41 signal ARem: std_logic_vector(1 downto 0);
42 signal writeAcc: std_logic;
43
44 signal AReg, IRload, PCload, opfetch, jmpAbs, Oload, we, rbe : std_logic;
45
46 begin
47
48 ControlUnit: controller port map(clk, reset, AReg, IR, ARem, ALORem, AReg, writeAcc, IRload,
49 PCload, Oload, jmpAbs, opfetch, we, rbe);
50 Datapath: datapath port map(clk, reset, input, output, AReg, IR, ALORem, AReg, writeAcc, IRload,
51 PCload, Oload, jmpAbs, opfetch, we, rbe);
52 end imp;
```

Figure A.2: VHDL code for \( \mu P_{abs} \).
Appendix B
Datapath

Figure B.1: VHDL code for datapath.
Appendix C

ALU

Figure C.1: ALU.

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

entity ALU is port(
    I1 : in std_logic_vector(1 downto 0);
    A, B, C : in std_logic_vector(7 downto 0);
    f1 : out std_logic_vector(7 downto 0);
    unsigned_overflow : out std_logic);
end ALU;

architecture imp of ALU is
begin
    signal X, Y : std_logic_vector(7 downto 0);
    signal C : std_logic_vector(7 downto 0);

    begin
        C(0) <= $13;
        Y(0) <= S(1) xor (S(0) and B(0));
        Y(1) <= S(1) xor (S(0) and B(1));
        Y(2) <= S(1) xor (S(0) and B(2));
        Y(3) <= S(1) xor (S(0) and B(3));
        Y(4) <= S(1) xor (S(0) and B(4));
        Y(5) <= S(1) xor (S(0) and B(5));
        Y(6) <= S(1) xor (S(0) and B(6));
        Y(7) <= S(1) xor (S(0) and B(7));

        f1 <= C(0) or (Y(7) or (Y(4) xor (Y(5) and (Y(6) or Y(3) or Y(2) or Y(1) or Y(0))));
    end imp;
```
Appendix D

Registers

Figure D.1: VHDL code for Program Counter (PC) register.

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

entity PC is port(
    clk: in std_logic;
    reset: in std_logic;
    load: in std_logic;
    INPUT: in std_logic_vector(7 downto 0);
    OUTPUT: out std_logic_vector(7 downto 0);
);
end PC;

architecture impl of PC is

component FF is port(
    clk: in std_logic;
    reset: in std_logic; -- reset
    load: in std_logic;
    D: in std_logic;
    Q: out std_logic);
end component;

begin
    U0: FF port map(clk, reset, load, INPUT(0), OUTPUT(0));
    U1: FF port map(clk, reset, load, INPUT(1), OUTPUT(1));
    U2: FF port map(clk, reset, load, INPUT(2), OUTPUT(2));
    U3: FF port map(clk, reset, load, INPUT(3), OUTPUT(3));
    U4: FF port map(clk, reset, load, INPUT(4), OUTPUT(4));
    U5: FF port map(clk, reset, load, INPUT(5), OUTPUT(5));
    U6: FF port map(clk, reset, load, INPUT(6), OUTPUT(6));
    U7: FF port map(clk, reset, load, INPUT(7), OUTPUT(7));
end impl;
```
```vhdl
entity regfile is port(
  clk: in std_logic;
  reset: in std_logic; -- reset
  we: in std_logic; -- write enable
  addr: in std_logic_vector(3 downto 0); -- write and read address [32 registers]
  D: in std_logic_vector(7 downto 0); -- input
  cbe: in std_logic;
  portA: out std_logic_vector(7 downto 0);
  portB: out std_logic_vector(7 downto 0));
end regfile;

architecture imp of regfile is
  subtype reg is std_logic_vector(7 downto 0);
  type regarray is array(0 to 31) of reg;
  signal RF: regarray;
begin
  WritePort: Process(clk, reset)
  begin
    if(clk'event and clk='1') then
      if(reset='1') then
        RF(0) <= (others => '0'); -- register A (accumulator)
        RF(1) <= (others => '0');
        RF(2) <= (others => '0');
        RF(3) <= (others => '0');
      elsif(we='1') then
        RF(conv_integer(addr)) <= D;
      elsif(writeA='1') then
        RF(0) <= D;
      end if;
    end if;
  end process;
  ReadPortB: Process(cbe, addr)
  begin
    if(cbe='1') then
      PortB <= RF(conv_integer(addr));
    else
      PortB <= (others => 'X');
    end if;
  end process;
  ReadPortA: PortA <= RF(0); -- PortA always accumulator register
end imp;
```

Figure D.2: VHDL code for register file.
Appendix E
Internal Program Memory (ROM)

```vhdl
28 PACKAGE opcodes IS
29   --conditions
30   SUBTYPE t_condi IS std_logic_vector (2 DOWNTO 0);
31
32   CONSTANT stb : t_condi := "000"; -- mov A, Reg
33   CONSTANT ldx : t_condi := "001"; -- mov Reg, A
34   CONSTANT movl : t_condi := "010"; -- move 5 bit literal to Reg
35   CONSTANT inp : t_condi := "011";
36   CONSTANT outp : t_condi := "100";
37   CONSTANT jst : t_condi := "101";
38   CONSTANT adda : t_condi := "110";
39   CONSTANT suba : t_condi := "111";
40
41   -- register names
42   SUBTYPE t_reg IS std_logic_vector (4 DOWNTO 0);
43
44   CONSTANT A : t_reg := "00000";
45   CONSTANT B : t_reg := "00001";
46   CONSTANT C : t_reg := "00010";
47   CONSTANT D : t_reg := "00011";
48   CONSTANT E : t_reg := "00100";
49
50 END opcodes;
```

Figure E.1: Opcode definitions.
APPENDIX E. INTERNAL PROGRAM MEMORY (ROM)

Figure E.2: Program memory.

```vhdl
31 entity rom_256_8 is port(
32   cs: in std_logic;
33   addr: in std_logic_vector(7 downto 0);
34   data: out std_logic_vector(7 downto 0);)
35 end rom_256_8;
36
37 architecture imp of rom_256_8 is
38   subtype cell is std_logic_vector(7 downto 0);
39   type rom_type is array(0 to 255) of cell;
40
41 constant ROM: rom_type :=
42   (16 bits of ROM contents)
43   movi & 00001,
44   sta & C,
45   movi & 00000,
46   sta & B,
47   lda & D,
48   adda & B,
49   sta & B,
50   lda & B,
51   suba & C,
52   sta & B,
53   jnz & 10111,
54   outp & D,
55   others => others => '0'
56   );
57
58 begin
59   process(cs)
60     begin
61       if(cs = '1') then
62         data <= ROM(conv_integer(addr));
63       else
64         data <= (others => '0');
65       end if;
66     end process;
67 end imp;
```

Figure E.2: Program memory.
Appendix F

Multiplexers, Addsub circuitry, Full Adder

Figure F.1: 2x1 4-bit Multiplexer.

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

entity mux2 is port(
  s: in std_logic;
  x0, x1: in std_logic_vector(3 downto 0);
  y: out std_logic_vector(3 downto 0);
);

architecture imp of mux2 is
begin
  process(s, x0, x1)
  begin
    if(s = '0') then
      y <= x0;
    else
      y <= x1;
    end if;
  end process;
end imp;
```

Figure F.2: 4x1 8-bit Multiplexer.

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

entity mux4 is port(
  s: in std_logic;
  x0, x1, x2, x3: in std_logic_vector(7 downto 0);
  y: out std_logic_vector(7 downto 0);
);

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 <= (others => '0');
    end case;
  end process;
end imp;
```
entity addsub_pc is port |
  &: in std_logic_vector(7 downto 0); |
  B: in std_logic_vector(7 downto 0); |
  F: out std_logic_vector(7 downto 0); |
  sub: in std_logic; |
end addsub_pc;

architecture imp of addsub_pc is |
begin |
process(A, B, sub) |
begin |
if{sub='0'} then |
F <= A + B; |
else |
F <= A - B; |
end if; |
end process; |
end imp;

Figure F.3: 8-bit addsub circuit.

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

entity FA is port |
  cin: in std_logic; |
  carryOut: out std_logic; |
  x, y: in std_logic; |
  s: out std_logic; |
end FA;

architecture imp of FA is |
begin |
m <= x xor y xor cin; |
carryOut <= (x and y) or (carryIn and (x xor y)); |
end imp;

Figure F.4: Full Adder.
Appendix G

Control Unit

```vhdl
entity controller is port(
  clk: in std_logic;
  reset: in std_logic;
  -- status signals
  keep: in std_logic;
  to: in std_logic_vector(1 downto 0);
  -- control signals
  aladdr: out std_logic_vector(1 downto 0);
  alr: out std_logic_vector(1 downto 0);
  wraddr: out std_logic;
  EBl août: out std_logic;
  PI realizing: out std_logic;
  OCI: out std_logic;
  PGM: out std_logic;
  op_w: out std_logic;
  oe: out std_logic;
  cbe: out std_logic;
); end controller;

architecture map of controller is
begin
  type state_type is (a_start, a_fetch, a_decode, a_ins, a_mem, a_rst, a_rdy, a_store, a_load, a_revo);
  signal state: state_type := a_start;
  signal count: std_logic_vector(1 downto 0) := "00";
  begin
    NEXT_STATE: process(reset, clk) begin
      if (reset='1') then
        state := a_start;
      elsif (count='1') then
        state := a_fetch;
      else
        state := a_decode;
      end if;
      if (count='0') then
        count <= count + 1;
      end if;
      case state is
        when a_start => state <= a_fetch;
        when a_fetch => state <= a_decode;
        when a_decode =>
          case (to(1 downto 0)) is
          when "00" => state <= a_load;
          when "01" => state <= a_rdy;
          when "10" => state <= a_mem;
          when "11" => state <= a_store;
          when others => state <= a_fetch;
        end case;
      end case;
    end process;
end map;
```

Figure G.1: Control Unit.
Appendix H

Simulation

Figure H.1: Simulation in Modelsim.
Bibliography

