16-BIT RISC PROCESSOR

BY

JOHN TAYLOR

PREETI CHITRE

FABIAN ROSADI

JOEL MONTES DE OCA

Purpose of the project:

The main purpose of the project is to design a 16-bit RISC processor. The basic Data path is given and it has to be modified to execute all the instructions properly.

PROJECT OVERVIEW

Reduced instruction set computer (RISC) and Complex instruction set computer (CISC) are two conceptually different approaches to the CPU architecture. Most of the newest CPUs have RISC architecture. The IBM RT processors, the Sun SPARC, the PowerPC CPUs built by Motorola and IBM, the Digital Alpha chip and the MIPS processors are all categorized as RISCs.

The RISC architecture is an attempt to produce more CPU power by simplifying the instruction set of the CPU. Each instruction of a RISC can be completed in one clock cycle. The main features of a RISC processor are:

- A limited and simple instruction set: The RISC instruction set consists of instructions that can execute quickly, at high clock speeds and using a hard-wired and pipelined implementation. There are no limitations on the number of instructions in a RISC instruction set, but usually the number of instructions is smaller than a CISC.
- Register-oriented instructions, with limited memory access: A RISC instruction set typically supplies only a few basic LOAD and STORE instructions that can access data in memory. All other instructions operate only with registers. Memory access instructions require extra time to execute.
- A fixed length, fixed format instruction word: By making every instruction word identical in size and format, instructions can be fetched and decoded independently. It is not necessary to wait until the length of a previous instruction is known, in order to fetch and decode the next instruction. Instructions can therefore be fetched and decoded in parallel.

- Limited addressing modes: Many RISC machines provide only a single addressing mode for addressing memory. This simplifies the implementation and speeds up instruction execution.
- A large bank of Registers: RISC CPUs provide many registers so that variables and intermediate results used during program execution do not require the use of memory. Thus many LOADs and STOREs can be avoided.

LOAD, STORE and BRANCH instructions pose a problem to the implementation of a RISC design. As these instructions require memory access, their execution cannot be completed in a single clock cycle. This means that the instruction immediately following a LOAD or a STORE instruction in the pipeline may not use the register that is being stored. This is called a latency requirement. To avoid this problem many RISC machines provide hardware that locks out an instruction that attempts to violate the latency requirement. Others rely on their programmers or compilers to fill spaces immediately following these instructions, by NO OPeration instructions.

Many studies on RISC processors indicate that the simple instruction set leads to larger memory requirements, but this is compensated by the use of small, fixed size instructions. Many studies also indicate that RISC approach leads to faster program execution. RISC designers have no doubt provided some of the most powerful CPU architecture available today.

The above is a basic overview of the RISC processor. The purpose of this project is to design a RISC processor with a minimal instruction set that is universal. This implies that a user can describe any kind of algorithm with the instruction set. The design process involves hardware as well as software design.

GROUP MEMBERS SHORT BACKGROUND AND RESPONSIBILITIES

Joel Montes de Oca

ECE Major with no specific option.

Technical Experience:

Experience gained from the various laboratories during the curriculum

Worked as an intern and gained experience in the production of microscopic resistor networks. Also had some experience in updating technical documents such as work procedures and procedure log cards.

Project Responsibilities:

Provided the necessary support in every area of the project from programming to documentation and diagrams, wherever there was a backlog of work as the project moved along. Coordinated with all the group members to ease the workload of the whole team.

Fabian Rosadi

ECE Major with Computer and Controls and Robotics option.

Technical experience:

Worked as a PC technician and had hands on experience in troubleshooting PC and Macintosh hardware and software

Also experienced with troubleshooting complex circuit boards down to component level

Project Responsibilities:

Provided support to the documentation by doing the related technical diagrams.

Preeti Chitre

ECE Major with Computer engineering option.

Technical Experience:

Worked for 5 years in the design and development of microprocessor based control instruments, in India. Was involved in the projects from design to documentation.

Hands on experience of various instruments such as logic analyzers and oscilloscopes.

Project Responsibilities:

Provided documentation support for the entire project.

John Taylor

ECE Major with Computer engineering option.

Technical Experience:

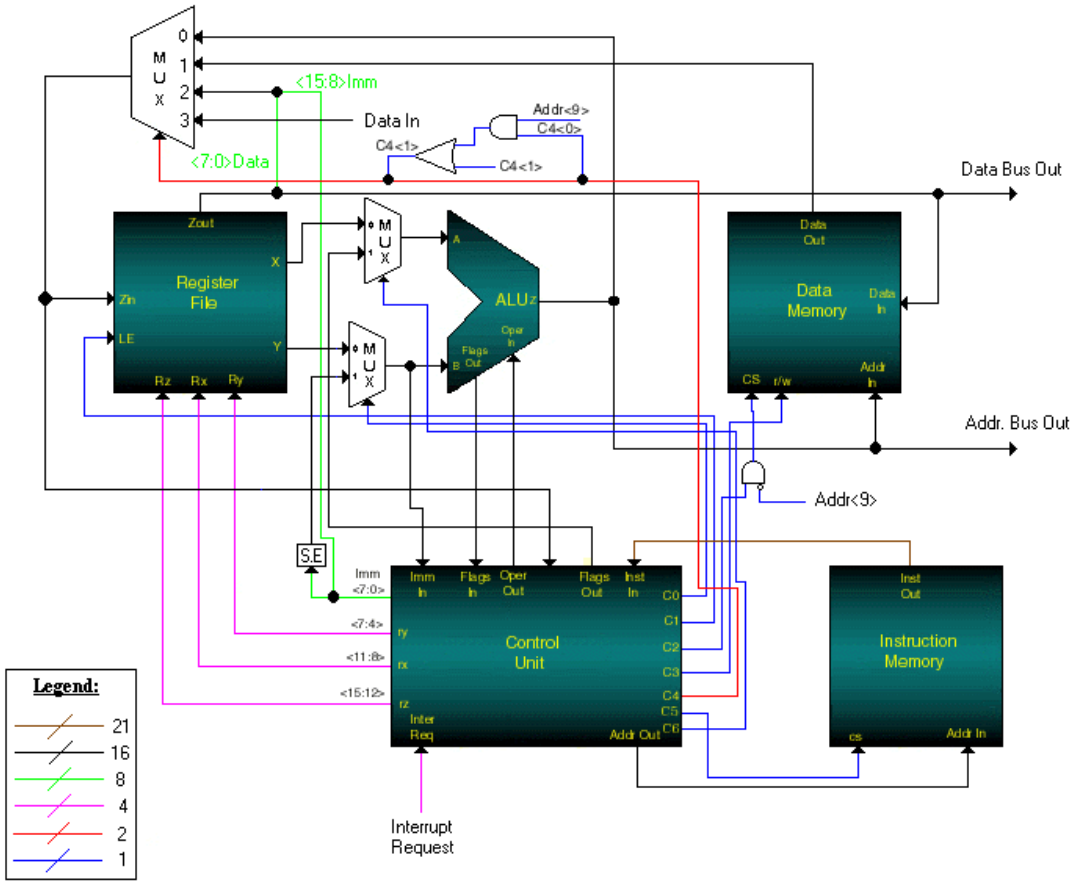
Good experience in various programming languages, particularly C and C++.

Project responsibilities:

Provided programming support for the entire project.

All the four members of the group contributed towards the designing of the project.

DETAILED DESIGN DESCRIPTION



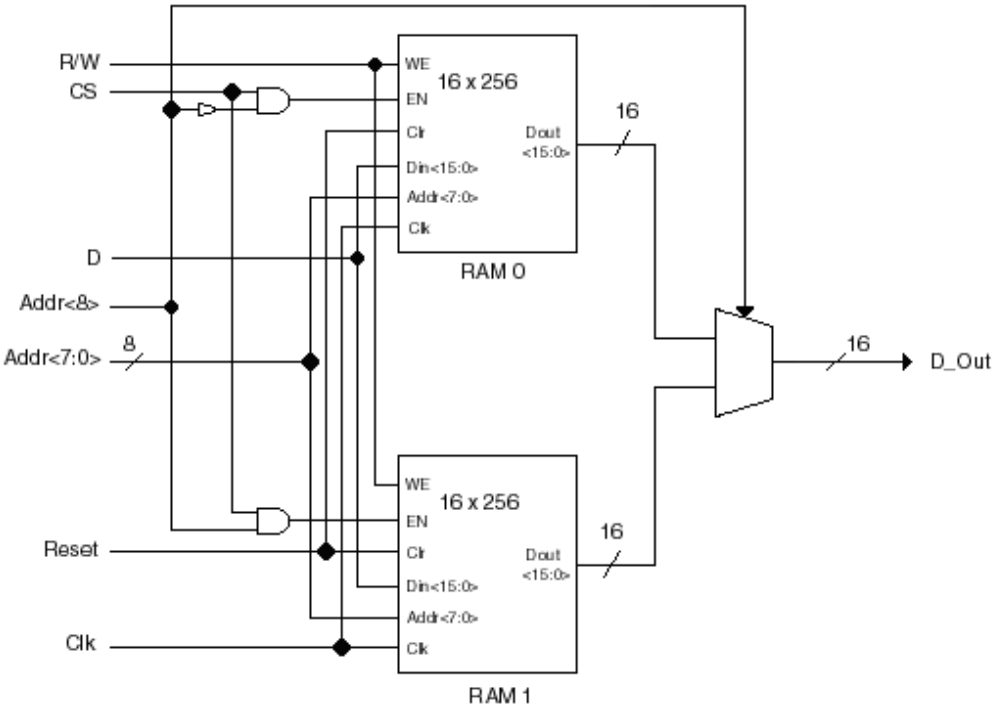
The RISC processor consists of five basic blocks: the control unit, the arithmetic and logic unit (ALU), registers, data memory and instructions memory as can be seen from the above block diagram. These blocks reside inside the SPARTANII–XC2S150 chip. The Spartan-II family of chips from Xilinx is the field programmable gate arrays (FPGAs), as they are commonly known. The FPGAs have a large number of programmable logic blocks called the configurable logic blocks (CLBs). We will use a modular approach to describe the design, starting by describing each block with its functions and then the data path and the interactions between the blocks.

- **Control Unit:** The control unit provides the necessary timing and control signals to all the operations in the RISC processor. It controls the flow of data between each block of the data path. Thus the control unit is basically a ROM with the number of **Logic Unit:** In this area of the address lines equal to the number of bits in the opcode.
- **Arithmetic/** RISC processor, computing functions are performed on the data. The ALU performs arithmetic operations such as ADD and SUB and logic operations such as AND, OR, XOR and NOT. The results from the ALU are stored either in the registers or the memory. The ALU has two inputs x and y and one output.
- **Registers:** This area of the RISC processor consists of various registers. The registers are used primarily to store data temporarily during the execution of a program. The register block in this RISC processor contains 16, 16-bit registers and their associated control logic. Registers rx and ry act as the source registers and register rz acts as the destination register. The address lines rx, ry and rz are each 4 bits wide and select two source registers and one destination register. The

contents of the two selected source registers are output to rx OUT and ry OUT. The destination register rz outputs its contents on rz OUT, but it also loads whatever is on rz IN at the following clock cycle, if the ~load enable signal is asserted.

- **Data Memory:** The Data Memory stores the data used during the operations and provides that information to the processor whenever necessary. The read/~write signals work in conjunction with the chip select signal to determine the operation of the Data Memory.
- **Instruction Memory:** The Instruction Memory stores instructions in binary form and provides necessary information to the control unit to execute the instructions.

DATA MEMORY:

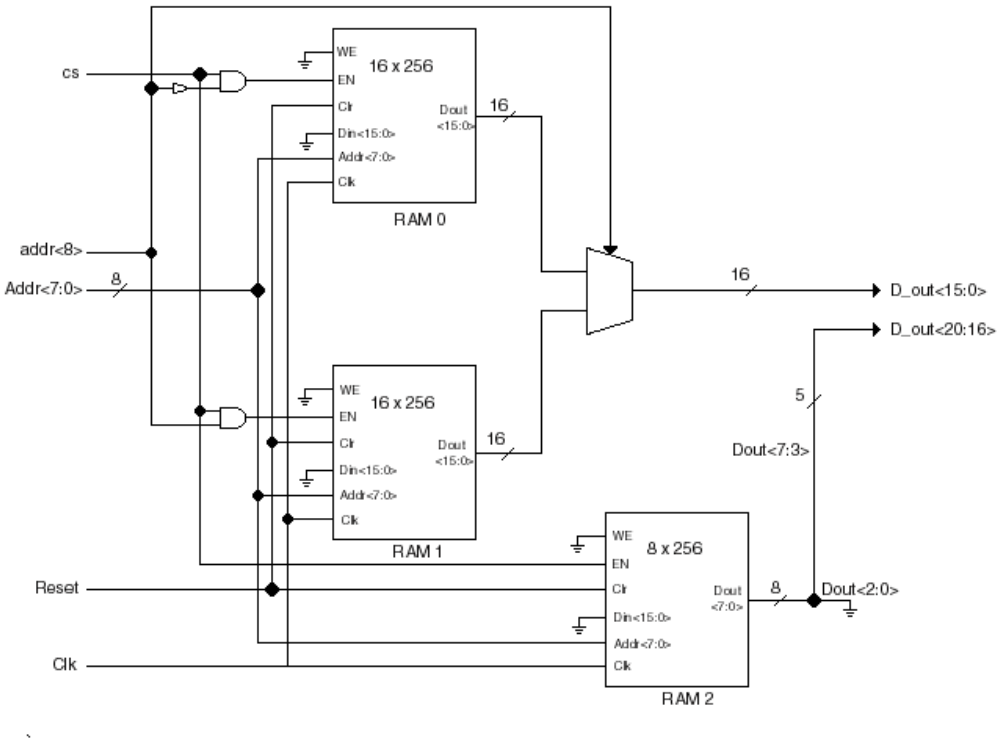


The Spartan-II family of chips provides Block SelectRAM+ memory, providing from 4 to 12 blocks of 4K bits of dual-port RAM. Both ports are configurable to any size from 4Kx1 to 256x16,allowing built-in bus width conversion. Each port is completely independent and fully synchronous, allowing the creation of flexible RAM structures. This dedicated RAM provides very high speed, comparable to discrete memories or ASIC memory cores. The Block SelectRAM+ memory allows the Spartan-II FPGAs to handle on-chip memory requirements. For further details about the SelectRAM and its applications refer to Appendix D.

In the RISC processor application, the RAM ports are configured in 256x16 blocks. Since the Data memory is 16 bits wide, two 256x16 blocks are used in the 512x8 configuration. The address bits <8:0> are used for this purpose. Address bit <8> will be used to select the block RAM. If bit <8> is 0, the lower address memory block will be selected. If bit <8> is 1, the higher address memory block will be selected.

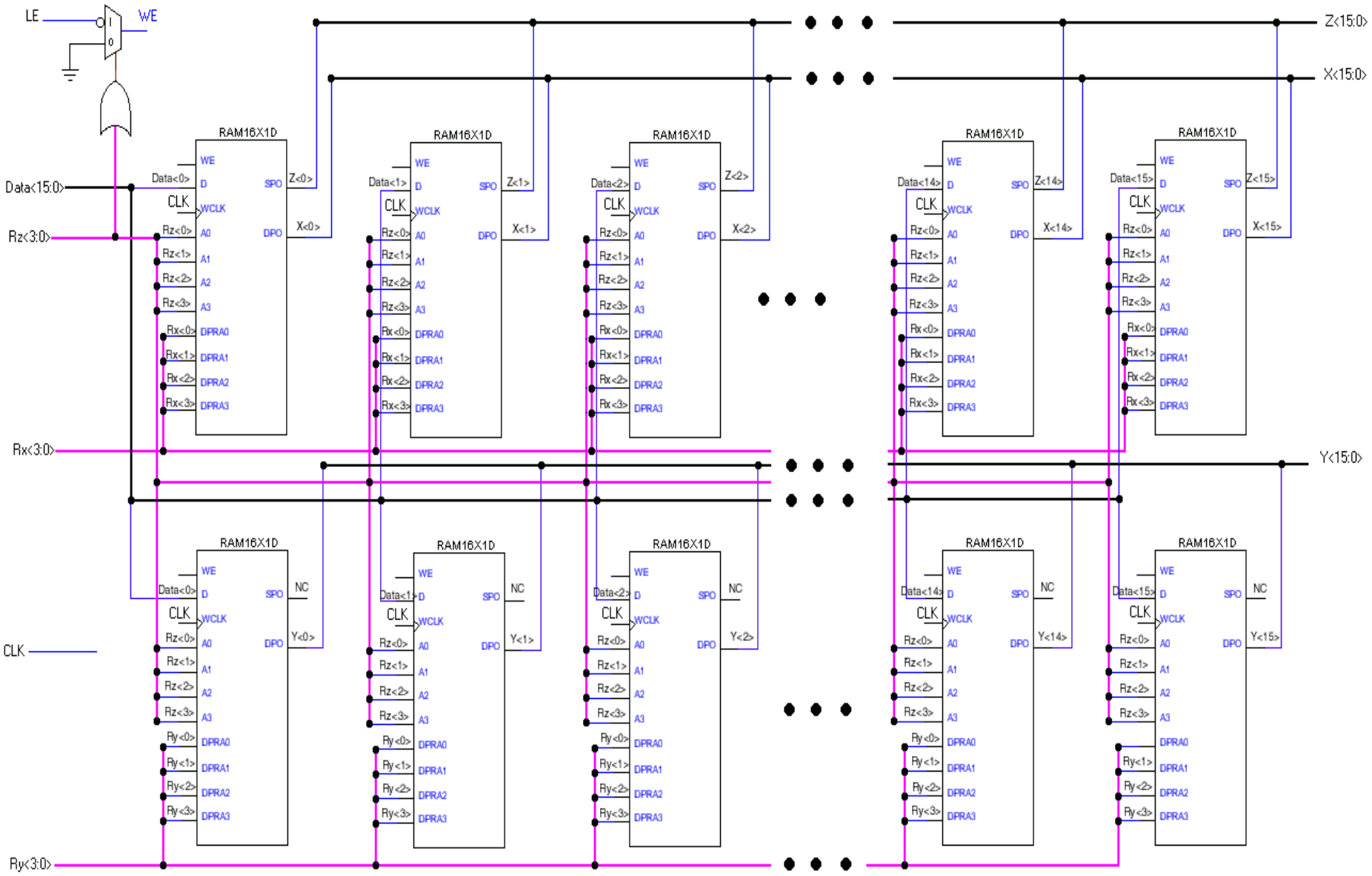
The common D_Out of both the blocks goes to the register file through the selection logic, which is the MUX

INSTRUCTION MEMORY:



The Instruction Memory is similar to the Data memory, except that different configurations of the Block SelectRAM+ are used. For Instruction memory, 21 bits of address is required. Hence two 256x16 blocks of memory provide 512 address locations for the lower 16-bits of the instruction memory and one 512x8 block RAM is used for the upper 5-bits of the instruction memory.

Register File



The Register file is implemented using the dual-port distributed SelectRAM. These modules are 1 bit wide and 16 address spaces deep. They have one input port, two output ports and two sets of address lines. The first address controls the input and the first output, making it capable of both read and write operations. The second address controls the second output, making it read-only. 16 of these basic modules are combined in parallel to form a 16x16 dual-port memory register. Since the register file needs three output ports, two of these 16x16 registers are used.

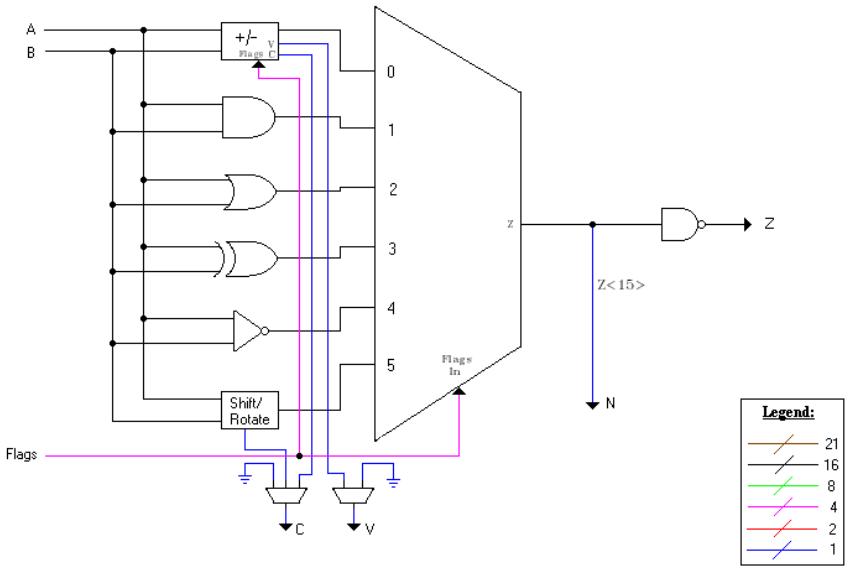
The first register set up is as follows:

The first address is connected to the Z address of the Register file. The second address is connected to the X address of the register file. The Z input of the register file connects to the register's input. The two outputs are connected to the Z and X outputs of the register file respectively.

The second set up is as follows:

The first address of this register is also connected the Z address of the register file. The second address is connected to the Y address of the register file. The Z input of the register file also connects to this register's input, but the first output of this register is left unconnected. The second output of this register is connected to the Y output of the register file.

ARITHMETIC AND LOGIC UNIT:



The ALU is implemented using the built-in arithmetic function generator inside the SPARTAN-II chip. The SPARTAN-II FPGA is based on the VIRTEX architecture. The VIRTEX architecture employs a powerful CLB that uses fewest logic cells with a minimum delay, which are used in the arithmetic operations. This VIRTEX architecture and the VHDL coding help to achieve a powerful performance. This built-in arithmetic function generator is the basic block of the Arithmetic and Logic Unit block of the RISC processor.

The basic building block of a VIRTEX CLB is the Logic cell. A Logic cell includes a 4-input look-up-table and can implement any 4-input logic function. The 2 Look-up-tables within a slice can be combined to create a 16x2-bit or 32x1-bit synchronous RAM or 16x1-bit dual-port synchronous RAM. The most relevant feature of

the CLB is the dedicated carry logic to implement fast efficient arithmetic functions. The VHDL coding of these features to implement a two input 16-bit ALU can be found in the Application notes attached in the Appendix E.

The ALU is further divided into functional blocks-the arithmetic block and the logic block. The arithmetic block consists of the Adder and Subtractor. The logic block consists of the AND, OR, XOR, NOT functions and the shift and the rotate function. There are 4 flags that set /reset during the various ALU operations. They are: carry, overflow, sign, zero. These flags are set/ reset in the flags status register. The various functions and the flags are fed through a multiplexer and depending on the select logic, the appropriate function is chosen and the results are fed through the multiplexer output to the Data memory or the Register file.

THE DESIGN PROGRESS OF THE RISC PROCESSOR

The RISC processor design was a three quarter assignment. We started in Fall of 2000 with a class in VHDL design. With no previous background of VHDL, we started learning VHDL from scratch, in this class. Thus we got acquainted with the Xilinx Foundation Series 2.1 student version software for programming in VHDL. Learning the Xilinx software was a tough job as the software is not user friendly. During this quarter we were given the basic instruction set for the RISC processor and we had to proceed from there with the design.

We started with the design of the Arithmetic and Logic Unit (ALU). Initially we had designed a Look Ahead Carry Generator for the arithmetic part of the ALU. But as the design progressed we found that working with the built-in dedicated carry logic was more efficient and used less space inside the SPARTAN II chip. Hence we shifted from the Look Ahead Carry generator to the built-in arithmetic function generator. Also we added four more logic instructions for the RISC processor. They were the shifts (SLL and SLR) and the rotates (ROR and ROL).

The next step in the design was the development of the Register File. We started the Register File with a basic D flip-flop, which acts as a 1-bit register. We then converted this 1-bit register to a 16x16 Register File keeping the same D flip-flop as our basic block and using decoders for selecting any particular register. Tri-state buffers were used at the output of the Register File. While simulating everything together we found out that this logic consumed a lot of resources on the chip. Though with the SPARTAN-II chip, the space was not an issue, the tri-state buffers caused some bus conflicts in the simulator. Hence we decided to use the distributed Select-RAM that is inside the

SPARTAN-II look-up tables. This new design turned out a lot smaller and a bit faster than the older version.

Then came the most important and difficult block, the Control Unit. Initially we decided to implement the Control Unit special registers such as the PC, PCRC, FLAGS, PCRI and INTV outside the Control Unit. We decided to incorporate them as a part of the Register File. But things became too complicated. So we finally decided to put the special registers inside the Control Unit itself. The difficult part in the control unit was the addition and implementation of various control signals. The Control Unit design occupied 80% of the Winter quarter.

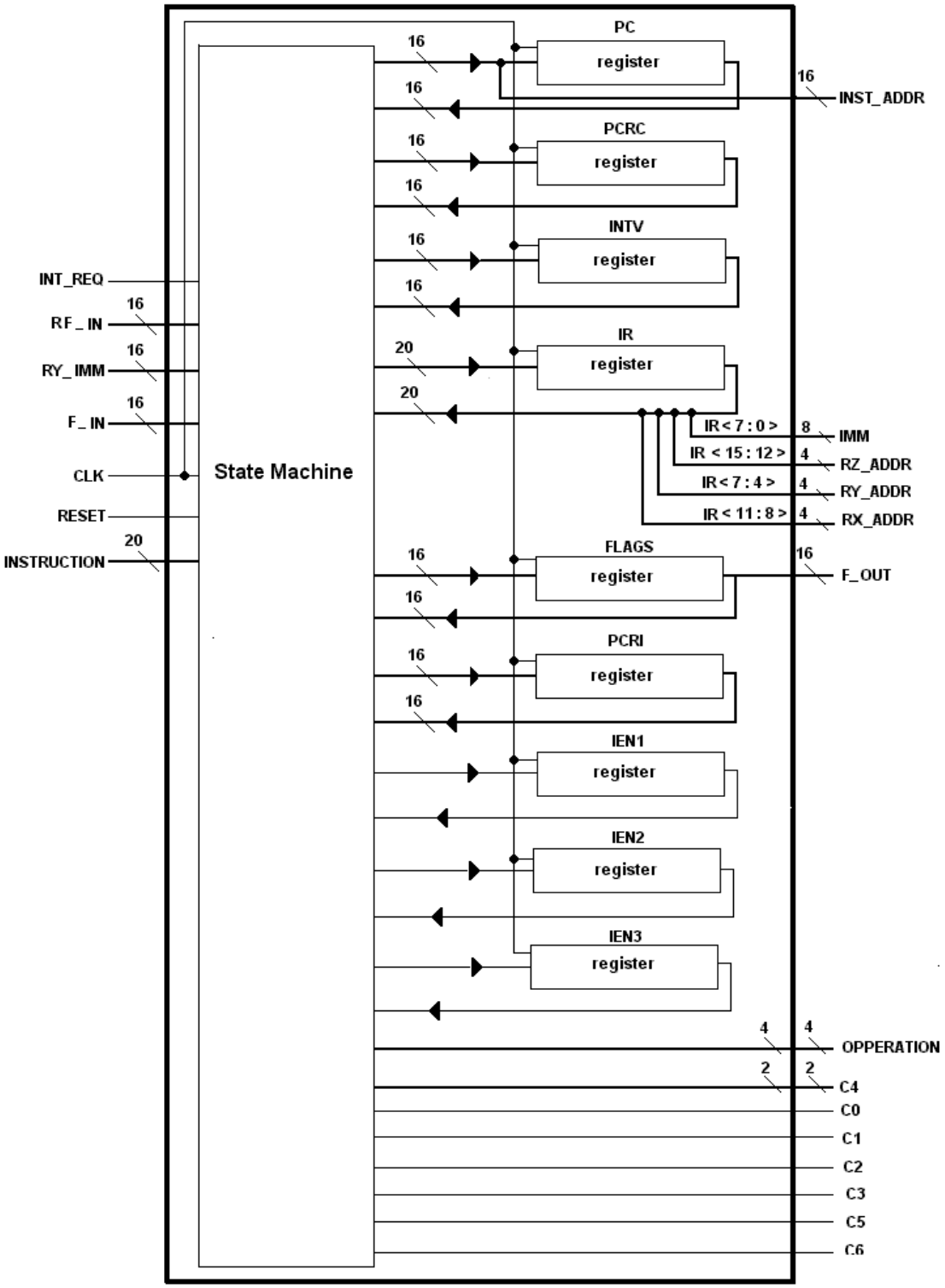
The design of the Data and Instruction memory was comparatively easy task. We used the built-in Select-RAM, which simplified the design and simulation process.

The Interrupt handling was the last part of our RISC processor design. We had to synchronize the Interrupt Request with the clock. Also some changes had to be made in the Data Path for the Interrupt.

Finally the design was complete and the simulation was perfected. The Digilent boards that contained the SPARTAN-II 150 chip that we were using to implement the design, arrived a little later than expected. When they finally arrived they had the SPARTAN-II 200 chip instead of the SPARTAN-II 150. The problem was that the Xilinx 2.1 Student version did not support the 200 chip and we had to use WebPack software instead to program the chip. So additional time was spent on learning the new software.

But in the end the project was a huge success!

Control Unit



CONTROL UNIT:

The control unit is the module that directs the traffic of data throughout the processor datapath. The control unit consists of a state machine that controls the execution of instructions, and a set of specialized registers that are used for program control, CPU status and interrupts.

The state machine cycles between two states: the fetch state and the execute state. Instructions are retrieved from the instruction memory during the fetch state. Data is retrieved, processed and stored in the execute state.

The Program Counter register (PC) contains the address of the current instruction being executed. It is incremented automatically during every fetch cycle. The PC register cannot directly be accessed by the user, but its contents can be modified using the program control instructions.

The Program Counter – Return from subroutine Call register (PCRC) maintains the return address for a subroutine. PCRC is updated with PC when a subroutine is called, and PCRC is transferred back to PC when the subroutine ends.

The Program Counter – Return from Interrupt register (PCRI) maintains the return address for an interrupt routine. When an interrupt routine is called, PCRI is updated with PC. When the routine finishes, PCRI is transferred back to PC.

The Flags register contains the four arithmetic flags and the three interrupt flags in the lower seven bits of the register. The upper nine bits are not used by the processor and can be used as user-defined bits:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
									ACK	IEN	REQ	Z	N	V	C

C The carry flag indicates if a carry was detected in an arithmetic operation. This flag is pertinent to operations on unsigned numbers. This flag is cleared to zero when a logical operation is performed.

V The overflow flag indicates if an overflow was detected in an arithmetic operation. This flag is pertinent to operations on signed numbers.

N The negative flag indicates that the result of any arithmetic or logical operation results in a negative number. This flag comes directly from the most significant bit of the result in the ALU.

Z The zero flag indicates that the result of any arithmetic or logical operation results in zero.

REQ The interrupt request flag is updated directly from the INT_REQ pin on the CPU. This flag is hardwired and cannot be manipulated by the user.

IEN The interrupt enable flag is cleared to zero upon reset of the computer and can be modified by the user via the OR and AND instructions.

ACK The interrupt acknowledge flag is available to the user to be set during an interrupt routine. This flag is tied directly to the ACK output pin on the CPU to be used in interfacing with external devices. It is up to the user to clear and set this flag as necessary in the interrupt routines.

The Instruction Register (IR) contains the instruction currently being executed. This register is updated from the instruction memory during each fetch cycle.

The Interrupt Vector (INTV) contains the address of the interrupt service routine. When an interrupt occurs, PC receives the value in INTV and the interrupt routine begins.

There are three interrupt enable registers that are one bit each: IEN1, IEN2 and IEN3. These registers are used to make sure that the interrupt system is enabled for four cycles before an interrupt call can be made. At the end of each instruction cycle, the IEN3 is updated with the value of IEN2, which in turn is updated with the value of IEN1, which in turn receives the value of the IEN bit of the flags register. Hence, when the user enables the interrupt system by setting the IEN bit in the flags register, it will take three more instruction cycles for the IEN1, IEN2 and IEN3 registers to be set.

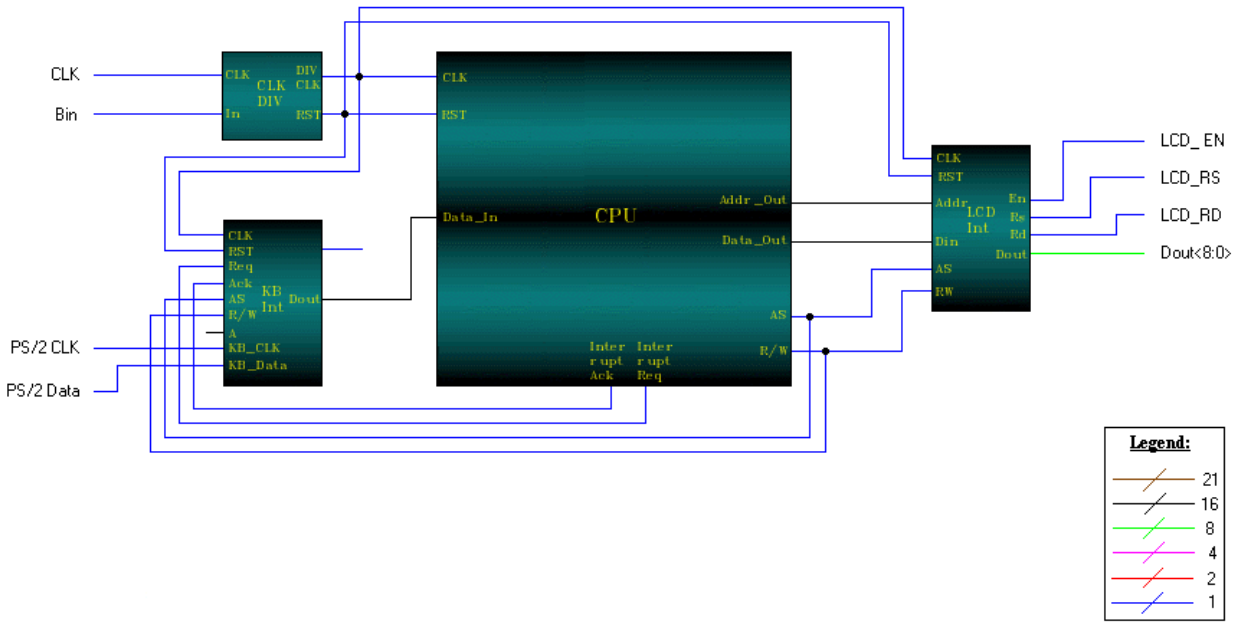
The processor is capable of executing a set of twenty instructions that are listed below:

Opcode	Hex	Instruction Format		Operation
00000	00	ADD	RZ, RX, RY	RZ ← RX + RY
00001	01	SUB	RZ, RX, RY	RZ ← RX - RY
00010	02	AND	RZ, RX, RY	RZ ← RX ∧ RY
00011	03	OR	RZ, RX, RY	RZ ← RX ∨ RY
00100	04	XOR	RZ, RX, RY	RZ ← RX ⊕ RY
00101	05	NOT	RZ, RX	RZ ← ~RX
00110	06	SHR	RZ, RX, #IMM	RZ ← IMM<0> RX<15:1>
00111	07	SHL	RZ, RX, #IMM	RZ ← RX<14:0> IMM<0>
01000	08	ROR	RZ, RX	RZ ← RX<0> RX<15:1>
01001	09	ROL	RZ, RX	RZ ← RX<14:0> RX<15>
01010	0A	STORE	RZ, RX, RY	M[RX+RY] ← RZ
01011	0B	LOAD	RZ, RX, RY	RZ ← M[RX+RY]
01100	0C	ADDI	RZ, RX, #IMM	RZ ← RX + #IMM
01101	0D	BTSS	RX, #MASK	* see below
01110	0E	BTSC	RX, #MASK	* see below
01111	0F	JR	#IMM	PC ← PC + #IMM
10000	10	JA	RY	PC ← RY
10001	11	LUI	RZ, #IMM	RZ<15:8> ← #IMM
10010	12	CALL	RY	PCRC ← PC, PC ← RY
10011	13	RET		PC ← PCRC

*Notes for BTSS and BTSC: Register RX is ANDed with the immediate value #MASK to perform a bit test. For BTSS, if the result is not zero then PC ← PC + 2, else PC ← PC + 1. For BTSC, if the result is zero then PC ← PC + 2, else PC ← PC + 1.

For a detailed explanation about the Control Unit and the processor instruction set please refer to the attached Appendix B.

DEMONSTRATION OF THE PROCESSOR



To demonstrate the processor, it was decided to develop a “hangman” game. This was chosen because it would be a relatively simple program to develop, yet entertaining enough to capture the interest of the user. This game also would take advantage of a PS/2 keyboard and an LCD display for input and output.

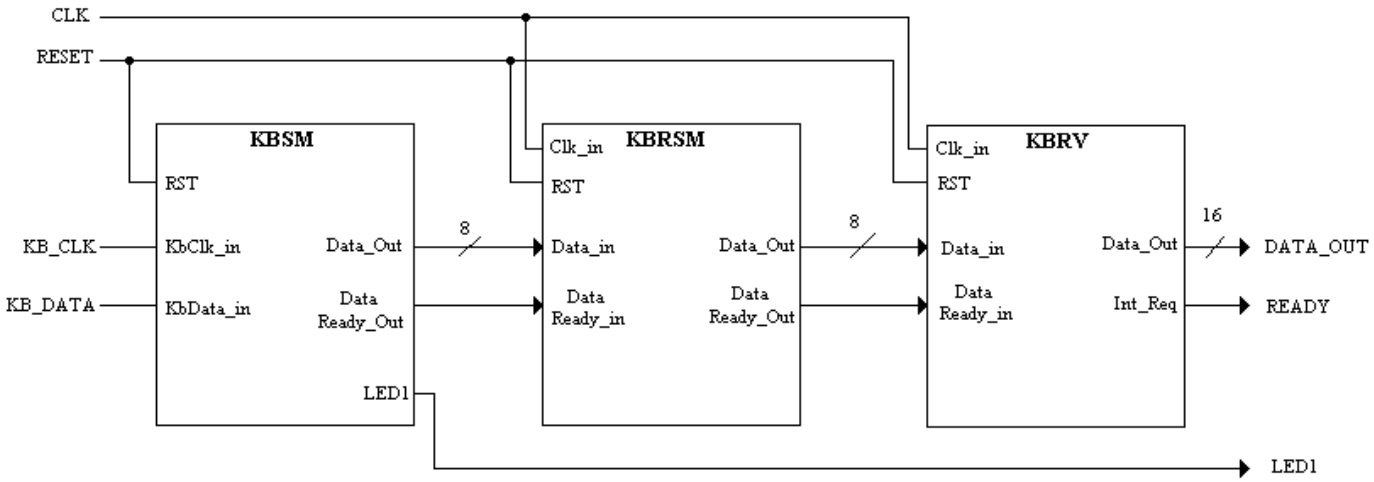
To demonstrate this program, it was necessary to develop interfaces in VHDL to the display and keyboard. Also, in order to step down the external clock from 50 MHz

down to 3.125 MHz that is suitable for the processor, a clock divider module needed to be developed. These are described below.

DISPLAY INTERFACE

The display interface is a simple module that waits for data to be sent to address 0200. This address is the first address mapped outside of the data memory of the processor. When data is sent to this address, the display interface updates its data register and routes the data through to the LCD display. When data is sent to other addresses, the module's data register remains unchanged.

KEYBOARD INTERFACE



The keyboard interface consists of three separate modules: KBD_SM, KBD_RSM and KB_DRV.

The first module, KBD_SM, is a state machine that reads data from the keyboard through the PS/2 port. Data is sent from the PS/2 device serially, using a data line and a clock line that originate from the keyboard. The module receives the data from the port, converts it to parallel data and alerts the next module via a READY line that there is valid data waiting to be processed.

The second module, KBD_RSM, synchronizes the keyboard data with the clock that runs the RISC processor. Since data is received serially from the keyboard at a frequency of roughly 15 kHz, and the processor runs at a speed of a few megahertz, it is necessary to synchronize the data with the processor. This module is small and consists

of a simple state machine that sends a READY signal to the next module with a one-shot, one-cycle pulse.

The third module, KB_DRV, interprets the data received from the keyboard and sends appropriate ASCII values to the processor. When a key is pressed on the keyboard, a "make code" is sent to the keyboard interface. When that key is released, the keyboard sends a "break code". This module sorts out the make and break codes, asserts the interrupt request line, and sends data to the processor.

HANGMAN PROGAM

The hangman program contains a set of 28 words for the user to guess. The user is presented with a series of underscores on the LCD display, one underscore per letter in the word to be guessed. When the user guesses a letter using the keyboard, the program checks to see if (a) there are any instances of that letter in the word, and (b) whether that letter had been guessed correctly before. The user has seven chances to guess incorrectly before the game ends, the word is displayed on the LCD, and a new word is selected for another game. With each successful guess, the letters appear on the LCD display in its proper location in the word. The LCD also informs the user of how many incorrect guesses are remaining. When all the letters in the word have been correctly guessed, the user is informed that the game has been won, and the user proceeds to a new word to guess.

CLOCK DIVIDER MODULE

The DLL_DIVR module was coded in VHDL to take advantage of a DLL inside the Spartan II chip to divide the external clock by 16 to produce a slower clock for the

processor to run on. While the processor is capable of running at speeds up to 25 MHz, it was decided to run it at 3.125 MHz. This is because many delays needed to be inserted between commands to the LCD, and these delays were simpler to code with a processor running at a slower speed. Also, the processor still runs fast enough that the user cannot notice any difference.

The internal DLL does all the work of producing a slower clock. All this module does is connect the DLL output back into the feedback input of the DLL, and connect the divided output of the DLL to a global clock buffer for distributing the clock signal throughout the FPGA.

Conclusion:

The main purpose of this project was fully achieved at the end. We were successful in designing a 16-bit RISC processor with a limited instruction set that is sufficient in itself. The capabilities of the FPGA and PLD technologies are ever increasing. If a FPGA based design does not work for the first time, it can often be fixed by changing the program and physically reprogramming the device, without changing any components or interconnections at the system level. The ease of prototyping and modifying FPGA based systems can eliminate the need for simulation in board-level design. Simulation is required only for chip-level design.

Thus we were fortunate to work with the latest trends in the chip industry. This project gave us the experience of VHDL programming, simulation and the actual FPGA programming. It also gave us the experience of designing in a team environment.

APPENDIX B

THE CONTROL UNIT

The following is a list of signals that are used in the control unit, along with a description of each:

RESET	system reset input
CLK	system clock
Don't Care	arbitrary value assigned to a particular signal

The following is a list of registers in the control unit, along with descriptions of each:

PC	16 bit program counter register
IR	21 bit instruction register
PCRC	16 bit program counter return from call register
PCRI	16 bit program counter return from interrupt register
FLAGS	16 bit flags register
INTV	16 bit interrupt vector register
INE1	interrupt enable step 1
INE2	interrupt enable step 2
INE3	interrupt enable step 3

In addition each register above has a corresponding XXXX_NEXT signal, where XXXX stands for the corresponding register name listed above. The XXXX_NEXT signals indicate what value the register will receive at the next rising clock edge. For example, the PC register is updated with the value of PC_NEXT at every clock cycle.

1. The main job of the control unit is to take its inputs, logically analyze them, and provide the rest of the processor's internal components with the adequate signals, to carry a desired operation.
    a. Inputs are:

INSTRUCTION	coming from the INSTRUCTION MEMORY
RF_IN	ALU signal, same input as that going to REGISTER FILE
RY_IMM	IMMEDIATE value coming from the MUX, which is controlled by control signal output C0
F_IN	FLAGS coming from ALU output
CLK	System clock
RESET	System reset
INT_REQ	External interrupt request signal

b. Output control signals are:

INSTRUCT_ADDR	signal going to the instruction memory ADDRESS input
F_OUT	Output of FLAGS register that has the option of serving as in input to the ALU by means of the MUX, which is controlled by control signal, output C6
OPERATION	4 bit output bus which serves as input to the ALU and determines the operation which the ALU must perform.
RZ_ADDR	Output that goes to the register file and serves as the RZ register address
RX_ADDR	Output that goes to the register file and serves as the RX register address
RY_ADDR	Output that goes to the register file and serves as the RY register address
C0	Signal going to the MUX that selects whether the “b” input to the ALU receives register Y from the REGISTER FILE or the sign-extended immediate value from the INSTRUCTION REGISTER
C1	Signal that activates the load enable on the register file
C2	Signal that activates the chip select on the data memory
C3	Signal which activates the read/~write option on the data memory
C4	2 bit signal that selects whether the REGISTER FILE input receives the output of the ALU, the data output of the DATA MEMORY, the immediate value from the INSTRUCTION, or data on the external data bus from an I/O device
C5	Signal that activates the chip select on the instruction memory
C6	Signal that selects whether the “a” input to the ALU receives register X from the REGISTER FILE or the FLAGS register from the CONTROL UNIT

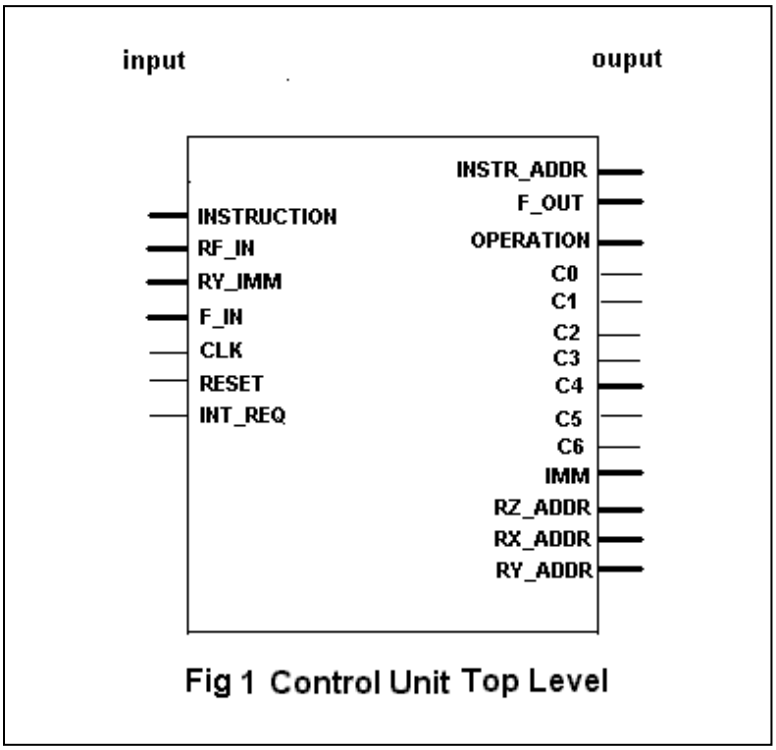


Fig 1 Control Unit Top Level

2. Preliminaries
    a. FLAGS is internally tied to output F_OUT
    b. IR (bits 12 through 15) are internally tied to output RZ_ADDR
    c. IR (bits 8 through 11) are internally tied to output RX_ADDR
    d. IR (bits 4 through 7) are internally tied to output RY_ADDR
    e. IR (bits 0 through 6) are internally tied to output IMM
    f. PC_NEXT is internally tied to output INSTR_ADDR
    g. When there is an interrupt request (via INT_REQ input), then such signal goes to FLAGS_NEXT register (bit 4)
    h. Control signal C6 output is manipulated by internal logic in the control unit, independent of a fetch or execute state (see 3 below):
        i. When C6 is high, the A input to ALU is allowed to come from the RX output from the register file. When C6 is low then the A input to ALU is allowed to come from the F_OUT output from the control unit.
        ii. If IR (bits 8 trough 11, RX address) are all low and internal signal R0_FLAGS is high then C6 control signal is low otherwise C6 is high.
            1. Internal signal R0_FLAGS is manipulated during the EXECUTE STATE and its value depends on the operations being executed.

3. The control unit is a two-state state machine
    a. Fetch state
    b. Execute state
    c. Processor boots on fetch state (meaning that when the system is RESET then the system will begin operations with a fetch state)
        i. Note: Control unit works on a fetch-execute, fetch-execute sequence

4. The sequential process of the control unit is affected by the RESET and the CLK
    a. When RESET is high the values for the control unit's registers are as follows:

        PC = 0
        IR = 0
        PCRC = 0
        PCRI = 0
        INTV = don't care
        FLAGS = 0
        INE 1 = 0
        INE 2 = 0
        INE 3 = 0
        STATE = FETCH

    b. Note: Since the state machine is one-hot encoded, technically there are two other possible states that could exist in the state machine (FETCH is state "01", EXECUTE is state "10", and the other non-used states are "00" and "11") . In the non-used states:
        i. All control signal C0 – C5 are set to don't cares arbitrary values
        ii. All zero values listed in 4.a. above are fed back into all registers except for STATE
        iii. STATE is sent to the FETCH STATE
    c. If not in RESET then the control unit updates the above registers at every CLK cycle as follows:

        PC updates to PC_NEXT value
        STATE updates to STATE_NEXT value
        IR updates to IR _NEXT value
        FLAGS updates to FLAGS_NEXT value
        PCRC updates to PCRC _NEXT value
        PCRI updates to PCRI _NEXT value
        INE 1 updates to INE 1_NEXT value
        INE 2 updates to INE 2_NEXT value
        INE 3 updates to INE 3_NEXT value

    d. In effect the control unit contains 9 different registers:

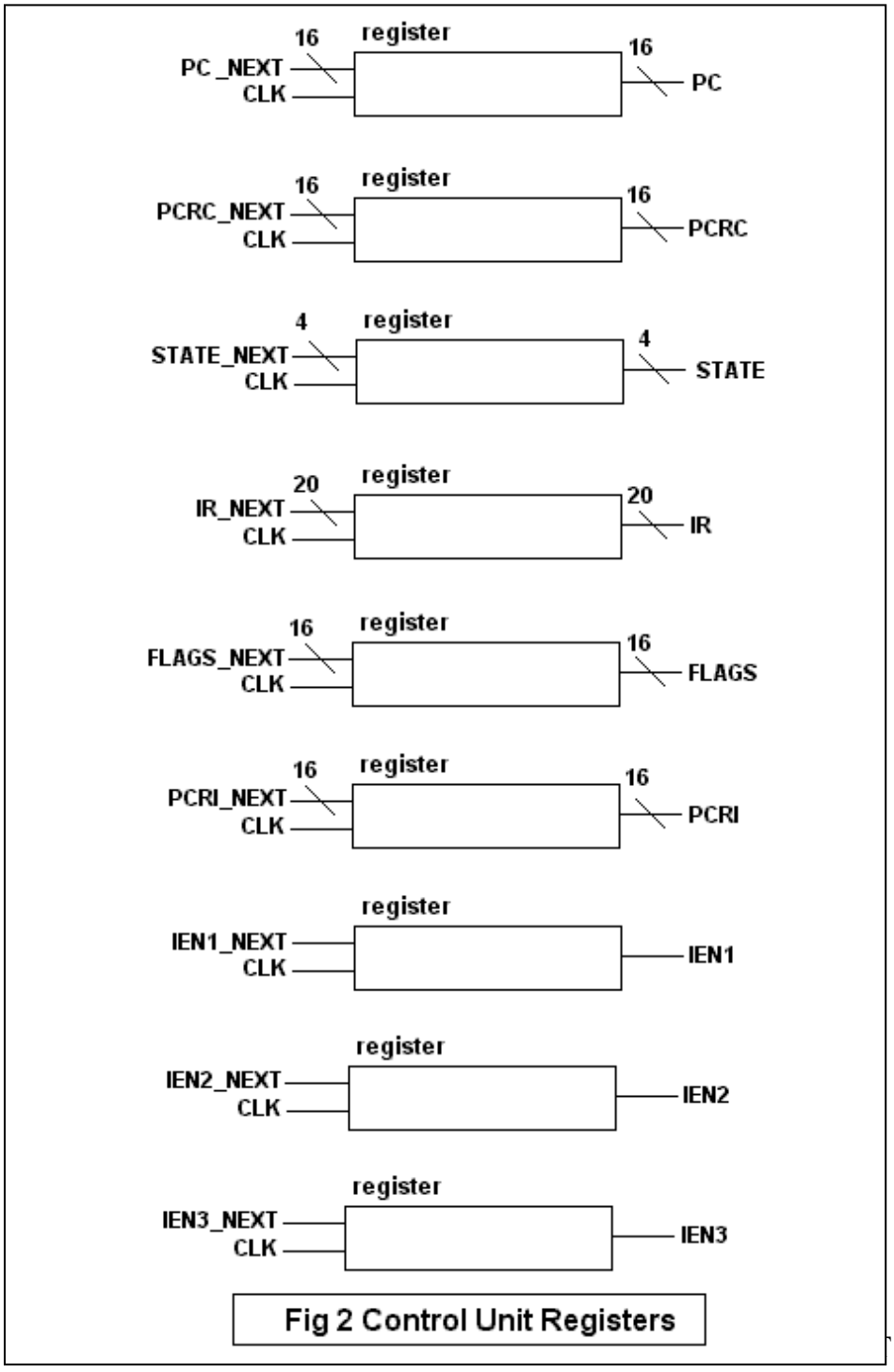


Fig 2 Control Unit Registers

The XXXX_NEXT signals are manipulated by the FETCH STATE and during the EXECUTE STATE as prescribed by the instruction being performed.

5. The FETCH STATE
    a. The main purpose of the fetch state is to get the next instruction from the instruction memory, on the subsequent clock cycle and to set the necessary control signals as required by both the controls unit's own internal operations as well as signals internal to it. Each instruction fetched is composed of a 21 bit external signal coming from the instruction memory (INSTRUCTION). The control unit can only fetch one instruction per FETCH cycle.
    b. There are two INSTRUCTION formats for the processor (format 1 and format 2)
        i. In format 1 the 21 bit ISTRUCTION signal is divided as follows:

Bit number →	< 20 – 16 >	<15 – 12>	< 11 – 8 >	< 7 – 4 >	< 3 – 0 >
	OPCODE	RZ address	RX address	RY address	Not used

        ii. In format 2 the 21 bit ISTRUCTION signal is divided as follows:

Bit number →	< 20 – 16 >	<15 – 12>	< 11 – 8 >	< 7 – 0 >
	OPCODE	RZ address	RX address	IMMIDIATE

    c. During the FETCH STATE the control unit sets its outputs and internal signals as follows:
        i. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	Don't care	ALU does not need to perform a function
C0	Don't care	Neither ALU nor Control unit need and immediate value or a register file input
C1	High	Register file does not need to be enabled
C2	Low	Data memory does not need to be enabled
C3	Don't care	Data memory does nothing
C4	Don't care	Register file is disabled therefore input to it does not matter
C5	High	Allow the control unit to receive instruction from the instruction memory

ii. Control unit's internal signals

Signal	Assigned value	Reason for Assigned Value
IR_NEXT	INSTRUCTION	Allow instruction from the instruction memory to be used IR on the next clock cycle
FLAGS_NEXT	FLAGS	Flags are updated
PC_NEXT	PC + 1	Increment program counter
R0_FLAGS	Don't care	Input to the ALU is irrelevant
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector
IEN1_NEXT	IEN1	Update interrupt enable 1
IEN2_NEXT	IEN2	Update interrupt enable 2
IEN3_NEXT	IEN3	Update interrupt enable 3
STETE_NEXT	EXECUTE_STATE	Allow the execute state to begin next clock cycle

6. The execute state
    a. At the beginning of this cycle the STATE_NEXT internal signal is assigned FETCH_STATE so that at the next CLK the control unit proceeds to fetch the next instruction. And IR_NEXT is allowed to maintain its current instruction value IR.
    b. Control signal C5 is always kept high during this state.
    c. It is during this cycle that the CPU executes an operation or function; it is during this state that the CPU does "work". The control unit must send the appropriate signals to every other CPU component involved in executing the fetched function. Each operation is contained within the OPCODE of the INSTRUCTION as outlined in 5.b.
    d. In effect during the execute cycle the control unit reads the OPCODE, of the newly fetched instruction, and set its internal signals and output signals according to the needs of the specific function. Note the control unit can only execute one function per execute cycle.
    e. There are a total of 20 OPCODE functions; these functions in effect constitute the CPU's instruction set.
        i. Each of the 20 OPCODE instructions will not be described below:

1. ADD
    a. The purpose of the add instruction is to have the ALU perform an ADD function given an A and a B value for its inputs.
    b. Usage:
        i. Typical: RX + RY and result stored in RZ using the register file.
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_ADD	ALU needs to receive signal to perform and add
C0	Low	B input to ALU comes from RY of the register file
C1	Low	Register file is enabled
C2	Low	Data memory does not need to be enabled
C3	Don't care	Data memory does nothing
C4	Low, Low	RF_IN is allowed to come from the ALU

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	F_IN	Flags next is given the value of the F_IN as set by the ALU
PC_NEXT	PC	Program counter is not incremented
R0_FLAGS	Low	Allow A input to ALU come from either RX of the register file
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

2. SUB
    a. The purpose of the SUB instruction is to have the ALU perform a SUBtract function given A and B value for its inputs.
    b. Usage:
        i. Typical: RX - RY and result stored in RZ using the register file.
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_SUB	ALU needs to receive signal to perform and subtraction
C0	Low	B input to ALU comes from RY of the register file
C1	Low	Register file is enabled
C2	Low	Data memory does not need to be enabled
C3	Don't care	Data memory does nothing
C4	Low, Low	RF_IN is allowed to come from the ALU

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	F_IN	Flags next is given the value of the F_IN as set by the ALU
PC_NEXT	PC	Program counter is not incremented
R0_FLAGS	Low	Allow A input to ALU come from either RX of the register file
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

3. AND
    a. The purpose of the AND instruction is to have the ALU perform the logical AND function given an A and B value for its inputs.
    b. Usage:
        i. Typical: RX & RY and result stored in RZ using the register file.
        ii. Manipulate the FLAGS stored in the control unit. If in the INTRUCTION the RX_ADDR is '0' then the flags will be appear on the A input to ALU and ANDed to what the ALU has on its B input and result stored in RZ using the register file
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_AND	ALU needs to receive signal to perform the AND operation
C0	Low	B input to ALU comes from RY of the register file
C1	Low	Register file is enabled
C2	Low	Data memory does not need to be enabled
C3	Don't care	Data memory does nothing
C4	Low, Low	RF_IN is allowed to come from the ALU

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	F_IN or (RF_IN except for bit 4)	FLAGS_NEXT is given the value of the F_IN as set by the ALU when RZ_ADDR is not '0' otherwise FLAGS_NEXT will be (RF_IN except for bit 4)
PC_NEXT	PC	Program counter is not incremented
R0_FLAGS	High	Allow A input to ALU come from either RX of the register file or F_OUT from control unit output
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

4. OR
    a. The purpose of the OR instruction is to have the ALU perform the logical OR function given an A and B value for its inputs.
    b. Usage:
        i. Typical: RX | RY and result stored in RZ using the register file.
        ii. Manipulate the FLAGS stored in the control unit. If in the INTRUCTION the RX_ADDR is '0' then the flags will be appear on the A input to ALU and ORed to what the ALU has on its B input and result stored in RZ using the register file
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_OR	ALU needs to receive signal to perform and OR
C0	Low	B input to ALU comes from RY of the register file
C1	Low	Register file is enabled
C2	Low	Data memory does not need to be enabled
C3	Don't care	Data memory does nothing
C4	Low, Low	RF_IN is allowed to come from the ALU

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	F_IN or (RF_IN except for bit 4)	FLAGS_NEXT is given the value of the F_IN as set by the ALU when RZ_ADDR is not '0' otherwise FLAGS_NEXT will be (RF_IN except for bit 4)
PC_NEXT	PC	Program counter is not incremented
R0_FLAGS	High	Allow A input to ALU come from either RX of the register file or F_OUT from control unit output
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

5. XOR
    a. The purpose of the XOR instruction is to have the ALU perform an eXclusive OR function given an A and B value for its inputs.
    b. Usage:
        i. Typical: RX □ RY result is stored in RZ using the register file.
        ii. Manipulate the FLAGS stored in the control unit. If in the INTRUCTION the RX_ADDR is '0' then the flags will be appear on the A input to ALU and exclusive ORed to what the ALU has on its B input and result stored in RZ using the register file
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_XOR	ALU needs to receive signal to perform the eXclusive OR
C0	Low	B input to ALU comes from RY of the register file
C1	Low	Register file is enabled
C2	Low	Data memory does not need to be enabled
C3	Don't care	Data memory does nothing
C4	Low, Low	RF_IN is allowed to come from the ALU

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	F_IN or (RF_IN except for bit 4)	FLAGS_NEXT is given the value of the F_IN as set by the ALU when RZ_ADDR is not '0' otherwise FLAGS_NEXT will be (RF_IN except for bit 4)
PC_NEXT	PC	Program counter is not incremented
R0_FLAGS	High	Allow A input to ALU come from either RX of the register file or F_OUT from control unit output
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

6. NOT
    a. The purpose of the NOT instruction is to have the ALU invert the A input.
    b. Usage:
        i. Typical: ~RZ and result stored in RZ using the register file.
        ii. Manipulate the FLAGS stored in the control unit. If in the INTRUCTION the RX_ADDR is '0' then the flags will be appear on the A input to ALU and inverted and the result stored in RZ using the register file
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_NOT	ALU needs to receive signal to perform a NOT
C0	Don't Care	B input to ALU is irrelevant
C1	Low	Register file is enabled
C2	Low	Data memory does not need to be enabled
C3	Don't care	Data memory does nothing
C4	Low, Low	RF_IN is allowed to come from the ALU

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	F_IN or (RF_IN except for bit 4)	FLAGS_NEXT is given the value of the F_IN as set by the ALU when RZ_ADDR is not '0' otherwise FLAGS_NEXT will be (RF_IN except for bit 4)
PC_NEXT	PC	Program counter is not incremented
R0_FLAGS	High	Allow A input to ALU come from either RX of the register file or F_OUT from control unit output
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

7. SHL
    a. The purpose of the SHL instruction is to have the ALU perform a SHift Left function given an A and a B value for its inputs.
    b. Usage:
        i. Typical: RX shifts to the left by one bit and RX bit zero is replaced by RY bit zero and result stored in RZ using the register file.
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_SHL	ALU needs to receive signal to perform and shift left
C0	Low	B input to ALU comes from RY of the register file
C1	Low	Register file is enabled
C2	Low	Data memory does not need to be enabled
C3	Don't care	Data memory does nothing
C4	Low, Low	RF_IN is allowed to come from the ALU

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	F_IN	Flags next is given the value of the F_IN as set by the ALU
PC_NEXT	PC	Program counter is not incremented
R0_FLAGS	Low	Allow A input to ALU come from either RX of the register file
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

8. SHR
    a. The purpose of the SHR instruction is to have the ALU perform a SHift Right function given an A and a B value for its inputs.
    b. Usage:
        i. Typical: RX shifts to the right by one bit and RX bit 15 is replaced by RY bit 0 and result stored in RZ using the register file.
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_SHR	ALU needs to receive signal to perform and shift right
C0	Low	B input to ALU comes from RY of the register file
C1	Low	Register file is enabled
C2	Low	Data memory does not need to be enabled
C3	Don't care	Data memory does nothing
C4	Low, Low	RF_IN is allowed to come from the ALU

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	F_IN	Flags next is given the value of the F_IN as set by the ALU
PC_NEXT	PC	Program counter is not incremented
R0_FLAGS	Low	Allow A input to ALU come from either RX of the register file
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

9. ROR
    a. The purpose of the ROR instruction is to have the ALU perform a ROtate Right function given an A input.
    b. Usage:
        i. Typical: RX rotates to the right by one bit and RX bit 15 is replaced by RX bit 0 and result stored in RZ using the register file.
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_ROR	ALU needs to receive signal to perform rotate right
C0	Low	B input to ALU comes from RY of the register file
C1	Low	Register file is enabled
C2	Low	Data memory does not need to be enabled
C3	Don't care	Data memory does nothing
C4	Low, Low	RF_IN is allowed to come from the ALU

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	F_IN	Flags next is given the value of the F_IN as set by the ALU
PC_NEXT	PC	Program counter is not incremented
R0_FLAGS	Low	Allow A input to ALU come from either RX of the register file
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

10. ROL
    a. The purpose of the ROL instruction is to have the ALU perform a ROtate left function given an A input.
    b. Usage:
        i. Typical: RX rotates to the left by one bit and RX bit 0 is replaced by RX bit 15 and result stored in RZ using the register file.
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_ROL	ALU needs to receive signal to perform rotate left
C0	Low	B input to ALU comes from RY of the register file
C1	Low	Register file is enabled
C2	Low	Data memory does not need to be enabled
C3	Don't care	Data memory does nothing
C4	Low, Low	RF_IN is allowed to come from the ALU

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	F_IN	Flags next is given the value of the F_IN as set by the ALU
PC_NEXT	PC	Program counter is not incremented
R0_FLAGS	Low	Allow A input to ALU come from either RX of the register file
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

11. STORE
    a. The purpose of the STORE instruction is to store the RZ to either DATA MEMORY or external I/O devices
    b. Usage:
        i. Typical: RX + RY result used as ADDRESS to the DATA MEMORY or I/O device where the data will be stored. RZ goes into DATA MEMORY as an input.
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_ADD	ALU needs to receive signal to perform and add
C0	Low	B input to ALU comes from RY of the register file
C1	High	Register file load enable is disabled
C2	High	Data memory is enabled
C3	Low	Data memory does a “write”
C4	Low, Low	RF_IN is allowed to come from the ALU

    d. Control unit’s internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	FLAGS	Flags are updated
PC_NEXT	PC	Program counter is not incremented
R0_FLAGS	Low	Allow A input to ALU come from RX of the register file
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

12. LOAD
    a. The purpose of the LOAD instruction is to load data into the REGISTER FILE from either DATA MEMORY or external I/O devices
    b. Usage:
        i. Typical: RX + RY result used as ADDRESS to the DATA MEMORY from where data is to be loaded. RZ obtains output from DATA MEMORY or I/O device.
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_ADD	ALU needs to receive signal to perform and add
C0	Low	B input to ALU comes from RY of the register file
C1	Low	Register file load enable is enabled
C2	High	Data memory is enabled
C3	Low	Data memory does a “read”
C4	Low, High	RF_IN is allowed to come from DATA MEMORY

    d. Control unit’s internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	FLAGS	Flags are updated
PC_NEXT	PC	Program counter is not incremented
R0_FLAGS	Low	Allow A input to ALU come from RX of the register file
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

13. LUI
    a. The purpose of the LUI instruction is to load the upper 8 bits from the IMMIDIATE value of the INSTRUCTION into the register file.
    b. Usage:
        i. Typical: RZ get bits 8 to 15 = IMMIDIATE bits 0 to 7 remain unchanged
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	Don't Care	ALU does nothing
C0	Don't Care	No B input to the ALU is needed
C1	Low	Register file load enable is enabled
C2	Low	Data memory is disabled
C3	Don't Care	Data memory does nothing
C4	High, Low	RF_IN is allowed to come IMMIDIATE value

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	FLAGS	Flags are updated
PC_NEXT	PC	Program counter is not incremented
R0_FLAGS	Don't Care	ALU does not need input
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV or RF_IN bits < 15 – 8 >	If RZ is '0' then INTV_NEXT gets RF_IN bits < 15 – 8 > else INTV_NEXT gets INTV

14. ADDI
    a. The purpose of the ADD Immediate instruction is to have the ALU perform an ADD function given an A and a B value that comes from the immediate portion of the INSTRUCTION for its inputs.
    b. Usage:
        i. Typical: RX + IMMIDIATE result is stored in RZ using the register file.
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_ADD	ALU needs to receive signal to perform and add
C0	High	B input to ALU comes from IMMIDIATE value
C1	Low	Register file is enabled
C2	Low	Data memory does not need to be enabled
C3	Don't care	Data memory does nothing
C4	Low, Low	RF_IN is allowed to come from the ALU

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	F_IN	Flags next is given the value of the F_IN as set by the ALU
PC_NEXT	PC	Program counter is not incremented
R0_FLAGS	Low	Allow A input to ALU come from either RX of the register file
PCRC_NEXT	PC	Program counter is stored in PCRC
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV or RF_IN	If RZ is '0' then INTV_NEXT gets RF_IN, else INTV_NEXT gets INTV

15. CALL
    a. The purpose of the CALL instruction is to CALL a subroutine with the RY value from the output of the register file going to the program counter.
    b. Usage:
        i. Calling a subroutine by altering the value of the program counter
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	Don't Care	ALU does nothing
C0	Low	B input to ALU comes from RY of the register file
C1	High	Register file load enable is disabled
C2	Low	Data memory is disabled
C3	Don't Care	Data memory does nothing
C4	Don't Care	RF_IN is irrelevant

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	FLAGS	Flags are updated
PC_NEXT	RY_IMM	Program counter gets RY_IMM input value
R0_FLAGS	Don't Care	ALU does not need input
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

16. RET
    a. The purpose of the RET instruction is to RETurn from a subroutine with the PCRC value from the control unit register going to the program counter.
    b. Usage:
        i. Returning from a subroutine by altering the value of the program counter
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	Don't Care	ALU does nothing
C0	Don't Care	B input to ALU is irrelevant
C1	High	Register file load enable is disabled
C2	Low	Data memory is disabled
C3	Don't Care	Data memory does nothing
C4	Don't Care	RF_IN is irrelevant

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	FLAGS	Flags are updated
PC_NEXT	PCRC	Allow program counter to get value stored in PCRC
R0_FLAGS	Don't Care	ALU does not need input
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

17. BTSS
    a. The purpose of the Bit Test Skip if Set instruction is to have the program counter skip an instruction if bits in a mask are set.
    b. Usage:
        i. Typical: Register RX is ANDed with the IMMIDIATE value, and an instruction is skipped if the zero flag is zero (meaning that a bit in the mask is set)
        ii. If in the INTRUCTION the RX_ADDR is '0' then the FLAGS register is read and modified instead of a register from the REGISTER FILE. This allows bit testing of any bit in the FLAGS register
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_AND	ALU needs to receive signal to perform the AND operation
C0	High	B input to ALU comes from IMMIDIATE value
C1	High	Register file is load enable is disabled
C2	Low	Data memory does not need to be enabled
C3	Don't care	Data memory does nothing
C4	Don't care	RF_IN is irrelevant

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	F_IN	Flags next is given the value of the F_IN as set by the ALU
PC_NEXT	PC or PC + 1	Program counter is incremented by one causing the next instruction to be skipped if zero flag is set if not program counter does not get incremented
R0_FLAGS	High	Allow A input to ALU come from either RX of the register file or F_OUT from control unit output
PCRC_NEXT	PC	Program counter is stored in PCRC
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

18. BTSC
    a. The purpose of the Bit Test Skip if Clear instruction is to have the program counter skip an instruction if bits in a mask are cleared.
    b. Usage:
        i. Typical: Register RX is ANDed with the IMMIDIATE value, and an instruction is skipped if the zero flag is one (meaning that a bit in the mask is cleared)
        ii. If in the INTRUCTION the RX_ADDR is '0' then the FLAGS register is read and modified instead of a register from the REGISTER FILE. This allows bit testing of any bit in the FLAGS register
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	ALU_AND	ALU needs to receive signal to perform the AND operation
C0	High	B input to ALU comes from IMMIDIATE value
C1	High	Register file is load enable is disabled
C2	Low	Data memory does not need to be enabled
C3	Don't care	Data memory does nothing
C4	Don't care	RF_IN is irrelevant

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	F_IN	Flags next is given the value of the F_IN as set by the ALU
PC_NEXT	PC or PC + 1	Program counter is incremented by one causing the next instruction to be skipped if zero flag is clear if not program counter does not get incremented
R0_FLAGS	High	Allow A input to ALU come from either RX of the register file or F_OUT from control unit output
PCRC_NEXT	PC	Program counter is stored in PCRC
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

19. JR

a. The purpose of the Jump Relative instruction is to have the program counter increment by the RY_IMM input.
b. Usage:
    i. Next instruction fetched will be the current instruction plus RY_IMM
c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	Don't Care	ALU does nothing
C0	High	B input to ALU comes from IMMIDIATE value
C1	High	Register file is load enable is disabled
C2	Low	Data memory does not need to be enabled
C3	Don't Care	Data memory does nothing
C4	Don't Care	RF_IN is irrelevant

d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	FLAGS	Flags are updated
PC_NEXT	PC + RY_IMM	Program counter is not incremented by value stored from IMMEDIATE
R0_FLAGS	Don't Care	ALU does not need input
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

20. JA
    a. The purpose of the Jump Absolute instruction is to have the program counter receive the value stored in a register from the REGISTER FILE specified by RY.
    b. Usage:
        i. Next instruction fetched will be from the address specified by RY_ADDR
        ii. If in the INTRUCTION the RX_ADDR is '0' then this instruction serves as a RETURN FROM INTERRUPT. In this case, the value in PCRI is stored into PC
    c. Control unit outputs

Output	**Assigned value**	**Reason for Assigned Value**
OPERATION	Don't Care	ALU does nothing
C0	Low	B input to ALU RY value of register file
C1	High	Register file is load enable is disabled
C2	Low	Data memory does not need to be enabled
C3	Don't Care	Data memory does nothing
C4	Don't Care	RF_IN is irrelevant

    d. Control unit's internal signals

Signal	**Assigned value**	**Reason for Assigned Value**
FLAGS_NEXT	FLAGS	Flags are updated
PC_NEXT	PC or RY_IMM	If RY is '0' then PC_NEXT is PCRI else PC_NEXT is RY_IMM
R0_FLAGS	Don't Care	ALU does not need input
PCRC_NEXT	PCRC	Update return call register
PCRI_NEXT	PCRI	Update return from interrupt register
INTV_NEXT	INTV	Update interrupt vector

7. Interrupt
    a. The interrupt allows the servicing of an external request to jump to a subroutine
        i. The external request comes from the INT_REQ input to FLAGS bit 4
        ii. If the interrupt has been enabled for 4 clock cycles (where the IEN bit in the FLAGS register is high, as are the three IEN step registers IEN1, IEN2 and IEN3) then PC NEXT get the INTV and PC is stored in PCRI
            1. This allows for the interrupt subroutine to get serviced.