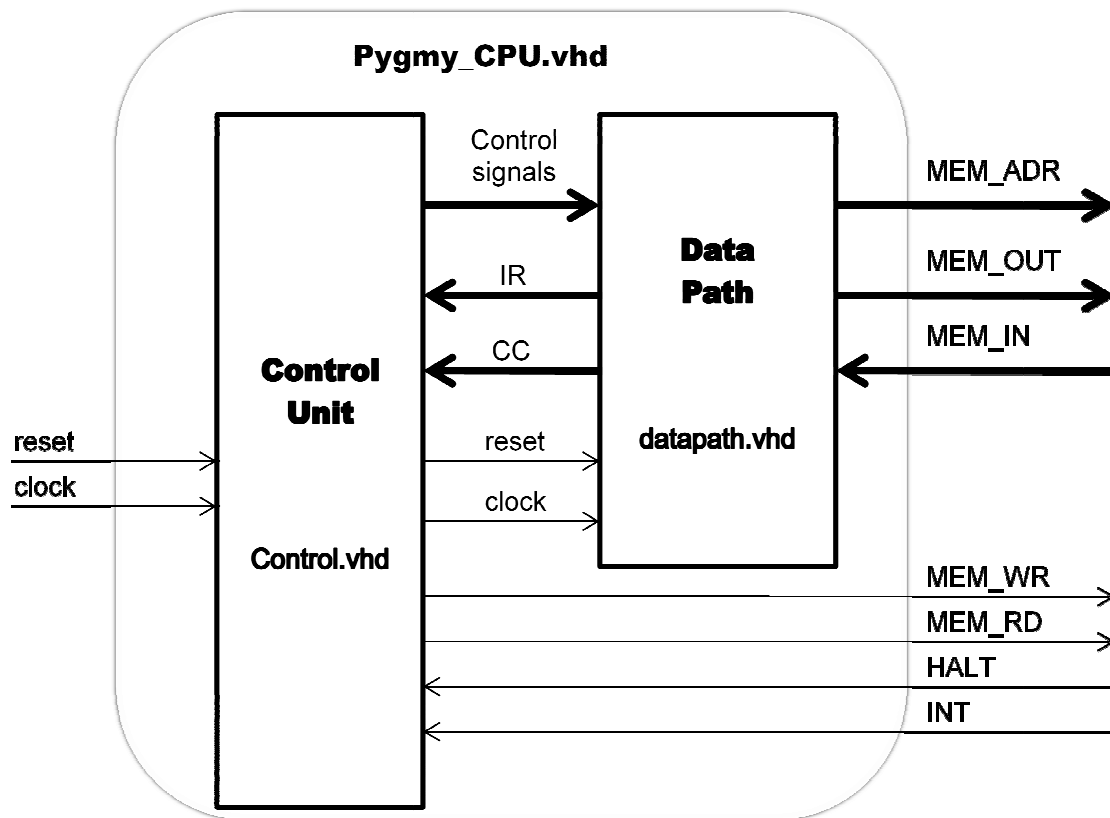


1. General Descriptions

The documentation herein describes the Pygmy CPU designed for the teaching purpose. This CPU is used in conjunction with the ELE405 Laboratory assignments. It will also serve as a design example for the semester project. However, you will receive only the level-0 realization of the Pygmy CPU initially. In each lab you will be asked to realize more instructions to make the current lab works. The appendices present the complete instruction set architecture (ISA) of the Pygmy CPU.

2. Instruction Set Architecture (ISA) Design



Here are the ISA components for this CPU:

1. 32-bit data bus (actually two buses; one for each direction, since we are using the on-chip memory for main memory).
2. 32-bit address bus; word addressable; Memory mapped I/O addresses.

3. Control bus includes: “mem_wr” and “mem_rd” for read/write control of memory or I/O; “halt” for interfacing slower memory; and “int” for hardware interrupt.
4. Four general purpose registers: R0, R1, R2 and R3.
5. Eight instructions have been implemented initially (See Appendix A for the complete list).
6. Load/Store architecture (operands for ALU operations are from registers only).
7. Five instruction formats (see below).
8. Two addressing modes: immediate and inherent (register indirect is supported in full version).
9. Only “z” flag or one condition code was implemented. (other condition codes (CC): N, V and C are also in full version)
10. Support only unsigned words (The full implementation supports 2’s complement words).

2.1. Instruction Set

The instruction set of the initial pygmy CPU is not a complete set. You will be asked to add instructions to this CPU in the laboratory assignments. The Pygmy CPU is consistent with the traditional load/store architecture in which only load and store instructions are allowed to access main memory. Arithmetic and logical operations are performed exclusively on the register contents.

<i>Assembly</i>	<i>Operations</i>	<i>Instruction format</i>	<i>Opcode</i>	<i>Z</i>
LD Rx, \$M	$R_x \leftarrow \text{MEM}(\$M)$	Opcode (4) ; x (2) ; empty (2) ; M (24)	0001	N
LDI Rx, #I	$R_x \leftarrow \#I$	Opcode (4) ; x (2) ; empty (2) ; I (24)	0011	N
ST Rx, \$M	$\text{MEM}(\$M) \leftarrow R_x$	Opcode (4) ; x (2) ; empty (2) ; M (24)	0100	N
ADD Rx, Ry	$R_x \leftarrow R_x + R_y$	Opcode (4) ; x (2) ; y (2) ; alu (4) ; empty (20)	0110	Y
AND Rx, Ry	$R_x \leftarrow R_x \text{ and } R_y$	Opcode (4) ; x (2) ; y (2) ; alu (4) ; empty (20)	0110	Y
XOR Rx, Ry	$R_x \leftarrow R_x \text{ xor } R_y$	Opcode (4) ; x (2) ; y (2) ; alu (4) ; empty (20)	0110	Y
ROR Rx, #Ry	$R_x \leftarrow R_x \text{ ror by } R_y$	Opcode (4) ; x (2) ; y (2) ; alu (4) ; empty (20)	0110	Y
JNZ \$M	$\text{PC} \leftarrow \$M \text{ if } Z=0$	Opcode (4) ; mode (4) ; M (24)	1000	N

NOTE1: \$ indicates address and # denotes data.

NOTE2: x and y are register numbers ranging from 0 to 3.

NOTE3: Opcode: bits 31-28, x: bits 27-26, y: bits 25-24, mode: bits 27-24 (for JNZ), M or I: bits 23-0.

NOTE4: The 24-bit M is appended with eight 0’s at MSB to form a 32-bit address

NOTE5: The 24-bit I is appended with eight 0's to MSB form a 32-bit data (unsigned number).

NOTE6: Only ALU operations will affect the condition code (z).

2.2. Micro-Operations

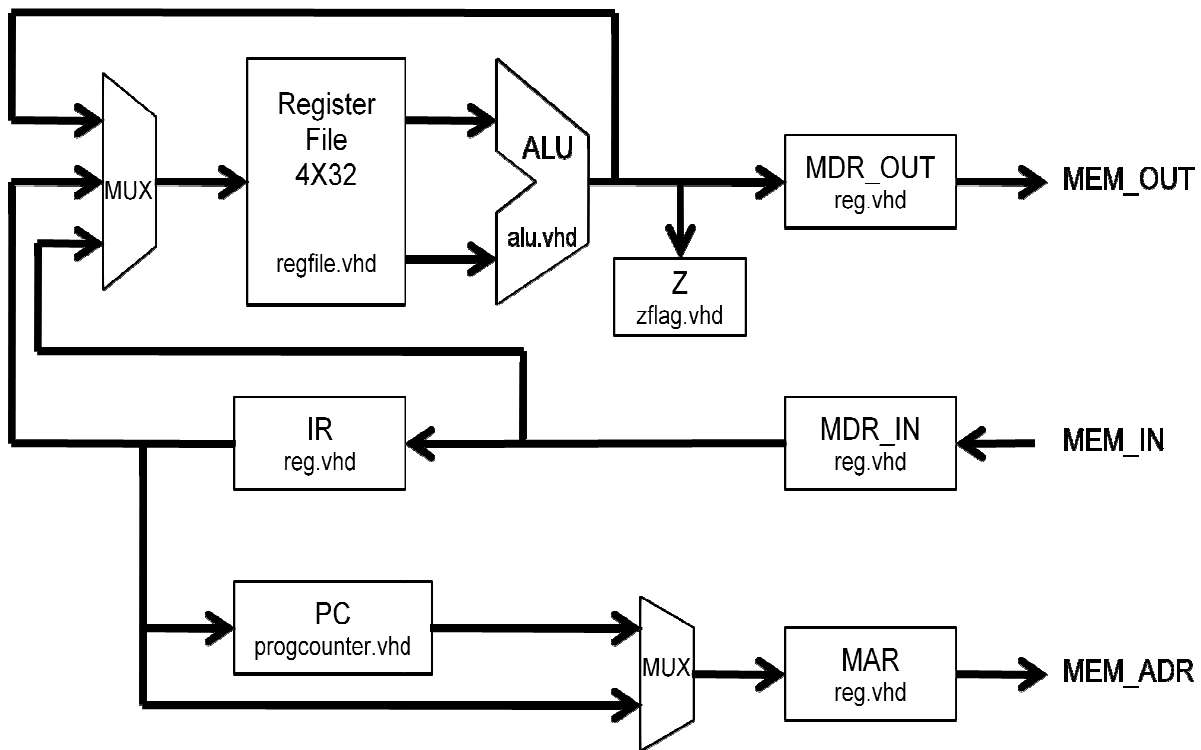
The above instructions are first broken down into series of micro-operations. Each micro-operation represents a single action that can be completed within one clock cycle by the CPU. If no conflict exists, several micro-operations can occur in the same cycle. Each instruction is carried out by a sequence of micro-operations; sometimes referred to as the micro-program. Common to all instruction is the fetching of instruction (or IF: instruction fetch). The Pygmy CPU takes three cycles to fetch an instruction.

Instruction	State	Micro-operations
IF	S1	$MAR \leftarrow PC$; mem_rd;
	S2	$MDR_IN \leftarrow MEM_IN$;
	S3	$IR \leftarrow MDR_IN$; $PC \leftarrow PC + 1$;
LD(0001)	S4	$MAR \leftarrow X"00"&IR(23:0)$; mem_rd;
	S5	$MDR_IN \leftarrow MEM_IN$;
	S6	$R(x) \leftarrow MDR_IN$;
LDI(0011)	S4	$R(x) \leftarrow X"00"&IR(23:0)$;
ST(0100)	S4	$MAR \leftarrow X"00"&IR(23:0)$; $MDR_OUT \leftarrow R(x)$; mem_wr;
	S5	mem_wr;
	S6	mem_wr; (Extra cycle to allow write to complete)
ADD(0110)	S4	$R(x) \leftarrow R(x) + R(y)$; IR(23:20)=0001
AND(0110)	S4	$R(x) \leftarrow R(x) \text{ and } R(y)$; IR(23:20)=1000
XOR(0110)	S4	$R(x) \leftarrow R(x) \text{ xor } R(y)$; IR(23:20)=1010
ROR(0110)	S4	$R(x) \leftarrow R(x) \text{ ror by } \#R(y)$; IR(23:20)=1100
JNZ(1000)	S4	If (zflag='0') then $PC \leftarrow X"00"&IR(23:0)$;

3. Datapath Design

The Pygmy CPU's datapath was designed according to the above micro-operations. Major components are first identified: ALU, zflag, register file (4X32-bit), MAR, MDR_IN, MDR_OUT, PC and IR. Then connections are made according to each micro-operation. Once connections are made for all micro-operations, datapath is completed. Some design decision may be made here. For example, "MDR ← R(x)" at S4 of ST calls for a direct connection between the first register file read port to MDR_OUT. However, here the design decision is made not to add such connection. Instead, R(x) is read from the first read port of register file, bypassing ALU and reach MDR_OUT.

Details of the following data path block diagram can be verified from the VHDL source files. The following simplified block diagram show only the data flow connections. Control signals, which are used to select the multiplexers, enable register loading, etc., are not included here for simplicity. The functionalities of control signals are defined in the individual component VHDL description. One major feature to be noted here is: all registers are loaded on the falling edge of the system clock. This is because the state machine that forms the control unit will send out the enable signal on the rising edge of the system clock. Loading the register on the falling edge of the system clock will: (1) allow enough set-up time for the flip-flops; and (2) allow the loading of the register to finish in one system clock cycle.



3.1. Control Signals

The Datapath represents passive data flow potentials with control signals to dictate the data flow direction and/or destination. While the generations of these control signals is the job for the Control Unit, the control signals are actually defined in the Datapath design. Followings are the control signals of the Pygmy CPU. Note that “mem_wr” and “mem_rd” go directly from the control unit to the Main Memory (which actually includes ROM, RAM and I/O) and thus will not be seen in the Datapath. To simplify the design, and in consistent with the instruction format designed earlier, register number (or address) to the Register File has been hardwired: read port #0 to IR(27:26), read port #1 to IR(25:24) and write port to IR(27:26). Finally, similar arrangement is for IR(23:20) to be used as the selecting signals for ALU when the opcode is “0110” the ALU operations; and thus the ALU selecting signals need not be generated from the control unit.

Inst.	State	Control signals											
		mem_r	mem_w	pcl	pcinc	mmx	Md01	md11	ma1	lr1	aorm	rwr	ccl
IF (Inst. Fetch)	S1	1				1			1				
	S2							1					
	S3				1					1			
LD(0001) Rx, \$M	S4	1							1				
	S5							1					
	S6									01	1		
LDI(0011) Rx, #I	S4									10	1		
STR(0100) Rx, \$M	S4		1				1		1				
	S5		1										
	S6		1										
ADD(0110) Rx, Ry	S4										1	1	
AND(0110) Rx, Ry	S4										1	1	
XOR(0110) Rx, Ry	S4										1	1	
ROR(0110) Rx, #Ry	S4										1	1	
JNZ(1000) \$M	S4			1 ifz=0									

NOTE1: IF stands for instruction fetch. It occurs before any instruction.

NOTE2: The first micro-operation of each instruction always starts at S4, because the IF takes three states to complete (see below).

NOTE3: Some control signals can be multi-bit, such as 'aorm' in this case. Sometimes, you can even group together some control signals to reduce the number of control signals.

- ccl: when '1' enables loading of the condition code (zflag in this case).
- rwr: when '1' enables write to Register File. The write address is IR(27:26).
- aorm: when "00" register_file_input \leftarrow alu_out.
when "01" register_file_input \leftarrow mdr_in.
when "10" register_file_input \leftarrow X"00"&IR(23:0).
when "11" unused .
- irl: when '1' enables loading of instruction register (IR).
- mal: when '1' enables loading of memory address register (MAR).
- mdil: when '1' enable loading of memory data register input side (from memory/IO).
- mdol: when '1' enable loading of memory data register output side (to memory/IO).
- mmx: when '0' MAR \leftarrow X"00"&IR(23:0); when '1' MAR \leftarrow PC.
- pcinc: when '1' increments program counter (PC) by one.
- pcl: when '1' enables loading of PC.
- mem_wr: write control to memory/IO; 0: no write and 1: write.
- mem_rd: read control to memory/IO; 0: no read and 1: read.

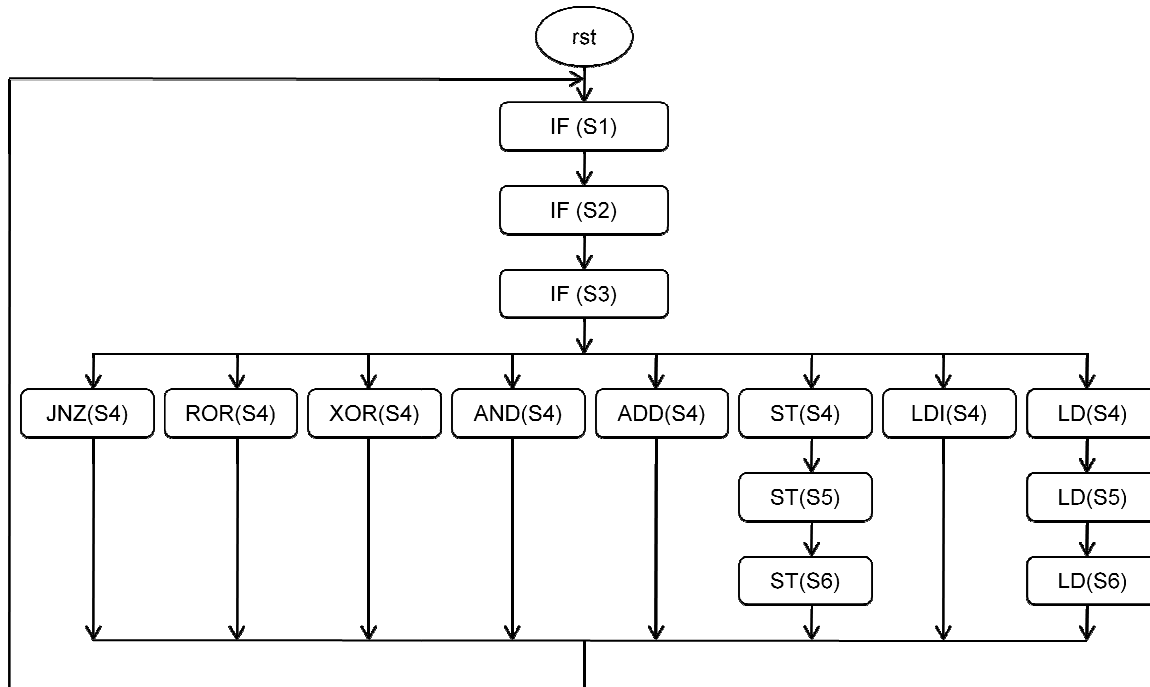
4. Control Unit Design

The control unit is designed as a finite state machine. Its mission is simply to generate the control signals specified in the above table.

4.1. Timing Generation

The control unit design is further divided into two parts: timing generation and control signal generation. The above flowchart is for the timing generation and thus we only concern about the number of state each instruction has to go through. With external reset signal, the state machine will go to its first state S1. In most cases, the state machine simply goes through S1-S4 and then

repeats the sequence again. The FSM or state machine that generates these timing signals is relatively simple. The generations of control signals are basically combinational logic circuits according to the control signal table in the last section.



4.2. Control Signals Generation

Once the FSM structure is established (the Timing_Machine in control.vhd), we will establish the logic that generates the control signals. For instance, 'irl' is '1' only when in S3 and therefore

```
irl <= '1' when (CPU_state=S3) else '0';
```

Here, CPU_state is the internal state variable as found in the VHDL code. Similarly, 'ccl' is '1' only in the S4 of ALU operations: ADD, AND, XOR and ROR. A similar VHDL assignment can be constructed:

```
ccl <= '1' when (IR(31 downto 28)="0110" and CPU_state=S4) else '0';
```

IR(31 downto 28) is of course the Opcode field of the IR and all ALU instructions have the same opcode.

4.3. Pygmy_CPU

Once the datapath and the control unit are completed, they are simply put together to form the Pygmy CPU. To demonstrate how this CPU works, we need to construct a simple demo system. We will add main memory (ROM for program and RAM for scratch data) and I/O to the CPU.

5. Main Memory and I/O

For the demonstration purpose, the CPU is explicitly connected to the main memory and the parallel I/O making it a complete embedded system. There are three components: ROM, RAM and I/O here. The ROM contains a simple program that (1) reads from the input port (the 18 switches on DE-2), (2) writes the data to RAM, (3) reads the data from RAM, (4) outputs the data to output port (which is connected to eight hexadecimal-to-7-segment converter and eventually to the eight 7-segment displays on DE-2 board), (5) perform an XOR function, and finally (6) conditional jump to repeat the loop.

5.1. Memory Map

For this simple demo, we use 1K words of ROM, 1K words of RAM and one address for both input and output port (mem_rd from input and mem_wr to output). Since our program counter will be reset to X"00000000" by the external reset signal, the program should start at that address. The program is stored in the ROM so modifying the instruction is prohibited. The full specification of the Pygmy memory map:

\$00000000	--	\$000003FF	ROM
\$00000400	--	\$000007FF	RAM
\$000007FF	--	\$00DFFFFFFF	Unused
\$00E00000	--	\$00EFFFFFFF	Video RAM (variable size; max=4M)
\$00F00000			LCD Data & Control
\$00F00001			LCD Data & Control
\$00F00002			Timer0
\$00F00003			Timer1
\$00F00004			RS-232 Rx
\$00F00005			RS-232 Tx
\$00F00006			PS/2 keyboard
\$00F00007			unused; Reserved for keyboard
\$00F00008			IQA mask
\$00F00009			IQA flag
\$00F0000A			VGA control
\$00F0000B	--	\$00FFFFFFE	unused
\$00FFFFFFF			Switches and LEDs
\$01000000	--	\$FFFFFFFFF	unused

Remember the memory mapping has nothing to do with the CPU design. Pygmy CPU simply supports 32-bit address and 32-bit data for the main memory and I/O. It is at the system level that such decisions are made. For Lab 1, you will see only the I/O for switches and LEDs. Other I/O devices will be gradually introduced in the later labs.

The ROM and RAM are generated via generic map from “ram.vhd”. Note that the read-only part of the ROM is emulated by disabling the write capability. This is true only in the FPGA environment. In addition, the VHDL compiler will read the “rom.mif” during compilation.

5.2. Pin Assignments

DE-2 board came with many components and features. However, this also means hundred of pins need to be assigned. To simplify the process, we will copy and paste the “Pin & Location Assignments” section in the “.qsf” file, from the DE-2 default file. However, this should be done after the first compilation and remember to copy and paste only this section. Since we will not use all the pins in each lab, some pins will be assigned and unused. You should ignore those warnings.

Another important new feature on Quartus II that you should know is the “Update Memory Initialization File” under “Processing”. When trying out a different program and thus only ‘mif’ file needs to be updated, this feature will save you from compiling the entire design.

5. Program listing of “simple_demo”

The program’s machine code is stores in the ROM’s memory initialization file (rom.mif)

Addr.	Codes	
00000000	3C000001	LDI R3,#000001 ;R3=1
00000001	10FFFFFF	LP0: LD R0,\$inport ;R0=switch inputs
00000002	3800FFFF	LDI R2,#00FFFF ;R2=FFFF
00000003	62800000	AND R0,R2 ;leave only last 16 bits
00000004	40000400	ST R0,\$000400 ;save to RAM
00000005	14000400	LP1: LD R1,\$000400 ;read from RAM to R1
00000006	44FFFFFF	ST R1,\$outport ;write R1 to LED outputs
00000007	10FFFFFF	LD R0,\$inport ;R0=switch inputs
00000008	380F0000	LDI R2,#0F0000 ;R2=the mask
00000009	62800000	AND R0,R2 ;leave only two higher bits
0000000A	8000000D	JNZ FLSH ;start flashing sequence
0000000B	63A00000	XOR R0,R3 ;dummy op to reset z flag
0000000C	80000001	JNZ LP0 ;obtain new switch inputs
0000000D	67C00000	FLSH: ROR R1,R3 ;rotate display one bit right
0000000E	44000400	ST R1,\$000400 ;store new content to RAM
0000000F	30200000	LDI R0,#200000 ;R0=delay count
00000010	18000015	LD R2,\$000015 ;R2=-1 (pre-store at \$15)
00000011	62100000	DLY: ADD R0,R2 ;decrement R0 by one
00000012	80000011	JNZ DLY ;delay for proper display
00000013	63100000	ADD R0,R3 ;dummy op to reset z flag
00000014	80000005	JNZ LP1 ;display new content
00000015	FFFFFFF	Constant -1

Appendix A: Complete ISA of Pygmy CPU

<i>Assembly</i>	<i>Operations</i>	<i>Machine Codes</i>	<i>lab</i>
LD Rx, \$M	$R_x \leftarrow \text{MEM}(\$M)$	0001 xx - mmmmmmmmmmmmmmmmmmmmmmmmmmm	0
LDR Rx, \$Ry	$R_x \leftarrow \text{MEM}(\$Ry)$	0010 xx yy -----	2
LDI Rx, #I	$R_x \leftarrow \#I$	0011 xx - iiiiiiiiiiiiiiiiiiiiiiiiiiiiii	0
ST Rx, \$M	$\text{MEM}(\$M) \leftarrow R_x$	0100 xx - mmmmmmmmmmmmmmmmmmmmmmmmmmm	0
STR Rx, \$Ry	$\text{MEM}(\$Ry) \leftarrow R_x$	0101 xx yy -----	2
ADD Rx, Ry	$R_x \leftarrow R_x + R_y$	0110 xx yy 0001-----	0
ADDI Rx, #I	$R_x \leftarrow R_x + \#I$	0110 xx -- 0010iiiiiiiiiiiiiiiiiiiiiiiiiii	1
SUB Rx, Ry	$R_x \leftarrow R_x - R_y$	0110 xx yy 0011-----	1
MUL Rx, Ry	$R_x \leftarrow R_x * R_y$	0110 xx yy 0100-----	1
AND Rx, Ry	$R_x \leftarrow R_x \text{ and } R_y$	0110 xx yy 1000-----	0
OR Rx, Ry	$R_x \leftarrow R_x \text{ or } R_y$	0110 xx yy 1001-----	1
XOR Rx, Ry	$R_x \leftarrow R_x \text{ xor } R_y$	0110 xx yy 1010-----	0
ROR Rx, #Ry	$R_x \leftarrow R_x \text{ ror } \#Ry \text{ bits}$	0110 xx yy 1100-----	0
MOV Rx, Ry	$R_x \leftarrow R_y$	0111 xx yy -----	1
JNZ \$M	$PC \leftarrow \$M \text{ if } Z=0$	1000 0000 mmmmmmmmmmmmmmmmmmmmmmmmmmm	1
JNN \$M	$PC \leftarrow \$M \text{ if } N=0$	1000 0001 mmmmmmmmmmmmmmmmmmmmmmmmmmm	1
JNV \$M	$PC \leftarrow \$M \text{ if } V=0$	1000 0010 mmmmmmmmmmmmmmmmmmmmmmmmmmm	1
JNC \$M	$PC \leftarrow \$M \text{ if } C=0$	1000 0011 mmmmmmmmmmmmmmmmmmmmmmmmmmm	1
JZ \$M	$PC \leftarrow \$M \text{ if } Z=1$	1000 0100 mmmmmmmmmmmmmmmmmmmmmmmmmmm	1
JN \$M	$PC \leftarrow \$M \text{ if } N=1$	1000 0101 mmmmmmmmmmmmmmmmmmmmmmmmmmm	1
JV \$M	$PC \leftarrow \$M \text{ if } V=1$	1000 0110 mmmmmmmmmmmmmmmmmmmmmmmmmmm	1
JC \$M	$PC \leftarrow \$M \text{ if } C=1$	1000 0111 mmmmmmmmmmmmmmmmmmmmmmmmmmm	1
JMP \$M	$PC \leftarrow \$M$	1000 1111 mmmmmmmmmmmmmmmmmmmmmmmmmmm	1

CALL Rx, \$M	$R_x \leftarrow PC$ then $PC \leftarrow \$M$	1001 xx-mmmmmmmmmmmmmmmmmmmmmmmmmmm	3
RETN Rx	$PC \leftarrow R_x$	1010 xx-- -----	3
RETI	$PC \leftarrow \text{Link}$; restore all regs	1011 ---- -----	4
TRAP \$M	$\text{Link} \leftarrow PC$ then $PC \leftarrow \$M$; save all regs	1111 ---- mmmmmmmmmmmmmmmmmmmmmmmmmmm	4

NOTES:

1. '\$' denotes address and '#' means immediate data
2. Bits 31 down to 28 are "opcode" field. Only 12 assignments have been made and thus room for future expansion of instruction set. Some instructions share the same opcode. (see below)
3. The immediate data in LDI instruction is 24-bit while ADDI supports only 20-bit data; both with sign extension to support 2's complement data format.
4. All ALU (arithmetic and Logic Unit) operations use "0110" as opcode, but with bits (23 down to 20) to indicate the individual operations. Note that only eight ALU operations have been defined, leaving room for seven more ('0000' is usually reserved for internal bypass function).
5. ROR is "rotate to right" which bits are shifted out through LSB and rotate back from MSB. While Rx (operand 0) is the source and destination, Ry (operand 1), between 0 to 31, is the number of bits to rotate. The condition code 'C' is set when the last bit that goes from LSB to MSB is '1'; C is cleared otherwise.
6. All jump instructions share '1000' as opcode and use bits (27 down to 24) to indicate the type of jump. With four condition codes: Z, N, V and C, a total of eight conditional jumps are implemented with one unconditional jump (JMP). There are rooms for seven more.
7. CALL use one of the four general purpose registers to store the return address. Programmer can realize nested subroutine calls by using different registers. RETN, return from subroutine restores the saved return address back to PC.
8. TRAP and hardware interrupt (which does not show in the instruction set listing, of course) all use an internal (hidden) "Link" register to save the return address. In addition to saving the return address before jump to the designated location, all registers contents are saved. This includes R0 – R3 and all condition codes. The registers contents are restored when RETI (return from interrupt) is executed. RETI works for both hardware and software interrupt service routines.

Appendix B: Complete Pygmy CPU Hardware Interface

Pygmy CPU has three groups of buses: address, data and control. The basic bus interfaces are given in the beginning of the semester, but, several will have to be implemented in the later laboratory assignments.

Address bus: 32-bit wide and goes to memory or I/O.

Data bus: 32-bit wide; one for each direction to and from CPU. Since only 32-bit words are supported, all data are accessed 32-bit at a time. The smallest addressable unit is a word.

Control bus: A collection of control signals between CPU and memory, and CPU and I/O.

- *mem_rd* and *mem_wr*: Indicates CPU's intention to read or write memory or I/O in the next clock cycle. Since there is a chance of CPU want neither read nor write, two control signals are used.
- *halt*: This is a signal sent from memory (usually slower memory) to CPU requesting a "time extension". When "halt" goes high, CPU will extend the current clock cycle until "halt" goes back to low again. However, this should happen only when CPU is reading from or writing to the memory. In other words, halt should remain low when *mem_rd* and *mem_wr* are both low.
- *int*: Interrupt (*int*) is used for I/O to interrupt current CPU operations for an immediate attention on I/O events. CPU checks for interrupt after the completion of the previous instruction and before the fetching of the next instruction (before S1). If "int" is high, CPU will initiate the interrupt service routine by (1) save the current PC to Link, (2) save all registers contents including condition codes, and (3) jump to location X"00000001". At location X"00000001", there should be a "JMP" instruction to transfer to a program that reads from the external interrupt request arbitrator (IQA), to determine which I/O device has send the interrupt request. Usually conditional jump are executed to go to the proper I/O handling routine. An I/O handling routine should end with an "RETI" instruction, which will (1) restore all registers contents including condition codes, and (2) restore PC from Link.