College of Engineering and Computer Science

Australian National University

ENGN3213 / ENGN6213

Digital Systems & Microprocessors

CLAB 4: Understanding the MU0 Microprocessor

V3.0

Contents

1 CLAB 4: Understanding the MU0 Microprocessor

1.1 Aims

In this prelab we are going to explore MU0 - the flagship **Register Transfer Level (RTL)** system in the course. This work will be supplemented by the work on state machines dealt with in lectures this week and will give you excellent background material for study.

The aims are,

- Analyse an RTL system
- Study the component parts of MU0's datapath
- Use GTKWAVE to simulate MU0
- Learn to program in a simple machine language

1.2 A Description of the MU0 Microprocessor

MU0 is a 16 bit machine with a 12 bit address space. Instructions are 16 bits long with a 4 bit opcode and a 12 bit address field. With a 4 bit opcode there can be at most 16 commands. MU0 consists of a **controller FSM** which controls its operation and a **datapath** for data processing.

1.2.1 Datapath

The datapath of MU0 is shown in Fig, 1. The essential idea of a datapath is that a number of blocks are wired together with a bunch of switches connecting to the controller. The controller operates by flipping the switches at the right moment.

Exercise 1. MU0's datapath consists of a number of sub-blocks that should be familiar to you by now. There are

- **An arithmetic logic unit:** $alu16.v$
- **Two data multiplexers, x-mux and y-mux both modelled by** $mux16.v$**: a 16-bit mux.**
- **An address multiplexer, a-mux modelled by** $mux12.v$**: a 12-bit mux.**
- **Two 16-bit registers for the prograsm counter (pc) and the instruction register (ir) modelled by** $vreg16.v$

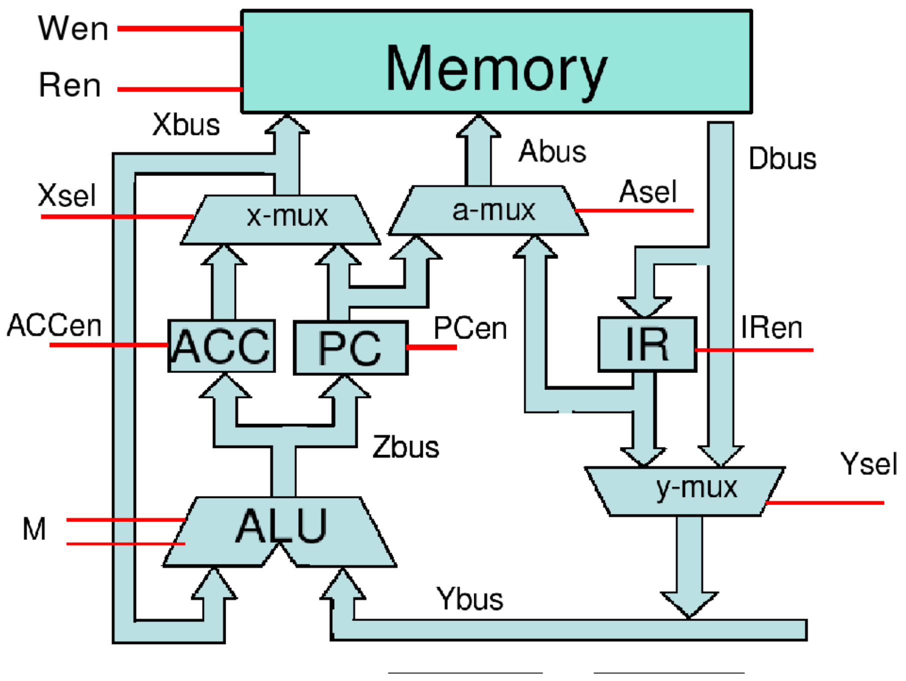


Figure 1: MU0 datapath.

Download MU0.zip from the web and go to the folder *testbenches*. Run ICARUS VERILOG simulations for each of the above and make sure you understand how they work.

1.2.2 The Controller

The circuit schematic of the controller is shown in Fig. 2. It has two inputs: *state* and the 4 most significant bits of the instruction register which contain the opcode of the instruction to be executed. At its output are the values of the switches to control the datapath.

The first task of the controller is to run the instruction cycle,

- Fetch the next command from memory and store the command in the IR
- Decode and execute the command

Fig 3 shows the instruction cycle. The FSM has just two states which are executed on consecutive clock cycles.

The second task of the controller is to understand what to do next based on the opcode of the cuurrent MU0 assembly language instruction. The MU0 instruction set is shown in Fig. 4.

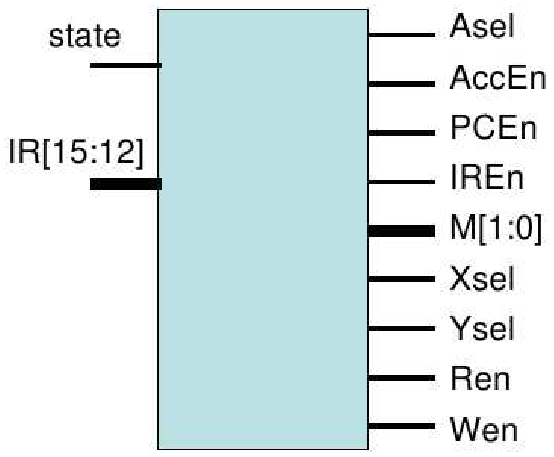


Figure 2: MU0 Control Path FSM Implementation

1.3 MU0 Operation

The first thing to do is to understand what goes on with these instructions. Note the syntax of the commands. The symbol S refers to a memory address. The notation [S] refers to the *contents* stored at memory addressed.

Consider the datapath of Fig. 1. It shows the following hardware systems,

1. A **program counter register** (PC) which stores the address in the memory of the current instruction. Exactly what is the current instruction and what is the next instruction we'll see in a minute. The addresses count from 0 upwards and in any program, the instructions are stored in the first contiguous memory locations while the data is store in the subsequent locations. This is the basis of the Von Neumann architecture wherein program and data are stored sequentially in memory.
2. An **instruction register** (IR) which contains the instruction while it is being executed.
3. An **accumulator** (ACC) which provides intermediate storage of data during instruction execution. The ACC is sometimes referred to as the **working register**.
4. **An Arithmetic Logic Unit** (ALU)
5. Several multiplexers

In MU0 the data has 16 bits and the memory has storage locations that are 16 bits wide. The data in memory is stored at locations that can be located by their address. These

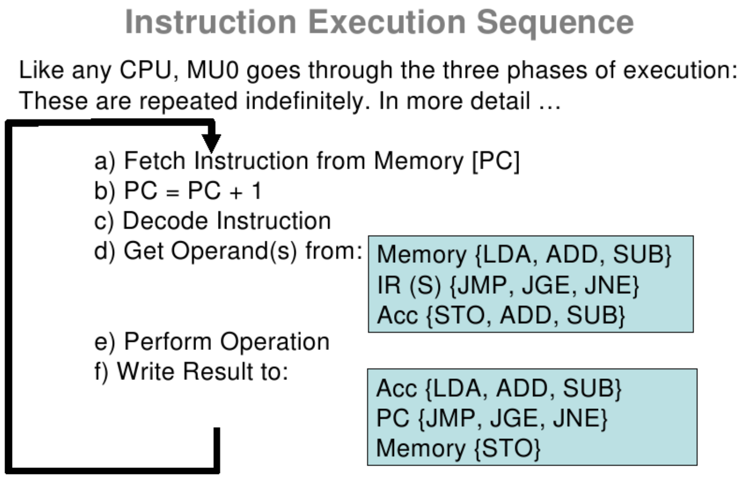


Figure 3: MU0 instruction cycle

addresses are represented by words that are 12 bits wide. That is MU0's memory has 2^{12} memory locations where data can be stored.

It is interesting and entirely pertinent to note that an instruction word consists of 16 bits and can therefore be stored in memory. The most significant four bits ([15:12] in VERILOG parlance) of the instruction word is referred to as the **opcode**. This is the machine language symbol that represents an instruction. This is the hex number **F** in the left most column of Fig. 4. There are 16 possible opcodes but only 8 are implemented in MU0. The meanings of the instructions are also described in Fig. 4. The remaining 12 bits in the instruction word is the address in memory of either the operand that the instruction operates on (in the case of LDA, ADD and SUB) or the destination of the data in the ACC (STO) or the address of the next instruction in the case of the JUMP commands (JMP, JGE, JNE).

Fig. 5 shows the instruction word format.

The program counter is incremented every instruction. It is only controlled by the active edges of the clock. Consequently MU0 automatically runs sequentially through the addresses in memory. Reading an instuction occurs when that instruction appears in the IR in the EXEC state.

The cunning in the design of the MU0 architecture is that each hardware block in the datapath of Fig 1 is configured to execute these instructions by appropriate changes in their control inputs. **This is our standard model of a datapath**. Examples of controls are **PCen (enable PC), ACCen (enable the ACC), Asel (choose the input that connects to the output of the address MUX, a-mux), M (choose the function to be performed by the ALU), etc**. The bit values of these controls are the outputs of the **control path FSM**.

MU0 Instruction Set

F	Mnemonic		Description
0	LDA	S	Acc := [S]
1	STO	S	[S] := Acc
2	ADD	S	Acc := Acc + [S]
3	SUB	S	Acc := Acc – [S]
4	JMP	S	PC := S
5	JGE	S	If Acc >= 0, PC := S
6	JNE	S	If Acc ≠ 0, PC := S
7	STP		Stop

Figure 4: MU0 assembly language instructions

Op code	**12 bit operand address**

Figure 5: MU0 instruction format.

1.3.1 The Control Path Finite State Machine

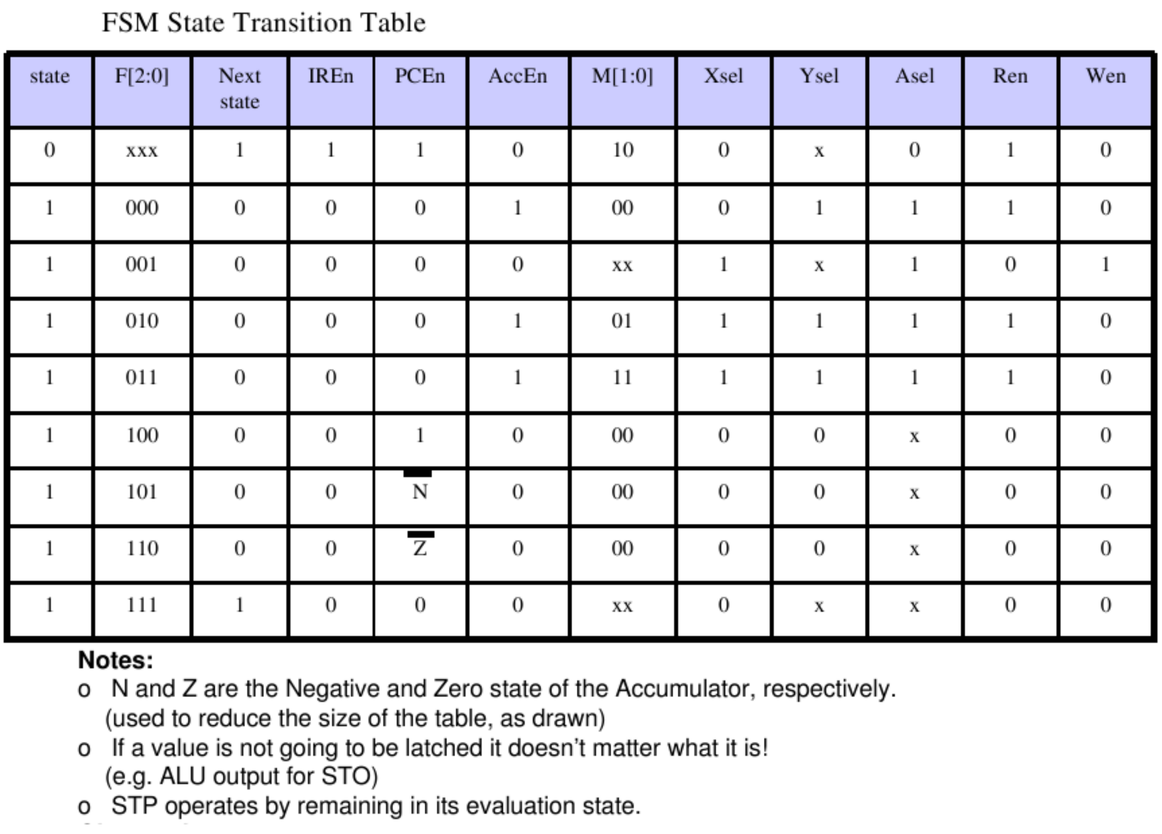
The **next state diagram** of the controller in Fig. 6 describes how the FSM works. The first column shows the states of which there are just two,

0 **FETCH** (fetch instruction and store instruction in IR) and,

1 **EXEC** (decode instruction in the IR and execute instruction).

The second column shows the opcode, F. The opcode is MU0's only input. MU0 obtains the opcode when the controller FSM reads bits [15:12] of the IR. The third column is the next state that the controller jumps to. Note that the opcode is not needed in the FETCH state (**0**) because in this state the only steps are to mux the PC contents onto the address bus through the a-mux and to set up the ALU input control, M, for a PC increment. As a result, regardless of the F or opcode value, the next state is EXEC (**1**). This explains the dont cares, "XXX" in the F cell in the table.

If you look at the next-state diagram you should be able to confirm the interconnects of MU0 in Fig. 7 for the FETCH state. The grey tracks indicate connected paths in the datapath.



FSM State Transition Table

state	F[2:0]	Next state	IREn	PCEn	AccEn	M[1:0]	Xsel	Ysel	Asel	Ren	Wen
0	xxx	1	1	1	0	10	0	x	0	1	0
1	000	0	0	0	1	00	0	1	1	1	0
1	001	0	0	0	0	xx	1	x	1	0	1
1	010	0	0	0	1	01	1	1	1	1	0
1	011	0	0	0	1	11	1	1	1	1	0
1	100	0	0	1	0	00	0	0	x	0	0
1	101	0	0	$\overline{N}$	0	00	0	0	x	0	0
1	110	0	0	$\overline{Z}$	0	00	0	0	x	0	0
1	111	1	0	0	0	xx	0	x	x	0	0

Notes:
- N and Z are the Negative and Zero state of the Accumulator, respectively. (used to reduce the size of the table, as drawn)
- If a value is not going to be latched it doesn't matter what it is! (e.g. ALU output for STO)
- STP operates by remaining in its evaluation state.

Figure 6: MU0 next state diagram

1.3.2 MU0 in action

Try and follow the following verbal description. Remember that all registers (PC, IR and ACC) and state transitions occur on the positive edge of the clock but memory read and writes occur on negative clock transitions.

The sequence of events that occur in the FETCH state from the first positive transition of the system clock are as follows.

1. The FETCH cycle occurs at the first positive edge of the clock.
2. In the FETCH state the a-mux input is connected to the PC output. The MUX is a combinational device and so the PC contents should already be pointing to the address of the next instruction in memory.
3. At tbe ensuing negative clock transition the memory transfers the contents of the location whose address is in the PC to the Dbus. The Dbus is the output data line of the memory whereas the Xbus is the input data line.
4. The contents of the Dbus are now present at the input to the IR.

5. In the FETCH state the PC contents are also pointing at the ALU input through the x-mux. The ALU M-value is set so that the ALU increments the value on this input. Since both the x-mux and the ALU are combinational devices, the PC incremented contents are transferred instantaneously at the PC input. On the next positive clock transition the contents of the PC will be incremented ready for the next time the FSM is in the FETCH state.

From the second positive clock transition we are in the EXEC state. The sequence of events that occur in the this state are as follows.

1. At this transition the PC increments its contents as discussed previously.
2. The IR registers the contents of the Dbus to its output.
3. The controller reads the [15:12] bits (the opcode) from the IR.
4. Depending on the opcode value, several function controls in the datapath may be enabled or disabled as follows.
    - If the opcode is LDA then the ACC is enabled and the y-mux is set so that the Dbus is connected to the ALU input on the Ybus. The ALU M value is set for a through connnection on its Ybus input. At the positive edge of the next clock transition into the FETCH state, the ACC output will store the contents of the Dbus.
    - If the opcode is for ADD or SUB then the ACC is enabled and the Dbus is again connected to the ALU via the Ybus through y-mux. The x-mux is set to allow the contents of the ACC onto the Xbus and the ACC M value is set for ADD or SUB. On the subsequent negative clock transition the contents of the memory is transferred onto the Dbus. At the positive edge of the next clock transition into the FETCH state, the ACC output will store the sum of its previous value and that in the memory location.
    - If the opcode is STO, the x-mux places the contents of the ACC on the input to the Xbus which is also the memory input data line. The last 12 bits of the contents of the IR are sent via the a-mux to the address bus of the memory. On the next negative clock transition the memory stores the contents of the Xbus (the contents of the ACC).
    - In the case of the JUMP instructions, the last 12 bits of the instruction register are sent via the y-mux to the Ybus and the ALU. The ALU is set for straight through so that this new memory address is fed to the PC. At the ensuing posedge of clock (FETCH) the PC is changed to the address which is the operand of the JUMP instruction.

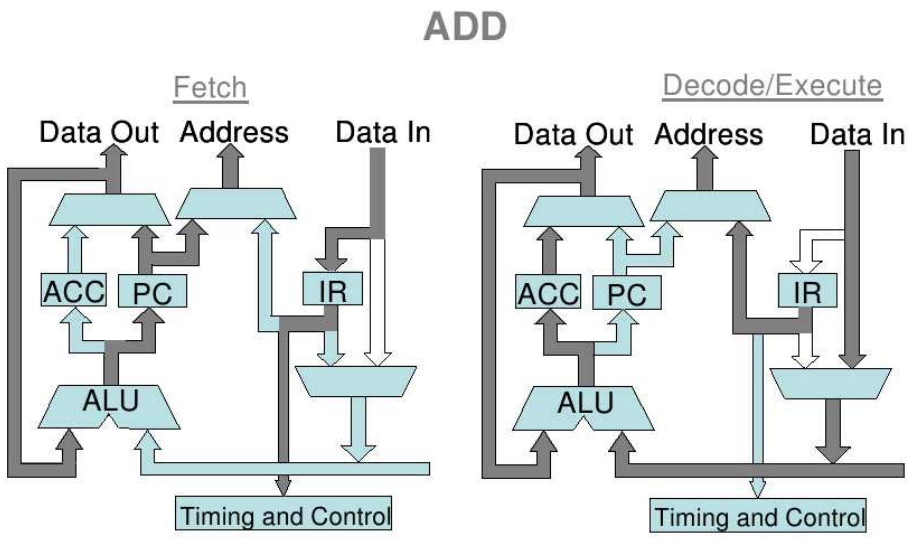


Figure 7: The MU0 datapath interconnects during FETCH and EXEC.

1.3.3 Running a Program on MU0

The following is a machine code listing of a MU0 program which adds the contents (000A) of memory location 4 to the contents (0001) of memory location 5. The first column consists of the addresses in memory in hex format. The second column contains the contents of the memory locations which are the machine language commands to be executed. The first hex digit (most significant 4 bits) in each command is the opcode. These are 0 (LDA), 2 (ADD), 1 (STO). The remaining 3 hex digits are the address operands as discussed above.

```
0  0004  (load (LDA) the contents of memory adddress 4 into the ACC)
1  2005  (add (ADD) the contents of memory address 5 to that in the ACC)
2  1006  (store (STO) the contents of the ACC in memory location 6)
3  7000  (STOP)
4  000A  (data stored in memory location 4)
5  0001  (data stored in memory location 5)
6  0000  (data stored in memory location 6)
```

Notice how execution occurs in purely sequential fashion. MU0 does not know which memory addresses contain instructions and which data. Its proper operation depends entirely on proper programming and the march of the PC contents. The STOP command terminates execution and prevents the processor from trying to perform a false opcode in the first hex digit of the data at memory location 4.

Notice that for simulation a memory controller has been used that reads the file *prog.lst* from the disk which contains the machine language commands.

Fig 8 shows the complete GTKWAVE output from running MU0 with ICARUS VERILOG.

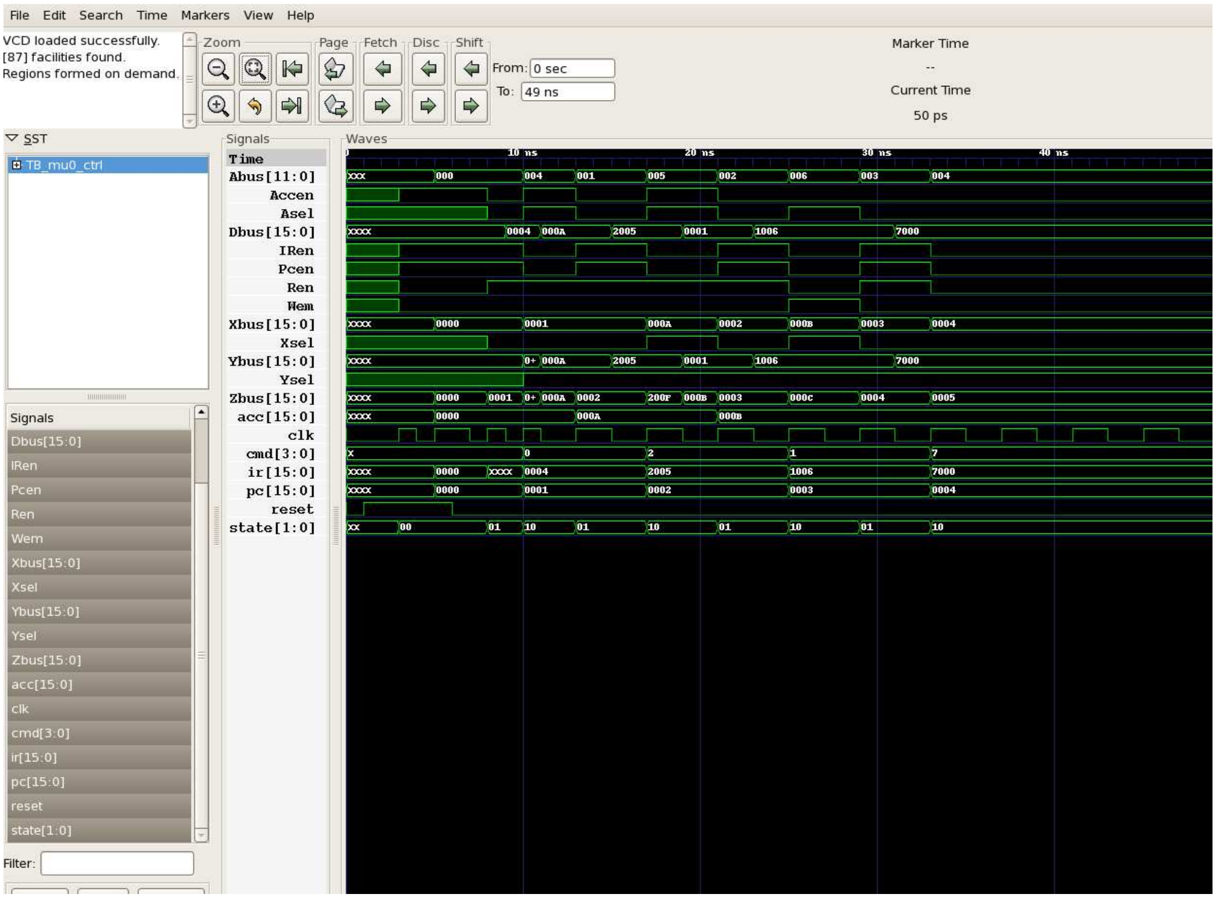

VCD loaded successfully.
[87] facilities found.
Regions formed on demand.

Figure 8: GTKWAVE traces of the MU0 data during execution of the above program

Fig. 9 expands the traces around the FETCH and EXEC states when the instruction 2005 is being executed. For instruction 2005 the PC is pointing to address 1 in memory. During this instruction the contents of memory address 5 (0001) is added to the contents of the accumulator which is by now 000A. Notice that the actual instruction 2005 does not appear in the IR until the EXEC state is reached and that the contents of the ACC do not register the sum, 000B, until the FETCH cycle of the following instruction.

Exercise 2: Run MU0 for the above program (stored in prog.lst) and try to understand the outputs.

1.3.4 MU0 Assembly Language?

The lexical commands in Fig. 4, LDA, STO, etc are referred to as assembly language instructions. Normally when writing programs for a microprocessor one only has to use

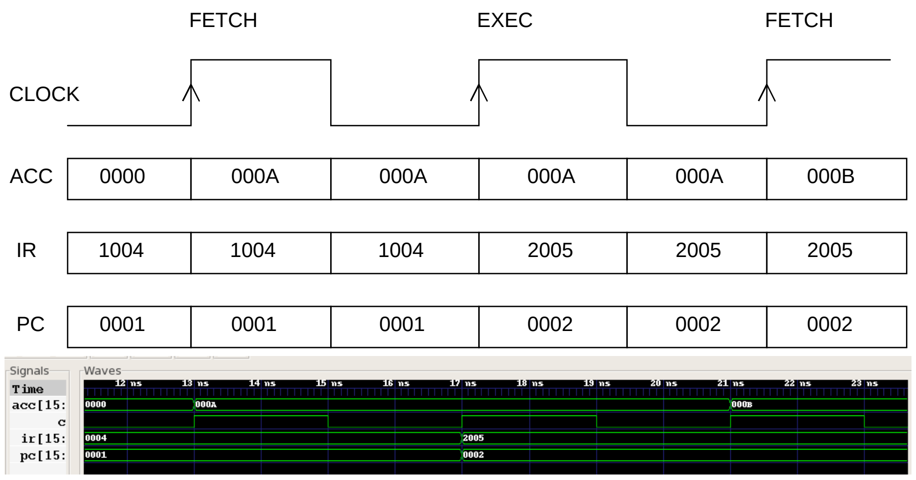


Figure 9: Expected and GTKWAVE traces of MU0 ACC, IR and PC registers around the execution of the 2005 instruction

these commands and some variables representing the data. This is clearly much easier to read than the column of numbers that form the machine code. However use of assembly language presumes the existence of an assembly language compiler or assembler for short which translates tbe assembly language into machine code. Unfortunately (to the best of my knowledge) MU0 does not have an assembler written for it (though you may be attempted to write one in JAVA or C, I dont think it would be difficult).

1.4 Programming Exercises

In this section we are go get some practice writing MU0 programs.

Exercise 3. Describe the function of the following MU0 program. (Be careful that there are enough time steps available in the simulation to complete the execution of the progam.

```
0006
3007
1006
0006
5000
7000
0004
0001
```

Exercise 4. Write a MU0 program to drive a siren to sound at an audio pitch of 2kHz with a duty cycle of approximately 1 Hz. Assume that the CPU clock is 50 MHz. N.B. There is no mechanism to output a signal to an actual siren

in MU0. However in Exercise 5 next, we will add some io ports to MU0 to make this possible. For now just implement the on and off phases of the siren as two nested loops.

Exercise 5. Download and study Q5(3) of the 2009 exam. On the basis of this example, add support for *inport* **and** *outport* **commands to read and write respectively an 8 bit word between an IO port and the accumulator by (a) drawing the schematic of the new datapath and (b) adding the necessary** *case* **statements to the controller. Hint: assume that the physical access to the port is via the address space of MU0 at memory location** $12'hfff$.