Building A RISC Microcontroller in an FPGA

Yap Zi He

yapzihe@hotmail.com

http://www.opencores.org/projects/riscmcu/

Supervisor : Muhammad Mun'im Ahmad Zabidi (Assoc Prof)

raden@utm.my

Faculty of Electrical Engineering,

Universiti Teknologi Malaysia

1 March 2002

Specially Dedicated To

My Beloved Parents

Yap Hong Chun, Lim Ah Mooi

And All My Friends

CHAPTER I

INTRODUCTION

1.1 Central Processing Unit

Figure 1.1 shows the block diagram of a basic computer system. A basic computer system must have the standard elements CPU, memory and I/O. All these elements communicate via the system bus, which is composed by the data, address and control buses.

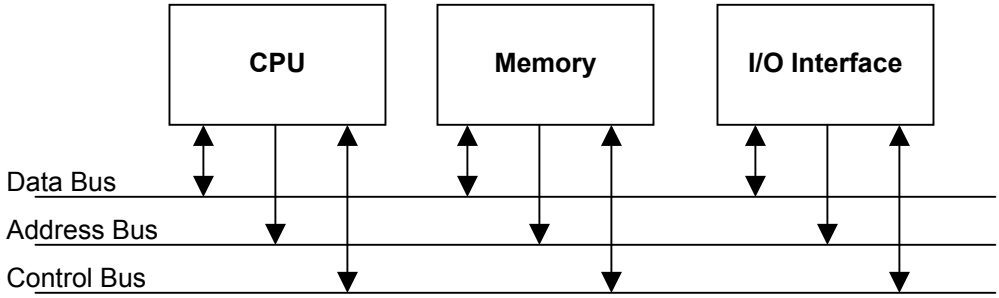


Figure 1.1 Basic Computer System

The CPU, as the 'brain' of the computer, administers all the activity in the system and performs all operations on data. The CPU has the ability to understand and execute instructions based on a set of binary codes, each representing a simple operation. These instructions are usually arithmetic, logic, data movement, or branch operations,

and are represented by a set of binary codes called the instruction set. The memory, is used to store all the programs formed by the instruction set and all the require data. I/O interface provide an interconnection with the outside world, such as the keyboard as an input and the monitor as an output.

Minicomputers and mainframe computers, have CPUs consisting multiple ICs, ranging from several ICs (minicomputers) to several circuit boards of ICs (mainframes). This is necessary to achieve the high speeds and computational power of larger computers. On the other hand, the CPU of a microcomputer is contained in a single integrated circuit. They are known as a microprocessor.

1.2 Microcontroller

It was pointed out above that microprocessors are single-chip CPUs used in microcomputer. A microcontroller contains, in a single IC, a CPU and much of the remaining circuitry of a basic computer system. A microcontroller has the CPU, memory (RAM, ROM) and the I/O interface (parallel, serial) all within the same IC. Of course, the amount of on-chip memory does not approach that of even a modest microcomputer system.

Microprocessors are most commonly used as the CPU in microcomputer systems. Microcontrollers, on the other hand, are found in small, minimum-component designs performing control-oriented activities, such as the traffic lights. These designs were often implemented in the past using dozens or even hundreds of ICs. A microcontroller aids in reducing the overall component count. All that is requires is microcontroller, a small number of support components, and a control program in ROM.

1.3 Objectives

The main objective of this project is to design a RISC microcontroller using VHDL and implement it in an FPGA. The microcontroller instruction set and features are based on Atmel AVR AT90S1200 RISC microcontroller.

1.4 Atmel AVR AT90S1200

The AT90S1200 is a low-power CMOS 8-bit microcontroller based on the AVR RISC architecture. It has 89 powerful instructions and 32 general purpose registers. Most instructions are executed in one cycle and so it can achieve up to 12 MIPS throughput at 12 MHz. The microcontroller also come with 1K Bytes of in-system programmable flash as the program memory and 64 bytes of in-system programmable EEPROM.

The AT90S1200 is equipped with one 8-bit timer/counter with separate prescaler, one on-chip analog comparator, a watchdog timer with on-chip oscillator and SPI for in system programming. It also features the external and internal interrupt. There are a total of 15 programmable I/O lines.

The IC come in 20-pin PDIP and SOIC with 2 speed grades, 0 - 4 MHz for AT90S1200-4 and 0 – 12 MHz for AT90S1200-12.

1.5 Project Background

Wan Mohd Khalid did a similar project titled “FPGA Implementation of a RISC microcontroller”. The design is also based on Atmel AVR AT90S1200 microcontroller. The project is designed using both VHDL and schematics. Only 50% of the instructions are designed using VHDL behavioral approach, which results in large area and slow performance. Parallel ports, timer, external interrupt and other peripheral features are not included. The project size is so large that it requires 3 pieces of Altera EPF10K20.

1.6 Work Scope

The aim of the project is to design the complete Atmel AVR AT90S1200. The microcontroller must be able to fit into the targeted FPGA device, which is Altera EPF10K20, provided in Altera UP1 Education Board. Features which cannot be implemented on an FPGA (analog comparator, pull-up resistors, etc) and which are not critical to the operation of the CPU (watchdog reset, etc) will be ignored.

CHAPTER II

LITERATURE REVIEW

2.1 Complex Instruction Set Computer (CISC)

In early days, computers had only a small number of instructions and used simple instruction sets, forced mainly by the need to minimize the hardware used to implement them. As digital hardware become cheaper, computer instructions tended to increase both in number and complexity. These computers also employ a variety of data types and a large number of addressing modes. A computer with a large number of instructions, are known as complex instruction set computer, abbreviated CISC.

Major characteristics of CISC architecture are:

- A large number of instructions – typically from 100 to 250 instructions
- Some instructions that perform specialized tasks and are used infrequently
- A large variety of addressing modes – typically from 5 to 20 different modes
- Variable-length instruction formats
- Instructions that manipulate operands in memory

2.2 Reduce Instruction Set Computer (RISC)

In the early 1980s, a number of computer designers were questioning the need for complex instruction sets used in the computer of the time. In studies of popular computer systems, almost 80% of the instructions are rarely being used. So they recommended that computers should have fewer instructions and with simple constructs. This type of computer is classified as reduced instruction set computer or RISC. The term CISC is introduced later to differentiate computers designed using the 'old' philosophy.

According to Daniel Tabak (1990), the first characteristic of RISC is the uniform series of single cycle, fetch-and-execute operations for each instruction implemented on the computer system being developed.

A single-cycle fetch can be achieved by keeping all the instructions a standard size. The standard instruction size should be equal to the number of data lines in the system bus, connecting the memory (where the program is stored) to the CPU. At any fetch cycle, a complete single instruction will be transferred to the CPU. For instance, if the basic word size is 32 bits, and the data port of the system bus (the data bus) has 32 lines, the standard instruction length should be 32-bits.

Achieving uniform (same time) execution of all instructions is much more difficult than achieving a uniform fetch. Some instructions may involve simple logical operations on a CPU register (such as clearing a register) and can be executed in a single CPU clock cycle without any problem. Other instructions may involve memory access (load from or store to memory, fetch data) or multicycle operations (multiply, divide, floating point), and may be impossible to be executed in a single cycle.

Ideally, we would like to see a streamlined and uniform handling of all instructions, where the fetch and the execute stages take up the same time for any instruction, desirably, a single cycle. This is basically one of the first and most important

principles inherent in the RISC design approach. All instructions go from the memory to the CPU, where they get executed, in a constant stream. Each instruction is executed at the same pace and no instruction is made to wait. The CPU is kept busy all the time.

Thus, some of the necessary conditions to achieve such a streamlined operation are:

- Standard, fixed size of the instruction, equal to the computer word length and to the width of the data bus.
- Standard execution time of all instructions, desirably within a single CPU cycle.

While it might not practical to hope that all instructions will execute in a single cycle, one can hope that at least 75% should.

Which instructions should be selected to be on the reduced instruction list? The obvious answer is: the ones used most often. It has been established in a number of earlier studies that a relatively small percentage of instructions (10 – 20%) take up about 80% – 90% of execution time in an extended selection of benchmark programs. Among the most often executed instructions were data moves, arithmetic and logic operations.

As mentioned earlier, one of the reasons preventing an instruction from being able to execute in a single cycle is the possible need to access memory to fetch operands and/or store results. The conclusion is therefore obvious – we should minimize as much as possible the number instructions that access memory during the execution stage. This consideration brought forward the following RISC principles:

- Memory access, during the execution stage, is done by load/store instructions only.
- All operations, except load/store, are register-to-register, within the CPU.

Most of the CISC systems are microprogrammed, because of the flexibility that microprogramming offers the designer. Different instructions usually have microroutines of different lengths. This means that each instruction will take a number of different cycles to execute. This contradicts the principle of a uniform, streamlined handling of all instructions. An exception to this rule can be made when each instruction has a one-to-one correspondence with a single microinstruction. That is, each microroutine consists of a single control word, and still let the designer benefit from the advantages of microprogramming. However, contemporary CAD tools allow the designer of hardwired control units almost as easy as microprogrammed ones. This enables the single cycle rule to be enforced, while reducing transistor count.

In order to facilitate the implementation of most instruction as register-to register operations, a sufficient amount of CPU general purpose registers has to be provided. A sufficiently large register set will permit temporary storage of intermediate results, needed as operands in subsequent operations, in the CPU register file. This, in turn, will reduce the number of memory accesses by reducing the number of load/store operations in the program, speeding up its run time. A minimal number of 32 general purpose CPU registers has been adopted, by most of the industrial RISC system designers.

The characteristics of RISC architecture are summarized as follow:

- Single-cycle instruction execution
- Fixed-length, easily decoded instruction format
- Relatively few instructions
- Relatively few addressing modes
- Memory access limited to load and store instructions
- All operations done within the registers of the CPU
- Hardwired rather than microprogrammed control unit
- Relatively large (at least 32) general purpose register file

CHAPTER III

DESIGN METHODOLOGY AND CAD TOOLS

3.1 Design Process

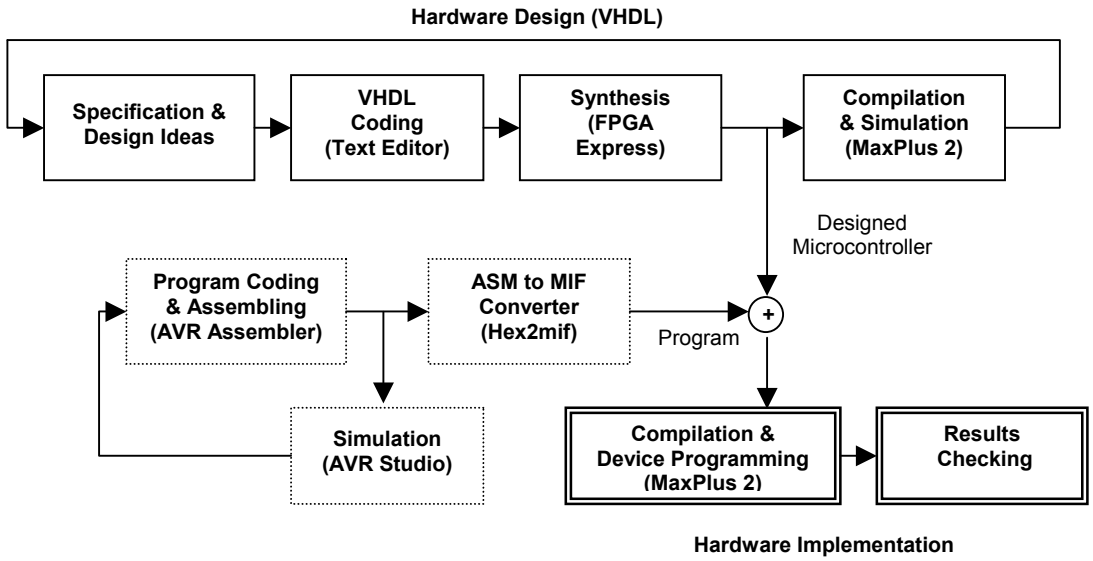


Figure 3.1 Design Process Flow

Figure 3.1 shows the design process of the project and their related CAD tools. The design process can be divided into 2 main parts – hardware design (with VHDL) and hardware implementation.

Hardware design is done with the related CAD tools. The first step in the hardware design is to prepare the specification of the design (the microcontroller). The architecture and the instruction set must be understood thoroughly. The design ideas are then describe with VHDL in a text editor. Then, the VHDL code is synthesized with FPGA Express. If synthesized successfully, FPGA express will generate a netlist files (EDF file). This file is then send to Max+Plus II for compilation and simulation. Results are verified by simulation. The hardware design process is repeated until the microcontroller is complete without any errors.

Hardware implementation is performed by downloading the design into the targeted FPGA device, Altera EPF10K20RC240-4. The hardware implementation tests the design in real physical environment by some control applications. A microcontroller can perform thousands of control applications. For every application, different programs must be written and store into the program ROM of the microcontroller before it can do the job. So, before the microcontroller is downloaded into the FPGA device, the specific program for the application must be written.

The program is written and assembled using the AVR Assembler. The AVR Studio is used to simulate and test the program. If no bugs are found, the program HEX file generated by the assembler is converted to MIF format with a tool written by the author, HEX2MIF. This MIF file, together with the EDF file of the complete microcontroller is then send to Max+Plus II for compilation and device programming. Once programmed into the device, the FPGA is reset to execute the application.

3.2 Synopsys FPGA Express

Synopsys FPGA Express is an industrial strength VHDL synthesis tool and is used to synthesize this project. First, VHDL files are written in a text editor such as the

Windows Notepad Editor. Then all the files are loaded in a project in FPGA Express. It will check the VHDL file for syntax errors. If there are no errors, we can ask FPGA Express to create implementation for the project. Once the implementation is created, the EDF net list file of the implementation can be exported and used by MAX+plus II for compilation.

3.3 MAX+Plus II

MAX+Plus II is a free software provided by Altera. It has many sub components and the important components are the compiler, simulator, waveform editor and programmer.

3.3.1 Compiler

The compiler consists of 6 sub modules - Compiler Netlist Extractor, Data Base, Logic Synthesizer, Fitter, Timing SNF Extractor and Assembler. All of them play an equally important role of compiling the EDF file into a simulation netlist file - SNF.

3.3.2 Simulator and Waveform Editor

After the EDF file is compiled, the generated SNF file will contain information of the circuit behavior and can be imported by the Simulator. The waveform editor let the user draws the pattern of the input waveform. The simulator then generates the output waveform based on the SNF file.

3.3.3 Programmer

The programmer is a tool used to download the compiled design into the FPGA device. The compiler will generate a SOF file which contains information to be written into the FPGA device (FLEX10K20). The programmer will program the SOF file contents into the FPGA via a PC parallel port using the ByteBlaster cable.

3.4 AVR Assembler

AVR Assembler is provided by Atmel to write and assemble programs for all the Atmel AVR RISC microcontrollers. The instruction set of this design is compatible with the Atmel AVR AT90S1200, so the assembler can also be used in this project. The assembler will assemble a program to create HEX and OBJ files.

3.5 AVR Studio

AVR Studio is a simulator for all Atmel AVR microcontrollers. It takes the OBJ file created by AVR Assembler. The simulator simulates the flow of instruction in the program one by one and the changes on the general purpose registers, memory contents, flags and I/O can be observed.

3.6 HEX2MIF

HEX2MIF is a HEX to MIF converter used to convert the HEX file generated by AVR Assembler into a MIF file. MIF files are used to define the initial value for the memory components in Max+Plus II. This simple program is written by the author with C and the source code is listed in Appendix C.

CHAPTER IV

INSTRUCTION SET

4.1 Instruction Set Summary

The operation of the CPU is determined by the instruction it executes, referred to as machine instructions or computer instructions. The collection of different instructions that the CPU can execute is referred to as the CPU's instruction set. Since the instruction set defines the datapath and everything else in a processor, it is necessary to study it first.

Table 4.1 shows the instruction set summary of the designed microcontroller, while the instruction set summary of the original AT90S1200 is shown in Appendix D. There are 92 instructions grouped into 4 categories: arithmetic and logic instructions, branch instructions, data transfer instructions and the bit and bit-test instructions. As mentioned earlier, instruction set of the design is based on Atmel AVR AT90S1200 instruction set. In this way, the design can use the same assembler and simulator provided by Atmel since the final design is actually an AT90S1200 compatible microcontroller.

One of the RISC characteristics mentioned earlier is single-cycle execution for most instructions. This can be seen in the # cycles column in Table 4.1. Most

instructions are single cycle except branch instructions, the LD/ST instructions and a few others.

Table 4.1 Instruction Set Summary

Mnemonic	Operation	Flags	# Clocks
ARITHMETIC AND LOGIC INSTRUCTIONS			
ADD	Add Two Registers	S,Z,C,N,V,H	1
ADC	Add with Carry Two Registers	S,Z,C,N,V,H	1
SUB	Subtract Two Registers	S,Z,C,N,V,H	1
SUBI	Subtract Constant from Register	S,Z,C,N,V,H	1
SBC	Subtract with Carry Two Registers	S,Z,C,N,V,H	1
SBCI	Subtract with Carry Constant from Register	S,Z,C,N,V,H	1
AND	Logical AND Registers	S,Z,N,V	1
ANDI	Logical AND Register and Constant	S,Z,N,V	1
OR	Logical OR Registers	S,Z,N,V	1
ORI	Logical OR Register and Constant	S,Z,N,V	1
EOR	Exclusive OR Registers	S,Z,N,V	1
COM	One's Complement Register	S,C,Z,N,V	1
NEG	Negate (2's Complement) Register	S,C,Z,N,V,H	1
SBR	Set Bit(s) in Register	S,Z,N,V	1
CBR	Clear Bit(s) in Register	S,Z,N,V	1
INC	Increment	S,Z,N,V	1
DEC	Decrement	S,Z,N,V	1
TST	Test for Zero or Minus	S,Z,N,V	1
CLR	Clear Register	S,Z,N,V	1
SER	Set Register	None	1
BRANCH INSTRUCTIONS			
RJMP	Relative Jump	None	3
RCALL	Relative Subroutine Call	None	3
RET	Subroutine Return	None	3
RETI	Interrupt Return	I	3
CPSE	Compare, Skip if Equal	None	1/2
CP	Compare (Rd - Rr)	S,C,Z,N,V,H	1
CPC	Compare with Carry (Rd - Rr - C)	S,C,Z,N,V,H	1
CPI	Compare Register with Immediate (Rd – K)	S,C,Z,N,V,H	1
SBRC	Skip if Bit in Register Cleared	None	1/2
SBRS	Skip if Bit in Register Set	None	1/2
SBIC	Skip if Bit in I/O Register Cleared	None	2/3
SBIS	Skip if Bit in I/O Register Set	None	2/3
BRBS	Branch if Status Flag Set	None	1/3
BRBC	Branch if Status Flag Cleared	None	1/3
BREQ	Branch if Equal (Z = 1)	None	1/3
BRNE	Branch if Not Equal (Z = 0)	None	1/3
BRCS	Branch if Carry Set (C = 1)	None	1/3
BRCC	Branch if Carry Cleared (C = 0)	None	1/3
BRSH	Branch if Same or Higher (C = 0)	None	1/3
BRLO	Branch if Lower (C = 1	None	1/3
BRMI	Branch if Minus (N = 1)	None	1/3

BRPL	Branch if Plus (N = 0)	None	1/3
BRGE	Branch if Greater of Equal, Signed (S = 1)	None	1/3
BRLT	Branch if Less than Zero, Signed (S = 0)	None	1/3
BRHS	Branch if Half Carry Set (H = 1)	None	1/3
BRHC	Branch if Half Carry Cleared (H = 0)	None	1/3
BRTS	Branch if T-Flag Set (T = 1)	None	1/3
BRTC	Branch if T-Flag Cleared (T = 0)	None	1/3
BRVS	Branch if Overflow Flag is Set (V = 1)	None	1/3
BRVC	Branch if Overflow Flag is Cleared (V = 0)	None	1/3
BRIE	Branch if Interrupt Enabled (I = 1)	None	1/3
BRID	Branch if Interrupt Disabled (I = 0)	None	1/3
DATA TRANSFER INSTRUCTIONS			
MOV	Move Between Registers	None	1
LDI	Load Immediate to Register	None	1
LD Z	Load Indirect with Z-Pointer	None	2
LD Z+	Load Indirect and Post-Increment with Z-Pointer	None	2
LD –Z	Load Indirect and Pre-Decrement with Z-Pointer	None	2
ST Z	Store Indirect with Z-Pointer	None	2
ST Z+	Store Indirect and Post-Increment with Z-Pointer	None	2
ST -Z	Store Indirect and Pre-Decrement with Z-Pointer	None	2
IN	In Port to Register	None	1
OUT	Out Register to Port	None	1
BIT AND BIT-TEST INSTRUCTIONS			
SBI	Set Bit in I/O Register	None	2
CBI	Clear Bit in I/O Register	None	2
LSL	Logical Shift Left	S,C,Z,N,V	1
LSR	Logical Shift Right	S,C,Z,N,V	1
ROL	Rotate Left through Carry	S,C,Z,N,V	1
ROR	Rotate Right through Carry	S,C,Z,N,V	1
ASR	Arithmetic Shift Right	S,C,Z,N,V	1
SWAP	Swap Nibbles	None	1
BSET	Flag Set	Any	1
BCLR	Flag Clear	Any	1
BST	Bit Store form Register to T	T	1
BLD	Bit Load from T to Register	None	1
SEC	Set Carry	C	1
CLC	Clear Carry	C	1
SEN	Set Negative Flag	N	1
CLN	Clear Negative Flag	N	1
SEZ	Set Zero Flag	Z	1
CLZ	Clear Zero Flag	Z	1
SEI	Global Interrupt Enable	I	1
CLI	Global Interrupt Disabl	I	1
SES	Set Signed Test Flag	S	1
CLS	Clear Signed Test Flag	S	1
SEV	Set Two’s Complement Overflow	V	1
CLV	Clear Two’s Complement Overflow	V	1
SET	Set T in SREG	T	1
CLT	Clear T in SREG	T	1
SHE	Set Half-carry Flag in SREG	H	1
CLH	Clear Half-carry Flag in SREG	H	1
NOP	No Operation	None	1
SLEEP	Sleep (Wait for Interrupt)	None	Any

Of course, some of the instructions will have different characteristics as the original AT90S1200 instructions. They are:

1. Unconditional branch instructions (RJMP, RCALL, RET, RETI) now take 3 cycles.
2. Conditional branch instructions take 1 cycle if the branch is not taken and 3 cycles if the branch is taken.
3. Skip if I/O register cleared/set instructions (SBIC, SBIS) take 2 cycles if the next instruction is not skipped and 3 cycles if the next instruction is skipped.
4. WDR (watch-dog reset) instruction is not available since the watch-dog timer features is not included in the designed
5. SLEEP will not enter any sleep modes (there are no sleep modes in the design), it will however stop the processor and wait for an interrupt. If an interrupt occurs, the processor will ‘wake up’, execute the interrupt routine and resumes execution from the instruction following SLEEP.
6. Data RAM is included in the design although AT90S1200 does not contain any data RAM. So 4 instructions are added, which are load and store instructions with post-increment and pre-decrement.
7. General purpose registers and I/O control registers are not mapped into the data addressing space for LD and ST instructions.
8. Only 16 registers are available for addressing - R16 to R32. This limitation is due to the area constraint of the targeted FPGA device.

Detail operation for each instruction requires further reference to the Instruction Set section in Atmel AVR RISC Microcontroller Data Book.

4.2 Addressing Modes

There are 7 addressing modes in the microcontroller. Rd and Rr are devoted to the destination register and soure register.

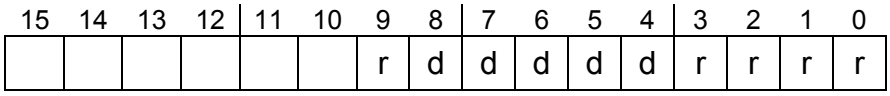
1. **Direct Single Register Addressing**
   The operand is in Rd.
2. **Direct Double Register Addressing**
   The operands are in Rd and Rr. Result is stored back to Rd.
3. **I/O Direct Addressing**
   First operand is one of the I/O registers. The address is contained in 6 bits of the instruction word. The second operand is either Rd or Rr. Used by IN and OUT instructions to read from or write to the I/O registers.
4. **Data Indirect Addressing**
   Operand address is the contents of the Z-register. Used when accessing the SRAM with LD and ST instructions.
5. **Data Indirect Addressing with Pre-Decrement**
   Z-pointer is decremented by 1 before the operation. Operand address is the decremented contents of the Z-register. Used when accessing the SRAM with LD and ST instructions.
6. **Data Indirect Addressing with Post-Increment**
   The Z-register is incremented by 1 after the operation. Operand address is the original content of the Z-register before increment. Used when accessing the SRAM with LD and ST instructions.
7. **Relative Program Memory Addressing**
   Program execution continue at address PC + offset. The offset is contains in the instruction word. Unconditional branch instructions (RJMP, RCALL) can reach the entire program memory from every location. However, conditional branch instructions can only reach –64 to 63 locations away from the current address.

Although there are 7 addressing modes of all, direct register addressing (mode 1 and 2) are used most of the time. Others mode are used when accessing the I/O, SRAM and when branching.

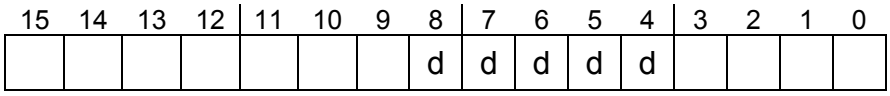
4.3 Instruction Formats

As mention earlier, RISC instructions have a fix length and are easily decoded. For this microcontroller, all instructions have a fixed-length of 16-bits. The instruction format is simple in order to be decoded easily.

For instructions that require two registers, *d* selects the destination register and *r* selects the source register. 5-bits can addressed a total of 32 registers ($N = 2^5 = 32$). Instructions of this format include ADD, SUB and AND.



For instructions that require one register, *d* addressed the destination/source register. Instructions of this format include NEG, ST and IN.



For immediate instructions, *K* is the 8-bit immediate value (constant) and *d* selects the destination register. Slightly different from the previous format, there are only 4 *d*-bits, which can address 16 registers only. The 5^{th} bit is assumed to be one in this case and address the upper 16 registers (R16 to R32). Instructions using this format are CPI, SUBI, SBCI, ORI, ANDI and LDI.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				K	K	K	K	d	d	d	d	K	K	K	K

For unconditional branch instructions, *k* is the offset in 2's complement. 12 bits wide offset provide a branch range from -2048 to 2047. Instructions using this format are RJMP and RCALL.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				k	k	k	k	k	k	k	k	k	k	k	k

For conditional branch instructions, *k* is the offset in 2's complement. The *s*-bits addressed which bit in the status register is to be tested for the branch. The 7-bit wide offset provide a branch range from –64 to 63. Instructions using this format are BRBC and BRBS.

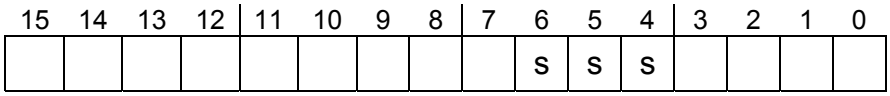
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
						k	k	k	k	k	k	k	s	s	s

An I/O addressing instructions will contain the I/O address (*A*-bits) plus the corresponding destination/source register (*d*-bits) or the corresponding bit in the I/O (*b*-bits). The first type has 6-bit wide *A*-bits, which provide 64 I/O addresses. The second type has 5-bit wide *A*-bits addressed only the lower 32 I/O. Instructions using the first format are IN and OUT while the second format are CBI, SBI, SBIC and SBIS.

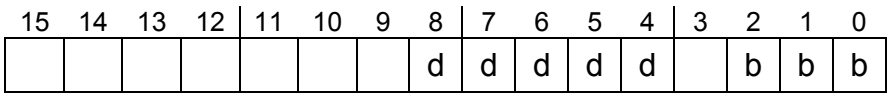
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					A	A	d	d	d	d	d	A	A	A	A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
								A	A	A	A	A	b	b	b

Clear/Set bit in status register instructions has the s-bits point to the corresponding bit. They are BCLR and BSET.



And finally instructions that use a single bit in the register have the b-bits point to the corresponding bit. They are BLD, BST, SBRC and SBRS.



4.4 Machine Codes

The instruction set contains 92 instructions and one might expect that there will be 92 different machine codes for all the instructions. But actually there are only 51 machine codes. This is because there are 41 equivalent instructions which share the same machine code with some others instructions. Table 4.2 shows these equivalent instructions.

Table 4.2 Equivalent Instructions

ADD	LSL
ADC	ROL
AND	TST
EOR	CLR
ORI	SBR
ANDI	CBR
LDI	SER
BCLR	CLI, CLT, CLH, CLS, CLV, CLN, CLZ, CLC
BSET	SEI, SET, SEH, SES, SEV, SEN, SEZ, SEC
BRBC	BRID, BRTC, BRHS, BRGE, BRVC, BRPL, BRNE, BRCC, BRSH
BRBS	BRIE, BRTS, BRHS, BRLT, BRVS, BRMI, BREQ, BRCS, BRLO

For an example, ADD is shares the same machine code as LSL. Left-shifting a number is actually multiplying the number by two, or adding a number to itself. So ADD can perform LSL operation by having the same register as destination and source. This is shown in Table 4.3.

Table 4.3 ADD And LSL Machine Codes

Instruction	Machine Code
ADD	0000 11rd dddd rrrr
LSL	0000 11dd dddd dddd

BCLR, BSET, BRBC and BRBS can choose one of the 8 status register flags as operand. The equivalent instructions in the right column state exactly which flag is to be used. Both are the same, except that instructions on the left are more meaningful and easier to used when writing a program. 8 status flags contribute to 8 specific instructions for both BCLR and BSET, and 9 specific instructions for both BRBC and BRBS because there are 2 instructions that test the C-flag.

Table 4.4 lists the machine codes for the 51 instructions. They are list according to the 4 most significant bits rather than the instruction format so that the instructions can be decoded easily when designing the control unit. The 41 equivalent instructions are place inside the bracket to the right of the correspondent instructions. It is very important to know that we only need to design 51 instructions. Once finish, the 41 equivalent instructions will immediately be available making the total to 92 instructions. The 51 instructions satisfy the RISC characteristic as having relatively few instructions.

Table 4.4 Machine Codes

```
NOP   0000 0000 0000 0000
CPC   0000 01rd dddd rrrr
SBC   0000 10rd dddd rrrr
ADD   0000 11rd dddd rrrr     (LSL)

CPSE  0001 00rd dddd rrrr
CP    0001 01rd dddd rrrr
SUB   0001 10rd dddd rrrr
ADC   0001 11rd dddd rrrr     (ROL)

AND   0010 00rd dddd rrrr     (TST)
EOR   0010 01rd dddd rrrr     (CLR)
OR    0010 10rd dddd rrrr
MOV   0010 11rd dddd rrrr

CPI   0011 KKKK dddd KKKK

SBCI  0100 KKKK dddd KKKK

SUBI  0101 KKKK dddd KKKK

ORI   0110 KKKK dddd KKKK     (SBR)

ANDI  0111 KKKK dddd KKKK     (CBR)

LD    1000 000d dddd 0000
ST    1000 001r rrrr 0000

LD Z+ 1001 000d dddd 0001
LD -Z 1001 000d dddd 0010
ST Z+ 1001 001r rrrr 0001
ST -Z 1001 001r rrrr 0010

COM   1001 010d dddd 0000
NEG   1001 010d dddd 0001
SWAP  1001 010d dddd 0010
INC   1001 010d dddd 0011
ASR   1001 010d dddd 0101
LSR   1001 010d dddd 0110
ROR   1001 010d dddd 0111
DEC   1001 010d dddd 1010

BSET  1001 0100 0sss 1000     (SE? - I,T,H,S,V,N,Z,C)
BCLR  1001 0100 1sss 1000     (CL? - I,T,H,S,V,N,Z,C)
```

```
RET   1001 0101 0000 1000
RETI  1001 0101 0001 1000

SLEEP 1001 0101 1000 1000

CBI   1001 1000 AAAA Abbb
SBIC  1001 1001 AAAA Abbb
SBI   1001 1010 AAAA Abbb
SBIS  1001 1011 AAAA Abbb

IN    1011 0AAd dddd AAAA
OUT   1011 1AAr rrrr AAAA

RJMP  1100 kkkk kkkk kkkk

RCALL 1101 kkkk kkkk kkkk

LDI   1110 KKKK dddd KKKK     (SER)

BRBS  1111 00kk kkkk ksss     (BR?? - CS,LO,EQ,MI,VS,LT,HS,TS,IE)
BRBC  1111 01kk kkkk ksss     (BR?? - CC,SH,NE,PL,VC,GE,HC,TC,ID)

BLD   1111 100d dddd 0bbb
BST   1111 101r rrrr 0bbb

SBRC  1111 110r rrrr 0bbb
SBRS  1111 111r rrrr 0bbb
```

CHAPTER V

PIPELINE PROCESSING

5.1 Instruction Cycle

Figure 5.1 shows the the instruction cycle – which is divided into two stages, the fetch stage and the execute stage. In the fetch stage, the machine code of an instruction is fetched into the instruction register. The control unit decodes the instruction to know what the instruction performs and what operands are needed. In the execute stage, the operands are fetched and the instruction is executed. The results is written back at the end of the stage.

FETCH		EXECUTE		
Instruction Fetch	Decode	Operand Fetch	Operate	Write

Figure 5.1 Instruction Cycle

As an example, the instruction ADD R18,R20 will add R18 and R20 and write the result back to R18. At the fetch stage (1st clock transition), the machine code for ADD R18,R20 is fetched into the instruction register. After the instruction is fetched, the CPU now know that the next instuction to be executed is ADD and the operands is R18 and R20. At the execute stage (the 2nd clock transition), the contents of R18 and R20 is latched into operand register A (ORA) and operand register B (ORB) which are connected directly to the ALU. The ALU perform the ADD operation between ORA and

ORB and the result is sent to the data bus. At the end of the execute stage (the 3rd clock transition), the result in the data bus is written into the destination register. The instruction cycle is then complete.

5.2 Instruction Pipeline

If an instruction cycle has 2 stages (Fetch and Execute), executing a series of instructions will have the form of:

Fetch1 → Execute1 → Fetch2 → Execute2 → Fetch3 → Execute3 → …

The first instruction is fetched and executed, then the second instruction is fetched and executed, and so forth. Executing an instruction takes 2 cycles and executing 10 000 instructions will take 20 000 cycles. By using instruction pipelining, the performance of the system can be further enhanced.

Shown in Figure 5.2 is the instruction pipeline structure. The fetch and execute stage are now overlapped to perform simultaneous operations. The next instruction is fetched while executing the current instruction. This is called instruction pre-fetch.

Clock Transition	T1	T2	T3	T4
1st instruction	Fetch	Execute		
2nd instruction		Fetch	Execute	
3rd instruction			Fetch	Execute

Figure 5.2 Instruction Pipeline Structure

Imagine after using the instruction pipeline, there will be one instruction executing at every cycle. Executing an instruction will take only one cycle and executing 10 000 instructions take only 10 000 cycle. The performance is now doubled.

Program counter (PC) addressed the instruction in the program and are tightly related to the pipeline structure. Figure 5.3 shows the PC change along the pipeline execution. After reset, the PC is cleared to 0. On the 1st clock transition (T1) after reset, instruction at address 0 is being fetched. At the same time, PC is incremented to 1. On the 2nd clock transition (T2), instruction 0 is executed and instruction 1 is fetched. PC is now incremented to 2. On the 3rd clock transition (T3), result of instruction 0 is written back, instruction 1 is executed and instruction 2 is being fetched. PC is now incremented to 3. The important point is, when instruction N is being executed, instruction N + 1 is being fetch and the PC is N + 2.

Clock Transition	Reset	T1	T2	T3	T4
PC	0	1	2	3	4
Instruction 0		Fetch 0	Execute 0		
Instruction 1			Fetch 1	Execute 1	
Instruction 2				Fetch 2	Execute 2

Figure 5.3 PC and Instruction Pipeline

5.3 Pipeline Conflicts

Executing a branch instruction will cause a pipeline conflict. In that case the pipeline must be flushed and all instructions that have been read from the memory after the branch instructions must be discarded.

Figure5.4 shows how a branch instruction will affect the pipeline. Instruction 20 is ‘Branch to 73’. At T2, the branch instruction is executed but at the same time instruction 21 is fetched as usual. On T3, the new value for PC is loaded and instruction

22 is fetched. Instruction 21 cannot be executed and is flushed from the pipeline. Only at T4, instruction 73 is being fetched. Instruction 22 must also be discarded. Finally at T5, instruction 73 is executed. So a branch instruction will take 3 cycles to complete. The first cycle is taken to load the PC with the new value. The following 2 cycles are just wait states to wait until the new instruction is executed.

Clock Transition	T1	T2	T3	T4	T5
PC	21	22	73	74	75
20 (Branch to 73)	Fetch 20	Execute 20	Execute 20	Execute 20	
Instruction 21		Fetch 21	Flushed		
Instruction 22			Fetch 22	Flushed	
Instruction 23				Fetch 73	Execute 73

Figure 5.4 Branch Instruction Pipeline

Some instructions like LD and ST require 2 execution cycles. This will also affect the pipeline flow. Figure 5.5 shows the pipeline structure when a 2 cycles instruction is encountered. Instruction 31 is a 2 cycles instruction. At T3, it is executed and instruction 32 is fetched. At T4, instruction 31 continues its execution. PC is not incremented and no new instruction is fetched. Only at T5, instruction 32 is executed. The next instruction is fetched and the PC is incremented. In this case, the pipeline is hold for one cycle.

Clock Transition	T1	T2	T3	T4	T5
PC	31	32	33	33	34
Instruction 30	Fetch 30	Execute 30			
31 (2 cycles)		Fetch 31	Execute 31	Execute 31	
Instruction 32			Fetch 32		Execute 32
Instruction 33					Fetch 33

Figure 5.5 2 Cycles Instruction Pipeline

Data dependency conflict arises when an instruction depends on the result of a previous instruction, but the result is not yet available. Lets examine the following instructions flow:

```
LDI   R18,$10     ; R18 = $10
LDI   R18,$20     ; R18 = $20
INC   R18         ; R18 = R18 + 1 = $21
```

The final value of R18 should be $21. But unluckily the real result is $11. In an instruction fetch cycle, operands are fetched at the start of the execute stage to the operands register and result is written back at the end of the execute stage to the destination register. As a result, operands of the next instruction are loaded into the operand register at the same rising edge as the write back of the current instruction to the destination register. So INC R18 is actually receiving the old value of R18, which is $10 rather then the result of LDI R18,$20.

To solve this problem, a technique called operand forwarding is used. If the destination register is needed as a source in the next instruction, the ALU result is forwarded to the operand register directly. This will require extra control logic to check for the conflict and perform the forwarding job.

CHAPTER VI

MICROCONTROLLER ORGANIZATION

6.1 Pin Description

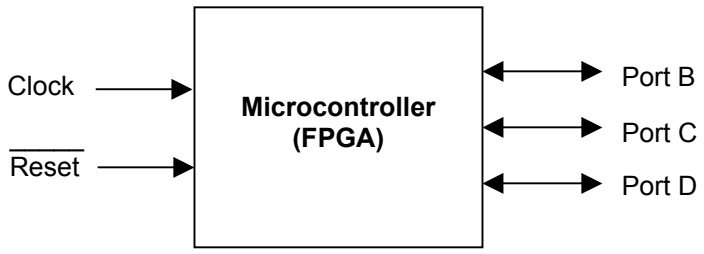


Figure 6.1 Microcontroller Pin Configuration

Figure 6.1 shows the pin configuration for the designed microcontroller. The microcontroller has 2 input pins and 3 bi-directional I/O ports. Each I/O port consists of 8 individual I/O pins. So 3 I/O ports contribute to a total of 24 I/O pins. The clock signal will drive the whole microcontroller directly. Reset is active low; when asserted it resets the microcontroller to the default state even if the clock is not running. Port B, Port C and Port D are all 8-bits port. Each bit can be configure to be input or output. All port pins are tri-stated when the microcontroller is reset. Pin D7 also serves as the external interrupt source and external timer clock source.

6.2 Architecture Overview

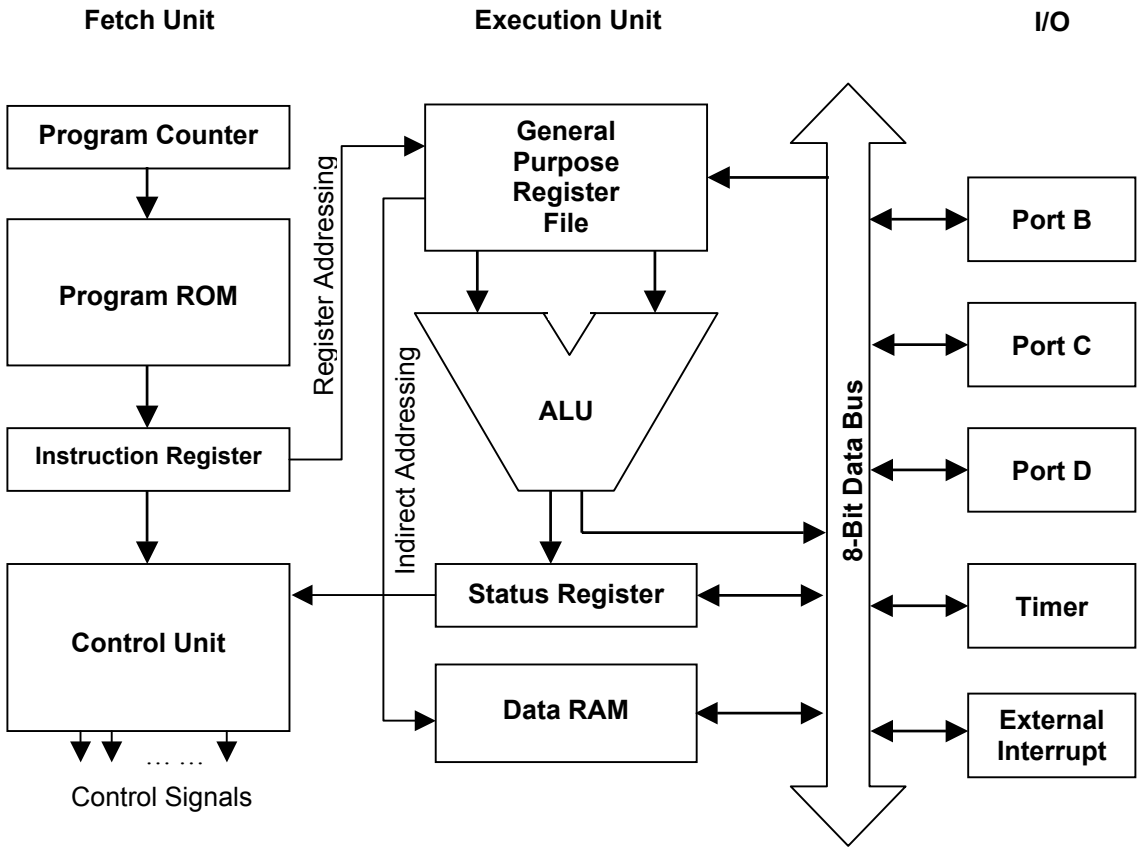


Figure 6.2 Top-level Block Diagram

Figure 6.2 shows the top-level block diagram of the design, the bus structure has been simplified, but every block represents a module to be designed. At first glace, there are 11 modules in the top-level, with the 3 ports sharing the same module. These 11 modules are to be design separately using the top down design approach. Some modules like the instruction register and status register are easy to design**,** but modules like ALU and the control unit require a lot of understanding. The overall dataflow and bus structure between all the modules must be understand before designing the modules individually.

Buses provide connection between modules. There are basically two kinds of buses, direct bus and common bus. Direct bus connects two modules directly and is used specifically by the connected modules. There are many direct buses, such as the connection between program counter and program ROM, between program ROM and IR, between register file and ALU, etc. No control signals are required for direct buses.

A common bus is a bus shared by many modules. The data bus is the only common bus in this design. The data bus provides connection between the general purpose register file, ALU, status register, SRAM and all the I/O features. The register file can only receive data from the data bus. All others modules can receive and send data to the data bus. Since there are so many possible data flows, control signals are required to control the correct flow direction. Only one source to the data bus is allowed at a time. If not, logic contentions will happen and the value of the data bus will be invalid. Tri-state bus is used to implement the common data bus. Only the correct source is connected to the data bus while other are in high impedance state. The impedance is so high that it can be seen as unconnected to the bus system. If the ALU is the data source, the data bus will be flooded with the result of the ALU and is available to all the connected modules. Control logic will generate an enable signal for the real destination to receive the data.

Next is a brief introduction to the whole system. The system can be divided into 3 units, the fetch unit, execute unit and I/O unit. Fetch unit is in charge of fetching the next instruction and the execute unit is in charge of executing the current instruction. I/O unit provide a connection with the outside world. The fetch unit and execute unit form the CPU of the microcontroller.

The first module of the fetch unit is the program counter (PC). The PC contains the address of the next instruction to be executed. It points to the program ROM to locate the instruction. The instruction from the ROM is then latched into the instruction register (IR). The control unit takes the content of the IR and decodes it. It then assert

the appropriate control signals to execute the instruction. All modules are connected with direct buses.

The execute unit in charge of executing most instructions. Normally, to execute an instruction, 2 operands are output from the register file to the ALU. The ALU then perform the operation and send the result to the data bus. Contents of the data bus (the result) is then stored back to the register file. The ALU also evaluate the status register flags and send them directly to the status register (SR). The whole execution process is done in a single cycle. The ALU perform many operations - include passing the contents of a general register to the data bus. SR also has a direct bus connection to the control unit required for branch evaluation. The register file (destination and source register) is addressed directly by some bits in IR.

A RISC has memory access limited to only LD and ST instructions. Direct addressing to the data RAM is not available. Only indirect addressing through the Z-pointer (R30) is allowed. It could be indirect addressing, indirect addressing with post-increment and indirect addressing with pre-decrement. Load and store instructions can only transferred data between the RAM and the register file. The Z-pointer contains the address of the RAM. A load operation sends the RAM data to the general registers through the data bus. A store operation sends the data to ALU, the ALU pass the data to data bus and store into the RAM.

To implement the fetch and execute pipeline in this microcontroller, memory are implemented using the Harvard architecture. Program and data are store in separate memories. As seen in the block diagram, program is stored in the program ROM while data are stored in the data RAM. The advantage of Harvard architecture is the ability to fetch the pre-fetch the next instruction easily. A normal RAM will have initial value zero when powered on. In FPGA, the RAM can have initial values and thus can make it act as a ROM.

All the I/O modules contain many control registers. Data are sent to and received from it through the common data bus. Table 6.1 shows the complete list of the I/O control registers and their corresponding address. Reserved and unused locations are not shown in the table. The SR is also mapped into one of the I/O address. IN and OUT instructions are used to transfer data between these control registers and the general registers. The lower half of the control registers ($00 - $1F, shaded in gray) are directly bit-accessible using the SBI and CBI (Set/Clear Bit in I/O) instructions. Using SBIS and SBIC (Skip if bit in I/O cleared/set) instructions can also check every single bit in these registers. In this design, the lower half of control registers are all the I/O ports control registers. Note that, PINB, PINC and PIND are not a real registers, only a read operation can apply to them and it will read the physical value holding by the external pins.

Table 6.1 I/O Address Space

Address Hex	Name	Function
$3F	SREG	Status REGister
$3B	GIMSK	General Interrupt MaSK register
$39	TIMSK	Timer/Counter Interrupt MaSK register
$38	TIFR	Timer/Counter Interrupt Flag Register
$35	MCUCR	MCU general Control Register
$33	TCCR0	Timer/Counter 0 Control Register
$32	TCNT0	Timer/Counter 0 (8-bit)
$18	PORTB	Data Register, Port B
$17	DDRB	Data Direction Register, Port B
$16	PINB	Input Pins, Port B
$15	PORTC	Data Register, Port C
$14	DDRC	Data Direction Register, Port C
$13	PINC	Input Pins, Port C
$12	PORTD	Data Register, Port D
$11	DDRD	Data Direction Register, Port D
$10	PIND	Input Pins, Port D

6.3 Register Transfer

The whole design contains many registers - instruction register (IR), instruction backup register (IBR), program counter (PC), general purpose registers, memory address

register (MAR), all the I/O control registers and many more (The program counter is treated as a special kind of register). They are found inside most of the modules see in the top-level block diagram. The whole system works by transferring data between these registers (register transfer). Some data are transferred without modification while some are manipulated before transfer to the next register. If the data are to be manipulated, they are manipulated by the combinational logic between these registers. How these data are transferred, how are they being manipulated before transfer, and what does different data inside the register means, will determine whether the design can work as a microcontroller. The design will perform a long series of register transfer to form the functioning of a microcontroller. Figure 6.3 shows the register transfer concept. It can be seen in the figure that register are transferred to another through many levels of combinational logic.

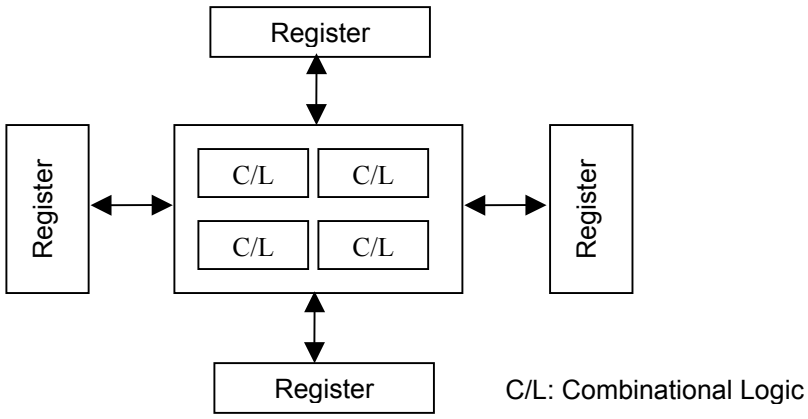


Figure 6.3 Register Transfer

A read of the status register will bring the contents of the status register to one of the general register directly without manipulation. However, the value of the Z-pointer is send to the memory address register (MAR) after a subtraction by 60. The combinational logic in this case is a subtracter. Performing an AND operation between two general registers, will pass the two registers through a combinational logic (the logic unit) before writing back to one of the register. Memory (program ROM and data RAM) are treated as a kind of combinational logic. Program counter (PC) are pass through the program

ROM to the instruction register. The instruction register will receive the instruction in from the program ROM pointed by the PC.

So, the design process is to design all the registers along with the combinational logic and the interconnection between them. This is called the datapath of the system. Control signals are then used to determine how the register transfer takes place. Control signals are asserted by the control unit. The datapath along with the control unit form the complete microcontroller. It is important to know what registers exists in the system. Table 6.2 lists all the modules and their respected registers.

Table 6.2 Registers List

Modules	Registers
Program Counter	Program Counter (PC) Program Counter Backup (PCB) 4 Hardware Stack (STACK0 – STACK3)
Instruction Register	Instruction Register (IR)
Control Unit	Instruction Backup Register (IBR)
General Purpose Register File	16 General Purpose Registers (R16 - R31) Z-Pointer (R30)
ALU	Operand Register A (ORA) Operand Register B (ORB)
Status Register	Status Register (SR)
Data RAM	Memory Address Register (MAR) Memory Buffer Register (MBR)
Port B	Data Register (PORTB) Data Direction Register, Port B (DDRB)
Port C	Data Register (PORTC) Data Direction Register, Port C (DDRC)
Port D	Data Register (PORTD) Data Direction Register, Port D (DDRD)
Timer	Timer/Counter Interrupt Mask Register (TIMSK) Timer/Counter Interrupt Flag Register (TIFR) Timer/Counter 0 Control Register (TCCR0) Timer/Counter 0 (TCNT0)
External Interrupt	General Interrupt Mask Register (GIMSK) MCU General Control Register (MCUCR)

CLK is the global clock signal for all the registers while CLRN is the global reset signal (active low). CLRN clears all the registers when asserted low.

6.4 Control Signals Characteristics

If the control signals are used to control the datapath then the characteristics of the control signals must be understood before we can proceed further. First, a control signal will have at least a length of one clock cycle. It usually asserted a short delay after a rising clock transition and unasserted a short delay after another rising clock. The datapath consists of many registers and combinational logic between them, so there are basically 2 kinds of control signals. The first kind controls the combinational logic and the second kind controls the registers.

When a combinational logic encounters a control signal, it will act towards the signal immediately. The ADD signal will cause the adder to perform the add operation immediately. The delay to get the valid result is the delay for the input to propagate through the combinational logic. The combinational logic can be functional unit such as adder and shifter, steering logic such as multiplexers and decoders or memory (program ROM and data RAM).

A register control signal requires a rising clock to operate. WR_REG signal will only latch the data into the destination register of the register file when it encounters the rising clock. Since control signals are asserted a short delay after a rising clock and unasserted on the next, the operations is actually happened at the end of the control signal where it meet the rising clock. These kinds of control signals are the enable signals for the registers, or the increment/decrement signal for a counter.

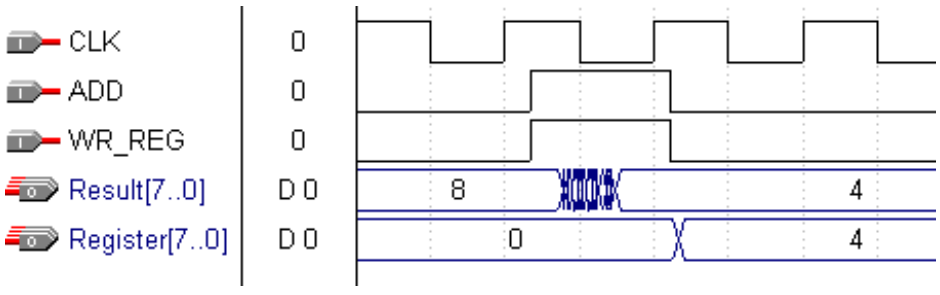


Figure 6.4 Control Signal Timing

Figure 6.4 explains the concept graphically. Both ADD and WR_REG control signals are asserted and unasserted after a rising clock. The ADD signal gives effect immediately after it is asserted by asking the adder to perform an add operation. The result is available after some delay depending of the speed of the adder. The WR_REG signal latches the result into the register at the end of the signal when it encounters the rising clock. Notice that the register changes value a short delay after the rising clock.

CHAPTER VII

DATAPATH DESIGN

7.1 Chapter Overview

The design of the microcontoller is discussed in 2 separate chapters – one for the datapath and one for the control unit. This chapter discusses the design of the datapath while the next chapter will discuss about the control unit. All modules in the top-level block diagram (Figure 6.2) except the control unit are part of the datapath. These modules are listed in Table 7.1. The design of each module will be discussed one by one in this chapter.

Table 7.1 Modules Inside The Datapath

1.	Program counter (PC)
2.	Instruction Register (IR)
3.	Program ROM
4.	General Purpose Register File
5.	ALU
6.	Status Register (SR)
7.	Data RAM
8.	Port
9.	Timer
10.	External Interrupt

7.2 Program Counter (PC)

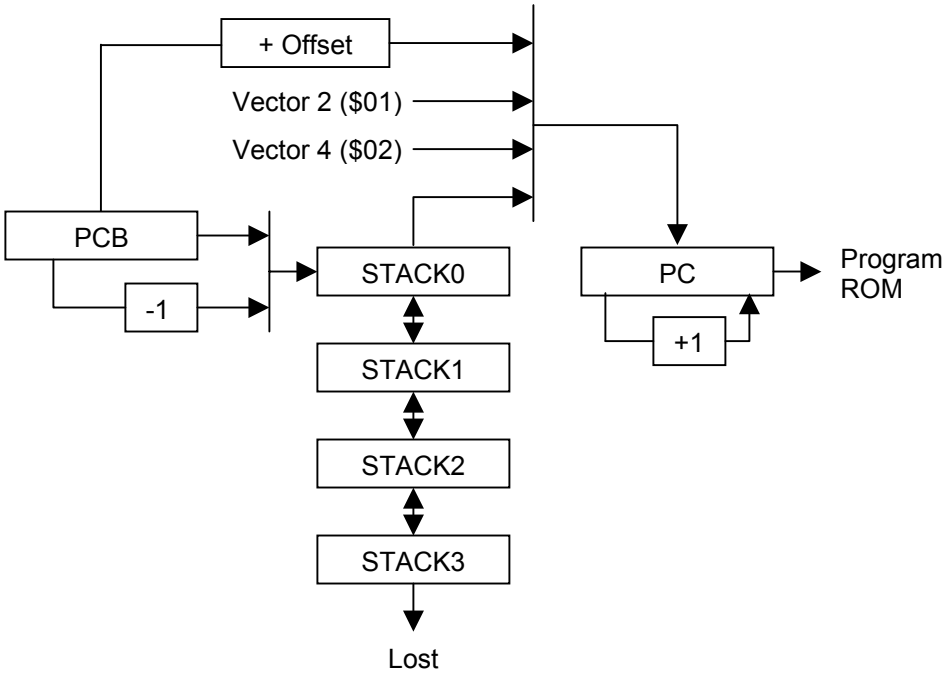


Figure 7.1 Program Counter Architecture

Figure 7.1 shows the architecture of the PC module. In the most basic execution stream, the PC is incremented on every clock transition. But in some cases, the PC will be loaded with a new value instead of incrementing it. Hardware stack is used to keep the return address of a subroutine call or interrupt request. Program counter backup register (PCB) always loaded with the last PC value.

There are 3 circumstances that the PC will be loaded with a new value instead of incrementing it. The first is serving a branch instruction (conditional or unconditional); the second is serving an interrupt request; and the third is returning from a subroutine or interrupt service routine (ISR).

The description for branch instructions is PC ← PC + 1 + offset. This can be a confusing description. Should the PC stands for the value in the real program counter itself; or the address of the current executing instruction; or the address of the next

instruction to be fetched? Recall the pipelining discuss in chapter 5, when the CPU is executing the N^{th} instruction, the PC has already increased to N + 2 and the instruction in the IR is the N + 1 instruction. The PC in the description is actually the address of the branch instruction itself, not the real hardware PC. So PC + 1 points to the next instruction that follow. If the real PC is always ahead of 2, another register - program counter backup register (PCB) is used to keep the last PC value that is ahead of 1. When serving a branch instruction, the new PC value will be the PCB plus the offset

Serving an interrupt request will cause the PC to be loaded with the interrupt vector address. There are 2 interrupt vectors and 1 reset vector all located at the start of the program memory space listed in Table 7.2 according to its priorities. An RJMP instruction that jumps to the interrupt service routine (ISR) is contained in the vector address. So when serving an interrupt request, the PC is first loaded with the vector address, then the CPU execute the instruction loaded from the corresponding vector address - a jump to ISR. The PC is then loaded with the address of the ISR. And finally the CPU starts executing the ISR.

Table 7.2 Interrupt Vector

Vector No.	Vector Address	Source	Interrupt Definition
1	$000	RESET	Reset Pin
2	$001	INT0	External Interrupt Request 0
4	$002	TIMER0, OVF0	Timer/Counter Overflow

The hardware stack is used to store the return address of a subroutine call and interrupt request. The stack is 4-level deep (STACK0 – STACK 3) and is LIFO (Last In First Out). When the CPU serves a subroutine call or interrupt request, the return address is pushed into STACK0. The original contents in the stack are push one level deeper, with STACK0 pushed into STACK1, STACK1 into STACK2, and so forth. This is called the push operation. On returning from a subroutine call or interrupt request, PC is being loaded with STACK0 and the original contents of the stack is pull up one level, from STACK3 into STACK2, from STACK2 into STACK1, and so forth. This is called

the pull operation. If there are more that 4 subsequent subroutine call or interrupt request, the first return address that is pushed into the stack will be lost.

The return address for a subroutine call is the PCB. However, the return address for an interrupt request is not PCB but PCB – 1. A short program in Figure 7.2 will help to clarify this. When executing the RCALL MAKE instruction, PCB is $22 and is pushed into the stack. The PC is loaded with $50 and the CPU start executing the MAKE subroutine. When it encountered the RET instruction, the PC is pulled from the stack, which contains $22, the next instruction address following RCALL MAKE.

```
Addr  Label Instruction

$00         rjmp  reset       ; reset vector
$01         rjmp  extirq      ; external IRQ vector
$02         rjmp  timer       ; timer overflow IRQ vector

$20         add   r25,r26     ; r26 = r26 + r25
$21         rcall make        ; call subroutine make
$22         dec   r26         ; r26 = r26 - 1
$23         inc   r25         ; r25 = r25 + 1

$50   make: ...
            ...
            ret               ; return from subroutine

$80   timer:...
            ...
            reti              ; return from interrupt
```

Figure 7.2 Subroutine Call and Interrupt Request Program

Now lets assume that the CPU serves the timer overflow interrupt request at $21. PCB is $22. PC is loaded with the vector address ($02) and the CPU executes RJMP TIMER and the ISR. When it encounters the RETI instruction, the return address is pop from the stack to the PC. If PCB was pushed into the stack earlier, the next instruction that follows will be DEC R26, which is wrong because RCALL MAKE has not been executed yet! So the correct return address for an interrupt request is PCB – 1.

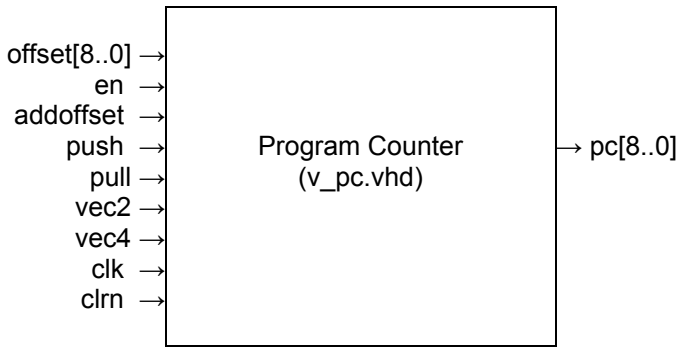


Figure 7.3 Program Counter Symbol

Figure 7.3 shows the symbol of the program counter (PC) module. PC, PCB and the 4 hardware stacks are 9-bits wide. So the PC can address up to 512 unique locations. The PC is connected to the program ROM directly. The offset for a relative branch is received from the control unit, so as others control signals. EN is the enable signal for all the registers – PC, PCB and stacks. Only when EN signal is asserted, operation can be performed. This signal is asserted only when executing a 2 cycles instruction to hold the pipeline.

ADDOFFSET signal load the PC with PCB + offset. VEC2 signal loads the PC with interrupt vector 2 ($01 - external interrupt) while VEC4 signal loads the PC with interrupt vector 4 ($02 - timer overflow interrupt). PUSH signal performs the push operation and PULL signal performs the pull operation. If none of the signals that load the PC is asserted, the PC will be incremented by one.

7.3 Program ROM

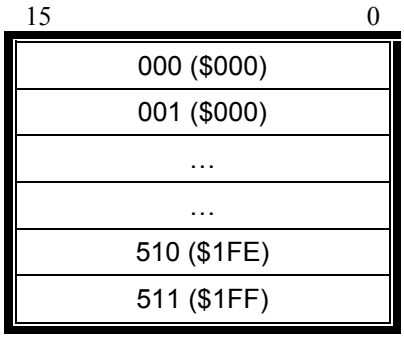


Figure 7.4 Program ROM Organization

The program ROM is used to store the program for the microcontroller. A program is a combination of many instructions to perform a specific task. Figure 7.4 shows the organization of the program ROM. Since all instructions have a fixed width of 16-bits, the ROM word size is also 16-bits so that the instruction can be fetched into the instruction register in a single cycle. The ROM size is 1 K bytes, or better stated as 512 words. This means that it can store up to 512 instructions. To address 512 locations, it requires 9-bit wide address.

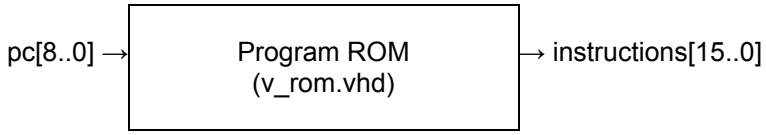


Figure 7.5 Program ROM Symbol

Figure 7.5 shows the symbol of the program ROM. It is implemented using the LPM_ROM module provided by Altera, which is the recommended way to implement memory in Altera FLEX10K devices. Program counter provide the 9-bit address through a direct bus to the ROM. The instruction output from the ROM is then send to the instruction register. No clock signal is require for the program ROM. It can be imagined as a combination logic where the output will be available some delay after the input has changed.

7.4 Instruction Register (IR)

As its name suggest, instruction register (IR) is used to store the instruction. The instruction is received from the program ROM through a direct 16-bit wide bus connection. The IR will only latch the new instruction in if the EN signal is asserted. The IR (the instruction) is connected to the control unit for decoding. The corresponding bits that form the immediate value are sent to the ALU. While the bits that addressed the destination and source register are sent to the general purpose register file. Please refer to chapter 4 for instruction format. Figure 7.6 shows the symbol for the IR module.

It is important to note that the instruction in IR is not holding the current executing instructions. IR is always holding the next instruction. So the control unit is always decoding the next instruction. Recall that the execute stage of the current instruction is also the fetch stage of the next instruction in the pipeline organization.

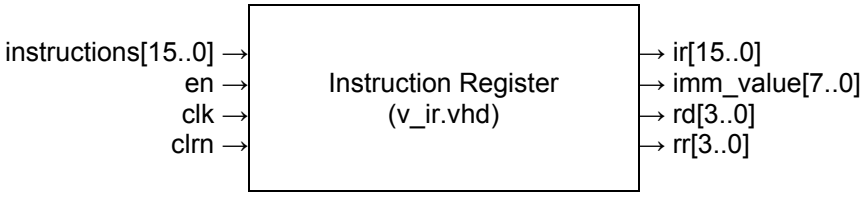


Figure 7.6 Instruction Register Symbol

Figure 7.6 show the symbol if the instruction register module. The EN signal seen here is same as the EN signal seen in the PC module. The IR can only load the data from program ROM when EN is asserted. Usually, it is always asserted except when the CPU is executing a 2 cycles instruction.

7.5 General Purpose Register File

A RISC CPU usually have a large general purpose register file. The standard number of registers are normally 32, so as in the AT90S1200. 32 registers will require about 52% area of the targeted device Altera EPF10K20RC240-4, which are unacceptable. As a result, only 16 registers are included in this design. The same instruction format discussed earlier is used, except that the 5th bit of the register address is now a don't care value.

Figure 7.7 shows the structure of the 16 general purpose registers. They are numbered from R16 to R31 instead of R0 to R15 due to 2 reason. Firstly, immediate instructions like LDI can only address the upper register file as discussed in the instruction format section in chaper 4. Secondly, the indirect Z-pointer share the same register as R30.

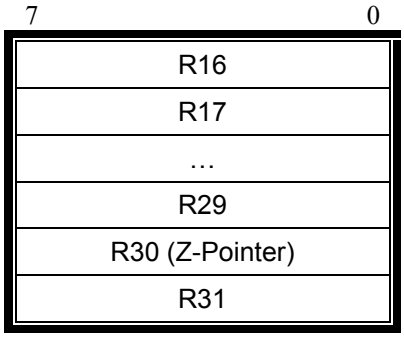


Figure 7.7 General Purpose Register File Organization

At any time, the register file will connect 2 registers to the ALU through two 16-to-1 multiplexers. The two registers are the destination register and source register, addressed directly by the instruction register. The data bus is connected directly to the register file. The value of the data bus can be written to the destination register if the WR_REG signal is asserted.

The address bus connects the register file and data RAM together. R30 can be used as either a general register or the Z-pointer (ZP) to address the data RAM. The

starting address for data RAM is $60. Unfortunately, the targeted device requires the starting address of a RAM to be $0. So, the address bus value is the value of ZP subtracted by $60. If indirect addressing with pre-decrement is used, then the address bus is the value of ZP subtracted by 61.

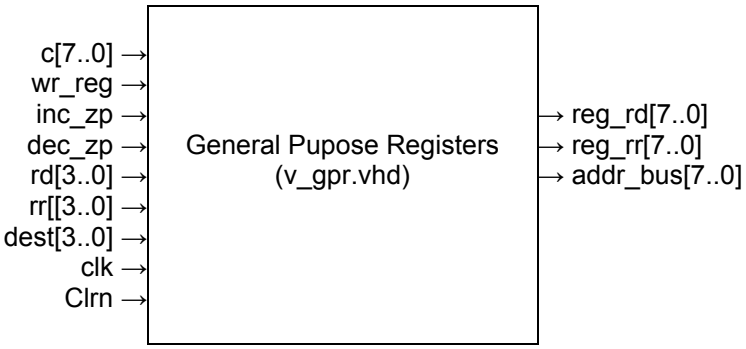


Figure 7.8 General Purpose Registers Symbol

Figure 7.8 shows the module symbol of the general purpose register file. INC_ZP is asserted when indirect addressing with post-increment is used. It will incrememt ZP by 1. DEC_ZP is asserted when indirect addressing with pre-decrement is used. It will decrement ZP by 1. When indirect addressing with pre-decrement is used, the MAR load the value of the address bus at the same rising edge the ZP is decremented. So MAR will not be able to load the decremented ZP value. This is the reason why the address bus is the value of ZP subtracted by $61 instead of $60 to correct this problem.

Recall that an instruction cycle are divided into fetch stage and execute stage. Operands are fetch at the start of the execute stage while the result is written back at the end of the execute stage. To be able to fetch the correct operands at the start of the execute stange, the operands must be known in the fetch stage. The instruction in the IR is in the fetch stage, so it addressed the operands. When the instruction enter the execute stage, another instruction is fetched. To write the result from the ALU back to the correct destination register must now tell by the control unit instead of the IR because IR only knows the destination register of the next instructions. This is the difference between RD (fetch stage) and DEST (execute stage).

7.6 ALU

The ALU executes many instructions, some directly and some indirectly. We first examine the 24 most basic instructions that are executed directly by the ALU. These instructions are listed in Table 7.3. They are divided into 5 groups – ADD, SUBCP, LOGIC, RIGHT and DIR. ADD group instructions perform add operations; SUBCP group instructions perform subtract and compare operations; LOGIC group perform logical operations; RIGHT group perform right shifting; DIR group perform direct wiring operations.

Table 7.3 Basic Instructions

Group	Instruction	Extra Signal	Wr_Reg	ORA	ORB	Flags
ADD	ADD		✓			HSVNZC
	ADC	WCARRY	✓			HSVNZC
	INC		✓		One	SVNZ
SUBCP	SUB		✓			HSVNZC
	SUBI		✓		Imm	HSVNZC
	SBC	WCARRY	✓			HSVNZC
	SBCI	WCARRY	✓		Imm	HSVNZC
	CP					HSVNZC
	CPC	WCARRY				HSVNZC
	CPI				Imm	HSVNZC
	DEC		✓		One	SVNZ
	NEG		✓	Zero	Rd	HSVNZC
LOGIC	AND	LOGICSEL = 00	✓			SVNZ
	ANDI	LOGICSEL = 00	✓		Imm	SVNZ
	OR	LOGICSEL = 01	✓			SVNZ
	ORI	LOGICSEL = 01	✓		Imm	SVNZ
	EOR	LOGICSEL = 10	✓			SVNZ
	COM	LOGICSEL = 11	✓			SVNZC
RIGHT	LSR	RIGHTSEL = 00	✓			SVNZC
	ROR	RIGHTSEL = 01	✓			SVNZC
	ASR	RIGHTSEL = 10	✓			SVNZC
DIR	MOV	DIRSEL = 0	✓			
	LDI	DIRSEL = 0	✓		Imm	
	SWAP	DIRSEL = 1	✓			

* If not stated, then ORA is default to Rd and ORB is default to Rr.

WR_REG signal is asserted if the result of the ALU will be written back to the destination register. It is a register file control signal, not the ALU. ORA and ORB columns shows what should be loaded into the operand register A and operand register B. If the cell is blank, it is default to Rd (destination register) for ORA and Rr (source register) for ORB. Zero is "0000 0000"; One is "0000 0001"; and Imm is the immediate value of the instructions.

Every instructions has its own combination of group + extra signal + WR_REG signal + ORA + ORB. This combination makes the 24 instructions unique to each other. To represent a group, a control signal with the same name as the group is asserted. ADC is executed by fetching Rd to ORA, Rr to ORB and assert the ADD signal, WCARRY signal and WR_REG signal. ORI is executed by fetching Rd to ORA, the immediate value to ORA, assert the LOGIC signal and set LOGICSEL to 01. All others instructions are executed according to their combination

A total of 21 instructions will change the status register (SR) flags based on the result of the operation. So the ALU need to evaluate the flags and send them to the SR. If an instruction will modified the C-flag, the control will enable the C-bit in SR in order to received the new flag from the ALU. The flag column in Table 7.3 shows the flags that are affected by the 21 instructions. If a flag is not affected, the control unit will not enable the corresponding bit in SR. The value ALU send to the SR is don't care.

Table 7.4 Bit Load instructions

Signal	Instruction	Wr_Reg	ORA	Description
BLD	BLD	✓	Rd	Load T-Flag to bit
CBISBI	CBI		I/O	Clear bit in I/O register
	SBI		I/O	Set bit in I/O register

We now add 3 more instructions to our discussion, listed in Table 7.4. They require a single operand, Rd for BLD and an I/O register for CBI and SBI. Their operation is similar where BLD loads the T-flag into a bit while CBI can be think as loading a 0 into a bit, so as SBI is loading a 1 to a bit. The control will tell which bit

should be loaded. SBI and CBI instruction require 2 cycles to complete. At the first cycle, the I/O register is fetched to ORA through the data bus. At the second cycle, a 0 or 1 is loaded to the bit location and result is written back to the I/O through the data bus.

The register file can only received data from the data bus. So, in order to send data out to the data bus, it needs to pass it through the ALU. Table 7.5 list the 4 instructions of this group. OUT instruction transfers the content of a register to an I/O register while LD Z, LD Z+ and LD –Z transfer the content of a register to the data RAM. Rd is fetch to ORA and the ALU perform a pass operation to pass ORA to data bus.

Table 7.5 Pass ORA Instructions

Signal	Instruction	ORA	Description
PASSA	OUT	Rd	Skip if bit in register cleared
	LD Z	Rd	Skip if bit in register set
	LD Z+	Rd	Skip if bit in I/O register cleared
	LD –Z	Rd	Skip if bit in I/O register set

Following next are the 5 skip instructions listed in Table 7.6. They determine whether the next instruction followed should be skipped. The SKIPTEST group instructions require single operand while the CPSE instruction requires two. I/O refers to the respective I/O register. SBIC and SBIC instruction require 2 cycles to operate just like CBI and SBI. At the first cycle, the I/O register is fetched to ORA. The skip test then performed at the second cycle.

Table 7.6 Skip Instructions

Signal	Instruction	ORA	ORB	Description
SKIPTEST	SBRC	Rd	X	Skip if bit in register cleared
	SBRS	Rd	X	Skip if bit in register set
	SBIC	I/O	X	Skip if bit in I/O register cleared
	SBIS	I/O	X	Skip if bit in I/O register set
CPSE	CPSE	Rd	Rr	Compare, skip if equal

Figure 7.9 shows the organization of the ALU module. It can be broken into 4 functional units – operand fetch unit, execution unit, skip evaluation unit and status flags evaluation unit. Operand fetch unit perform the fetching of operands to ORA and ORB, execution unit takes ORA and ORB and modified accordingly, status flags evaluation unit calculate the flags and send it to SR, and skip evaluation unit perform skip test. We will now assume that the control unit will send in the correct control signals at the correct time. More detail description about the control signals will be discuss in control unit section.

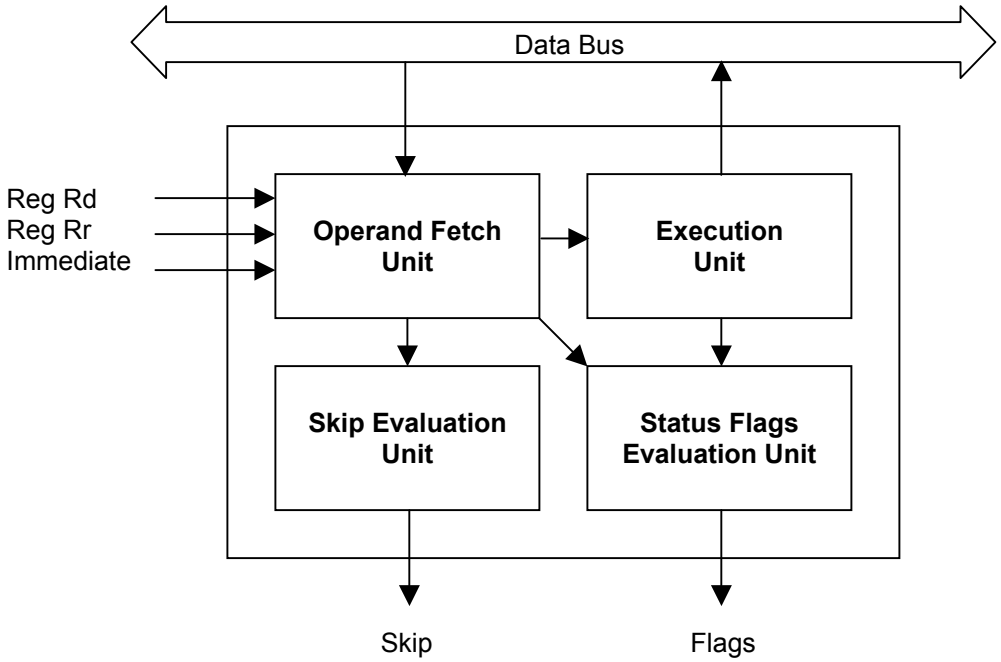


Figure 7.9 ALU Organization

7.6.1 Operand Fetch Unit

Figure 7.10 shows the structure of the operand fetch unit. There are two operand registers inside the operand fetch unit – operand register A (ORA) and operand register B (ORB).

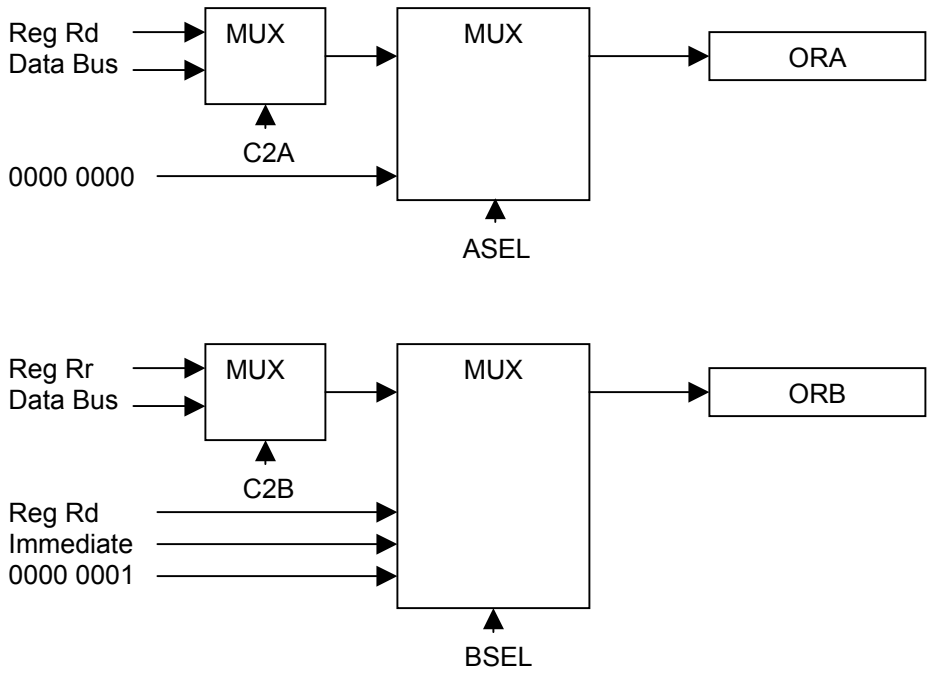


Figure 7.10 Operand Fetch Unit

ORA can be loaded with Rd or "0000 0000" while ORB can be loaded with Rr, Rd, immediate value or "0000 0001". They are selected by the ASEL signal and BSEL signal. The main purpose of the C2A signal and C2B signal is do perform operand forwarding as discussed in pipeline conflicts section in chapter 5. They are used to forward the result of the ALU (data bus) to the operand registers if the destination register of the currently executing instruction is found to be same as Rd or Rr

As discussed earlier, CBI, SBI, SBIC and SBIS instructions will fetch one of the I/O register to ORA as their operand. C2A also does the job by sending the data bus (that contains the I/O register value) to ORA.

7.6.2 Execution Unit

The execution unit executes 7 groups of instructions that are discussed earlier - 5 groups from the basic instructions (ADD, SUBCP, LOGIC, RIGHT and DIR), the bit load group and the pass ORA group. As shown in Figure 7.11, the execution unit is divided into 5 subunits. Adder-subtracter executes instructions from both the ADD and SUBCP group. Logic unit executes instructions from the LOGIC group. Shifter for the RIGHT group; direct unit for the DIR group; and bit loader for the bit load group.

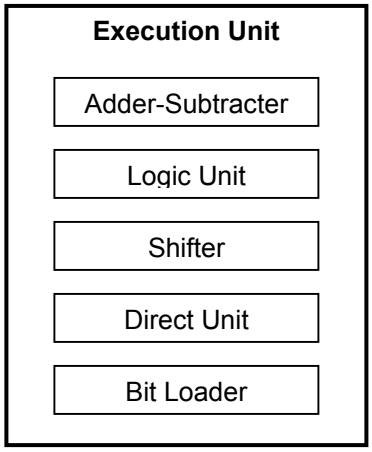


Figure 7.11 Execution Unit Organization

The adder-subtracter add ORA and ORB when the ADD signal is asserted, else it subtract ORB from ORA. Carry in of the adder-subtracter is determined by the ADD signal and WCARRY signal as shown in table 7.7.

Table 7.7 Carry In of Adder-Subtracter

ADD	WCARRY	Carry In	Related Instruction
0	0	1	SUB, SUBI, CP, CPI, DEC, NEG
0	1	Not C-Flag	SBC, SBCI, CPC
1	0	0	ADD, INC
1	1	C-Flag	ADC

The logic unit performs it operation based on the LOGICSEL signal. It performs a logical and between ORA and ORB when 00; logical or between ORA and ORB when 01; exclusive or between ORA and ORB when 10; and complement ORA when 11.

The shifter performs right shifting operation. The 7 least significant bits (LSB) of the result are the 7 most significant bits (MSB) of ORA. The result MSB is based on RIGHTSEL signal, which is '0' when 00; C-flag when 01; and the MSB of ORA when 10.

The direct unit performs direct data wiring based on the DIRSEL signal. It connects ORB to the result when DIRSEL is 0. If DIRSEL = 1, the 4 MSB of the result is the 4 LSB of ORA while the 4 LSB of the result is the 4 MSB of ORA (swap nibbles of ORA).

The bit loader receives the 3 control signals – BLD, CBISBI, BITSEL and SET. BLD signal loads the bit in ORA pointed by BITSEL with the T-flag. CBISBI signal will load the bit in ORA pointed by BITSEL with the value of SET.

The outputs of all the 5 units plus the value of ORA are multiplexed to the data bus through tri-state-buffers. Table 7.8 shows the data bus value with the respective control signal. 2 control signals asserted at the same time is impossible because each control signal represents totally different instructions and the CPU can only executes one instruction at a time. If none of the control signal in the table is active, nothing is sent out to the data bus, and it has a high impedance value. The data bus can be used for others purpose.

Table 7.8 Data Bus Value

Control Signal	Data Bus Value
ADD, SUBCP	Adder-Subtracter
LOGIC	Logic Unit
RIGHT	Shifter
DIRECT	Direct Unit
BLD, CBISBI	Bit Loader
PASSA	ORA
Default	High Impedance

7.6.3 Skip Evaluation Unit

Skip instructions (Table 7.6) are executed by first evaluating the skip condition by the skip evaluating unit. If the skip condition is fulfilled, the SKIP signal is asserted. The control unit will skip the next instruction that followed.

The skip evaluation unit receives 4 control signals – CPSE, SKIPTEST, BITSEL and SET. When CPSE signal is asserted, ORA are compared with ORB. The SKIP signal is asserted if they contain the same value. SKIPTEST test the bit in ORA pointed by BITSEL. If that bit has the same value as SET, the SKIP signal is asserted.

7.6.4 Flags Evaluation Unit

The 21 instructions (shown in table 7.3) that are executed by the ALU plus the BST instruction will modified the status register (SR). The control unit tests the status register bits to determine whether the branch of an unconditional branch instruction should be taken. A 7-bit wide flag bus is connected directly to SR to send the flags result. It is 7 but not 8 bit because none of these instructions modified the I-flag.

The ALU will evaluate all the 7 flags any time and send it to the SR through the flag bus. This will not create problems because every bit in SR receives an individual enable signal. The flag value send from the ALU will only be loaded if the enable signal for that bit is active. The control unit takes care of the enable signals. The ALU takes care of sending the correct flags to the bus. If a flag for an instruction is not modified, the ALU can send anything to that bus line.

Z-flag (Zero) is 1 when the result of an operation is zero. The evaluation unit can test the result (data bus) directly to determine the Z-flag. It works but it will slow down the design performance because signals need to pass through the execution unit and the tri-state buffers before reaching the data bus. Only instructions from ADD, SUBCP, LOGIC and RIGHT groups modified the Z-flag. To increase performance, the result of the adder-subtracter, logic unit and shifter is tested directly instead of the data bus.

N-flag (Negative) is always same as the value of the MSB of the result (bit 7). Again, testing the data bus will slow down performance. So, the N-flag is tested based of the result of the adder-subtracter, logic unit and shifter.

V-flag is directly generated by the adder-subtracter when performing ADD and SUBCP group instructions. It is always cleared for LOGIC instructions. For RIGHT instruction, the Boolean equation given in the datasheet is N-flag $\oplus$ C-flag. Recall that N-flag is equivalent to the MSB of the result (shifter result) and C-flag is the LSB of ORA.

S-flag (Sign) is an exclusive OR between the N-flag with the V-flag all the time.

C-flag (Carry) is the carry out of the adder-subtracter when performing ADD group instructions. RIGHT group instructions shift ORA one bit to the right and the LSB of ORA enter the C-flag. The COM instruction (from LOGIC group) always set the C-flag. For SUBCP instructions, C-flag is the borrow-in of the operation and is equal to the complement of carry out of the adder-subtracter.

H-flag (Half Carry) are modified by ADD and SUBCP group instructions. For ADD group, it is set if there is a carry out from bit 3 of the adder-shifter result. The Boolean equation for it is $A3.B3 + B3.\overline{C3} + \overline{C3}.A3$ with A is ORA, B is ORB and C is the adder-subtracter result. The H-flag is the borrow in from bit 3 and is given as $\overline{A3}.B3 + B3.C3 + C3.\overline{A3}$.

T-flag is always the bit in ORA pointed by BITSEL. When executing the BST (store bit to T-flag) instruction, the control unit simply asserts the enable signal for the T-bit in SR.

7.7 Status Register (SR)

The status register (SR) is mapped into the I/O space at $3F. Figure 7.12 shows the structure of the SR.

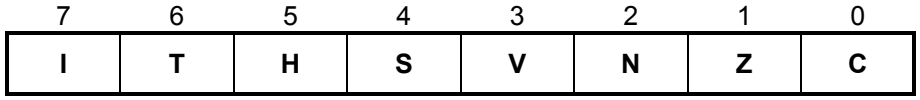


Figure 7.12 Status Register Structure

- Bit 7 – I: Global Interrupt Enable
- Bit 6 – T: Bit Copy Storage
- Bit 5 – H: Half Carry Flag
- Bit 4 – S: Sign Bit
- Bit 3 – V: Two's Complement Overflow Flag
- Bit 2 – N: Negative Flag
- Bit 1 – Z: Zero Flag
- Bit 0 – C: Carry Flag

All the flags except the I-flag have been discussed in the ALU section. The I-flag must be set to enable the interrupt. Only if the I-flag is set, an interrupt request can be served.

The SR can be modified in 4 conditions. First, the SR can be replace with the content of a general purpose register. This is done by writing to the I/O address $3F. The contents of the SR can also transferred to a general purpose register by reading the I/O address $35.

As discussed in the ALU section, the SR receives a 7-bit wide flag bus from the ALU. The C bus line is connected to the C-flag (a flip-flop); Z bus line to the Z-flag; and so forth. The I-flag is not connected to any bus line. The control unit sends in enable signals for all the flags (flip-flops). Only when the bit is enabled, the value of the bus line can be written into the flag.

If an interrupt request is served, the control unit will need to clear the I-flag before executing the interrupt service routine (ISR) so that another interrupt request will not be executed when serving the current one. When the ISR is completed (when RETI instruction is executed) the I-flag will be set again. So another interrupt request can be served. The control unit send 2 control signals to clear and set the I-flag.

Every bit in the SR can be cleared of set directly using the BCLR and BSET instructions. The SR receives BCLR, BSET and SRSEL signals. When BCLR is active, the flag pointed by SRSEL will be cleared. When BSET is active, the flag pointed by SRSEL will be set.

7.8 Data RAM

The actual AT90S1200 chip does not contain any SRAM. The AT90S2313 contains 128 Bytes of SRAM. Figure 7.13 show how the SRAM is organized in AT90S2313. The 32 general purpose registers and 64 I/O registers are mapped into the data space as well. The address space is accessed by LD and ST instructions with indirect addressing through the X-pointer, Y-pointer and Z pointer.

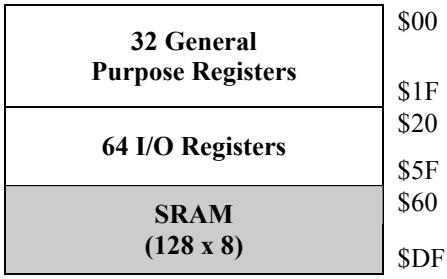


Figure 7.13 Data Address Space

In this design, the general purpose registers and I/O registers are not mapped into the data space. The data space consist of the the SRAM only, addressed from $60 to $DF. Only the Z-pointer (R30) is available. Data indirect with displacement is not supported.

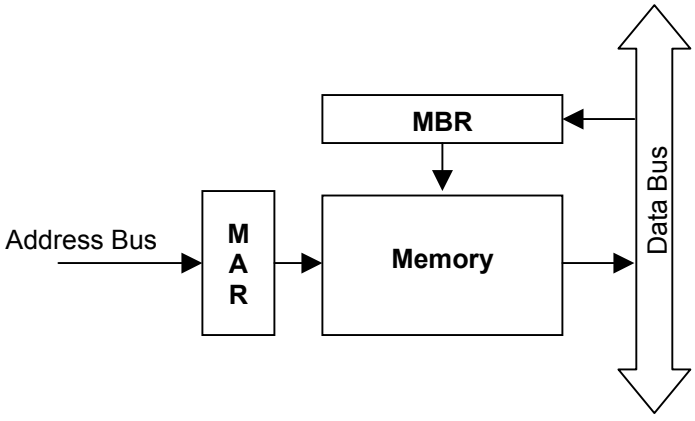


Figure 7.14 Data RAM Organization

Figure 7.14 shows the organization of the data RAM module. It contains two registers – memory address register (MAR) and memory buffer register (MBR). MAR is connected to the address input of the RAM. It receives data from the address bus which is send from the egister file. MBR are connected to the data input port of the RAM. It stores the data to be writen into the RAM. The MBR receives data from the data bus. A write operation will write the contents of the MBR to the memory addressed by MAR. A read operation will send the contents of the memory pointed by MAR to the data bus. If the read operation is not active, the RAM output will be tri-stated.

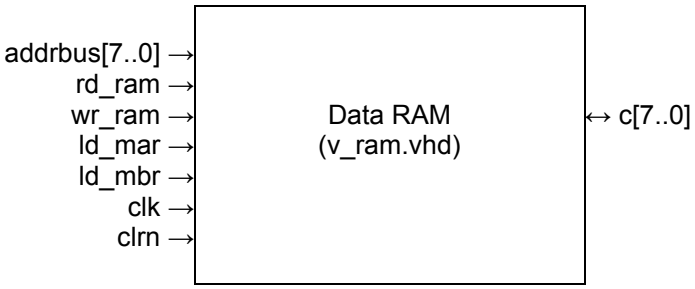


Figure 7.15 Data RAM Symbol

Figure 7.15 shows the symbol of the data RAM module. The RAM is implemented using LPM_RAM_DQ module from the LPM library. A special characteristic of this RAM is that it can have initial values by specifying the values in a MIF file. In this way, it also acts like a ROM as well. RD_RAM reads the content of the memory to data bus; WR_RAM writes the content of the MBR to memory; LD_MAR load the MAR with address bus; and LD_MBR loads MBR with data bus.

7.9 Port

There are three 8-bits bi-directional I/O ports in the design – Port B, Port C and Port D. Every port has its own data register, data direction register and input pins. They are mapped into the I/O space as listed in Table 6.1. All the data registers and data direction registers are cleared after reset. Data registers and data direction registers can be read and written to while the input pins can only be read.

A port is built using bit-slice approach where a single bit module is built and cascaded together to form the port. Shown in Figure 7.16 is the schematic of the bit-slice.

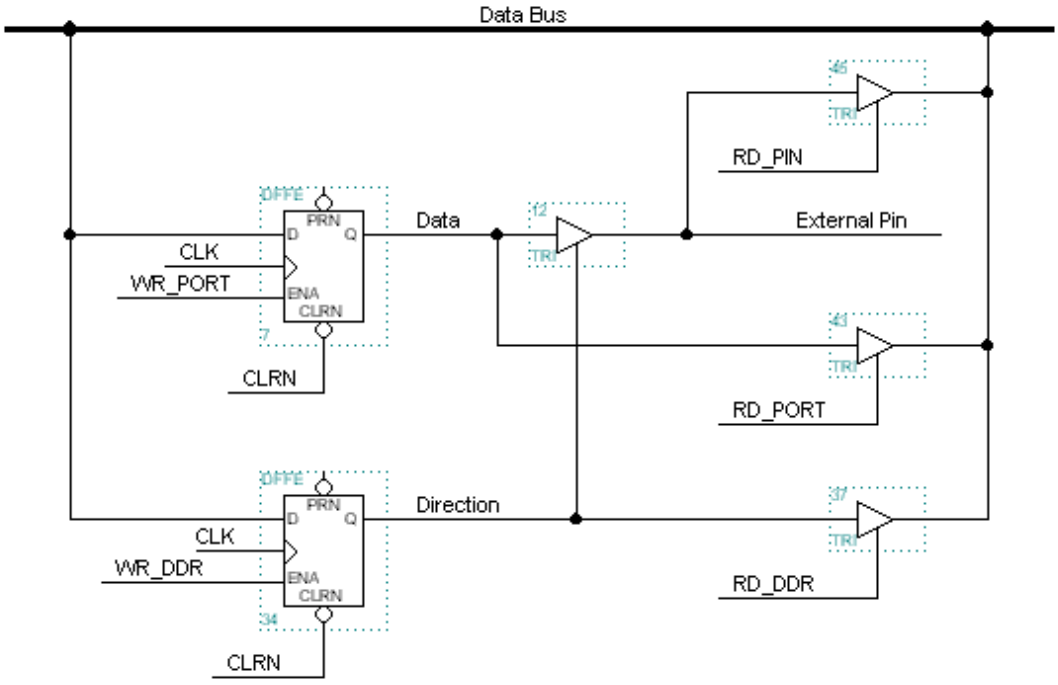


Figure 7.16 Bit-slice schematic of the I/O Port

The bit-slice contains 2 D-flip-flops, one is the data flip-flop while another is the data direction flip-flop. Data direction flip-flop control the direction of the I/O pin – 0 for input and 1 for output. When configure as input (direction = 0), the tri state buffer is not enable and the external bin will be in high impedanze state. A read on the pin will

read the value of the physical pin to the data bus. The data flip-flop value does not change according to the phisical pin.

When the pin is configured as output (direction = 1), the tri-state buffer that connects to the data flip-flop is now enabled. The physical pin will be directly driven the the value of the data flip-flop.

The port are connected directly to the data bus. When writing to the data flip-flop and direction flip-flop, data is received from the data bus and the write signal to the respected flip-flop is asserted. A read operation can read the contents of the data flip-flop, direction flip-flop and the external pin. A read signal to the respected flip-flops or pin will read its content to the data bus.

Eight copies of the same bit-slice are cascaded together to form a port module. Then the port module can be duplicate to form port B, port C and port D. Although they share the same port module, they are actually receiving different set of control signals from the control unit which differenciate them.

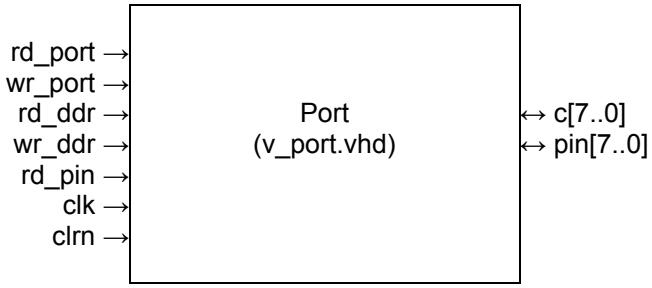


Figure 7.17 Port Symbol

Figure 7.17 shows the symbol of the port module. The port B will have the RD_PORT input connected to RD_PORTB signal while port C will have it connected to RD_PORTC, and RD_PORTD for port D. The same naming convention is applied to others control signals. The physical I/O pin is also named as PINB, PINC and PIND respectively.

7.10 Timer

The timer is a simple 8-bit timer with overflow detection and interrupt request. There are 4 control registers in the timer – timer/counter interrupt mask register (TIMSK) at $39, timer/counter interrupt flag register (TIFR) at $38, timer/counter 0 control register (TCCR) at $33 and timer/counter 0 (TCNT0) at $32. Figure 7.18 shows the control bits in these registers.

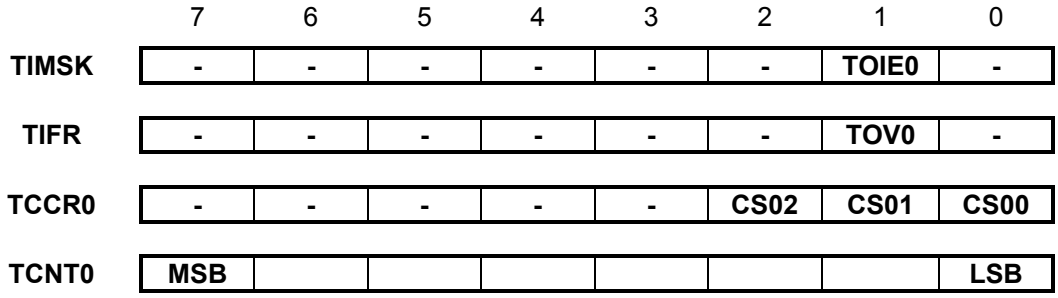


Figure 7.18 Timer Control Registers

The timer module contains a 10-bit prescaler/frequency divider drive by the system clock, which give a maximum division of 1024. CS02, CS01 and CS00 select the clock source for the timer according to Table 7.9.

Table 7.9 Timer Clock Source Select

CS02	CS01	CS00	Timer Clock Source
0	0	0	0 - the timer is stopped
0	0	1	System Clock (CLK)
0	1	0	CLK/8
0	1	1	CLK/64
1	0	0	CLK/256
1	0	1	CLK/1024
1	1	0	External Pin, falling edge
1	1	1	External Pin, rising edge

It is important to note that the timer clock source does not drive the TCNT0 directly. Instead, TCNT0 is driven by the system clock. The timer clock source are sampled at the rising edge of the system clock. If a low to high transition is detected (a low is sampled followed by a high), the increment signal for TCNT0 is asserted to increment it. Every transition detected will generate an increment signal pulse. If the timer clock source is the system clock, then no detection of rising edge is required - the increment signal is always asserted. To assure proper sampling of the external clock source, the frequency of the external clock should be smaller than the system clock frequency, and the smaller the better.

Every time the increment signal is active, TCNT0 will be incremented by 1. If TCNT0 is $FF before increment, it will become $00 after increment and at the same time the timer/counter 0 overflow flag (TOV0) will be set.

The timer/counter 0 interrupt overflow interrupt enable flag (TOV0) is ANDed with TOV0 to generate the timer overflow interrupt request. If the TOV0 is set (timer overflow interrupt enabled) and TOV0 is also set (timer overflow occurred), the timer will assert an interrupt request to the control unit. If the I-flag in the SR is enabled, the control unit will serve the interrupt request and clear the TOV0 flag by sending a clear TOV0 signal to the timer module.

Just like other control registers, the 4 timer registers can be read and write through the data bus. However, reserved bits are always read as zero; and the TOV0 flag can be cleared by writing a one to it. In this way, TOV0 flag can never be set by the user. Reserved bits are not implemented with flip-flops, they are connected directly to ground and this will save a lot of flip-flops. This is why the reserved bits are always read as zero and there are no way data can be written to them.

Figure 7.19 shows the symbol of the timer module. In this design, the EXTPIN is conneted to PIND7, the last pin of port D. It can easily configured to point to any of the 24 I/O pins. CLR_TOV0 is sent from the control unit to clear the TOV0 flag when the

interrupt request is served. The 4 RD signals read the timer control registers to the data bus while the 4 WR signals write the data bus value to the corresponding register.

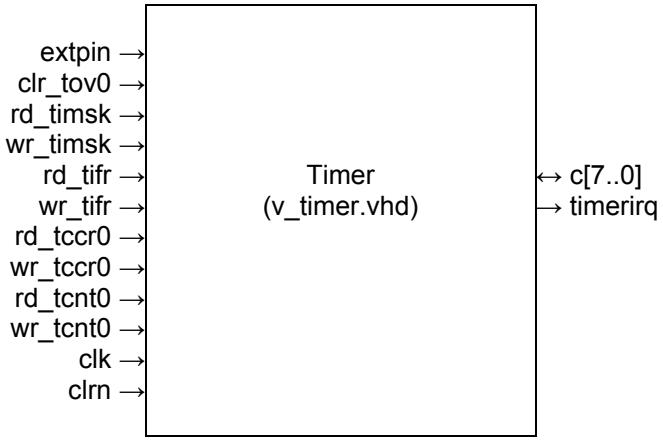


Figure 7.19 Timer Symbol

7.11 External Interrupt

The external interrupt is triggered by an external pin. In this design, the external pin share the pin with pin D7, the last pin of port D. This pin can be easily changed to share with one of the 24 I/O pins by modifying a singal line in the VHDL code. Shown in Figure 7.20 is the 2 control registers for external interrupt – general interrupt mask register (GIMSK) at $3B and MCU control register (MCUCR) at $35.

	7	6	5	4	3	2	1	0
GIMSK	-	INT0	-	-	-	-	-	-
MCUCR	-	-	-	-	-	-	ISC01	ISC00

Figure 7.20 External Interrupt Control Register

The MCUCR of AT90S1200 has the bits 4 and 5 for controlling the sleep modes of the microcontroller. Since the design does not include this feature, these bits are taken away from the register.

The interrupt can be triggered by the external pin on rising edge, falling edge of low level and is selected by the ISC01 and ISC00 bits (interrupt sense control 0) as shown in Table 7.10.

Table 7.10 Interrupt Source

ISC01	ISC00	Interrupt Source
0	0	Low Level
0	1	-
1	0	Falling Edge
1	1	Rising Edge

The interrupt can also be triggered when the external pin is configured as output. The difference now is that the interrupt signal is provided internally from the microcontroller instead of external signal. This provides a way to generate software interrupt by the programmer.

Transitions (falling edge and rising edge) are not detected using the clock input of a flip-flop. The external pin is sampled on every system clock to detect the transitions. A low sample follows by a high sample sense a rising edge while a high sample follows by a low sample sense a falling edge. When the interrupt source is set to falling or rising edge, the external interrupt flag will be set when the require edge is detected. The external interrupt flag are not accessible by the user. It is not placed inside any of the control register. The flag will stay until the interrupt request is served or after a reset.

Figure 7.21 shows how interrupt request is generated. To generate an interrupt request to the control unit, the INT0 bit (external interrupt request 0 enable) must be set. This bit is ANDed with the flag to generate the interrupt request (with ISC /= 00).

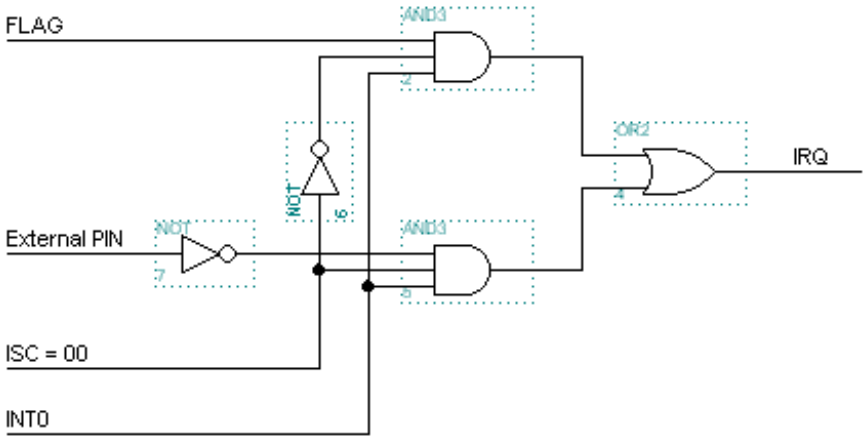


Figure 7.21 Generating External Interrupt Request

Low-level interrupt are difference from edge interrupt just discussed. It does not set the external interrupt flag to generate an interrupt request. Instead, it never touches the flag. The complement of the external pin (detect low-level) is directly ANDed with the INT0 bit to generate an interrupt request. So if INT0 is set, it will generate an interrupt request as long as the pin is held low. If the interrupt is not enabled when the pin is held low, it will be forgotten when the pin goes high.

If the external interrupt is set to edge triggered, the external signal must have sharp transition. If a physical switch is used to generate the interrupt, switch-bounce will occur. It will generate a second, third or more interrupt request even if the interrupt request has already been served. So, it is recommended that the low-level interrupt is used, or the switch is hardware de-bounced.

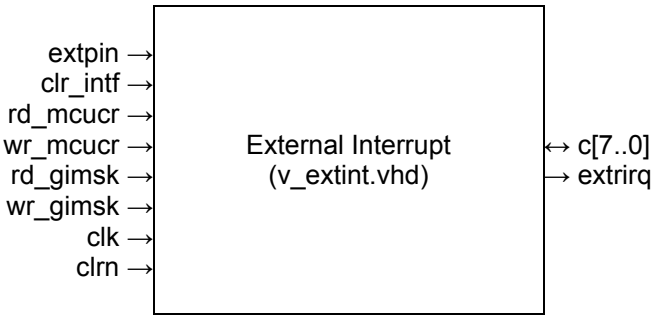


Figure 7.22 Timer Symbol

Figure 7.22 shows the symbol of the external interrupt module. CLR_INTF is sent by the control unit to clear the external interrupt flag when the interrupt request is served. RD and WR signals provide reading and writing the control registers through the system data bus.

CHAPTER VIII

CONTROL UNIT DESIGN

8.1 Chapter Overview

The design of the datapath has been discussed in the last chapter. Only one module is left for the design – the control unit module, which will be discussed in this chapter. We have touched the instruction set, pipeline processing and many control signals, which controls the datapath. The control unit plays the role on decoding the instruction, implements the pipeline processing and asserts the control signals for the datapath at the correct timing. This chapter covers the decoding of the instruction and the design of the finite state machine (FSM).

8.2 Instruction Decoder

The inputs of the control unit are the instruction machine code from instruction register (IR), the flags value from status register (SR), skip request, timer interrupt request (timer IRQ) and external interrupt request (external IRQ). The machine code is decoded first before sending to the FSM, while the others inputs are connected directly to the FSM.

As discussed in chapter 5, the design process involves 51 machine codes. The instruction decoder takes the 16-bit machine code from the IR and generates 46 output signals to represents the 51 instructions. There are 4 pairs of instructions that share a same signal. The share signal is active when either one is found. They are BRBC, BRBS; SBRC, SBRS; SBIS, SBIR; CBI, SBI. The NOP instruction is not decoded. So 51 take away 5 equals to 46 signals.

At any time, the IR can only have one instruction. So, it will not have more than one output signal active at a time. However, if the machine code received does not match any of the 51 instructions, or is actually the NOP instruction, then none of the decoder output signal is active. When none of the output signal is active, the FSM will not assert any control signal to perform an operation, so no operation (NOP) is executed in that cycle. Any undetermined instruction is executed as NOP.

8.3 Synchronous Mealy Model Finite State Machine

RISC control unit should be hard-wired (logic gates) rather than microprogrammed (ROM implementation). Microprogrammed control unit is used by CISC because the instruction has different length and execution cycles. So microprogrammed can make the control unit design easier. The disadvantage is slower speed performance. In RISC, instruction has fixed length and mostly single cycle execution. So design using hard-wired is not that complicated and it will have the advantage of speed.

The FSM in this design is hard-wired, using logic gates to generate the next state and output signals rather then a ROM. The FSM is implemented using synchronous Mealy model. Figure X.X shows the block diagram of a synchronous Mealy model FSM.

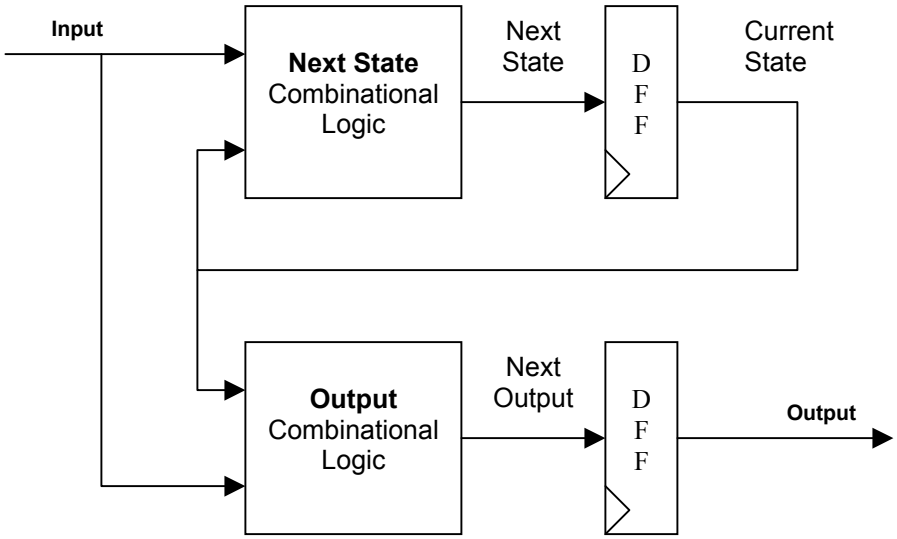


Figure 8.1 Synchronous Mealy Model FSM

Different with the normal Mealy FSM, the synchronous Mealy FSM has their output connected to flip-flops. That is why it is called synchronous. There are two combinational logics in the state machine, one to generate the next state base on the input and current state, while the other is used to generate the outputs base also on the input and current state.

There are basically 2 advantages from using a synchronous Mealy FSM. For a Moore or Mealy FSM, the outputs are generated by the output combinational logic. They will be delay for the signals to pass through the combinational logic before the output is generated. This will slow down the control signals output speed. If the datapath receive the control signals later, then will perform their operation later. In the synchronous case, outputs are still generated by the combinational logic, but they are now gated to D-flip-flops. On the next clock transition, the outputs are asserted immediately. The datapath receives the control signals at the very beginning of a cycle and therefore can complete its operation faster. This is the first advantage.

The FSM contains only 8 states. Such a small number of states are results of using synchronous Mealy implementation. This is the second advantage. Since the state machine outputs are now gated to flip-flops, all single cycle instruction can share the same state. The state is unchanged but the input changed, so it can determine the next output.

8.4 Finite State Machine States

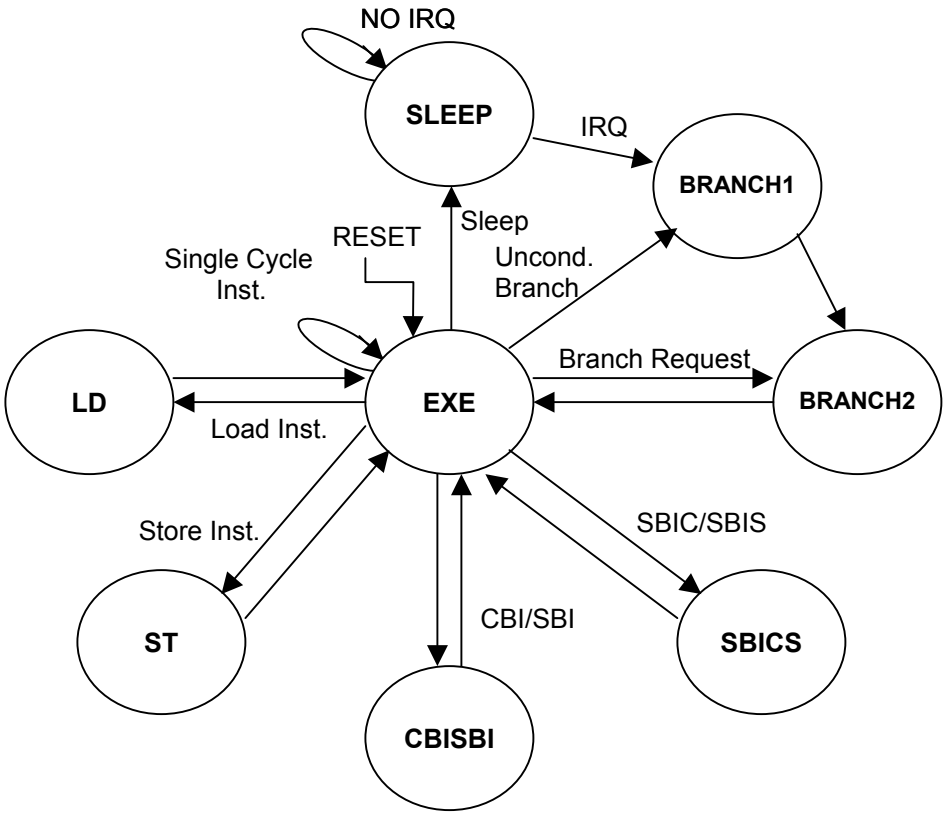


Figure 8.2 State Diagram

Figure 8.2 shows the state diagram of the finite state machine (FSM). The 8 states are EXE (execute), SLEEP, BRANCH1, BRANCH2, SBICS (skip if bit in I/O clear/set), CBISBI (clear/set bit in I/O), ST and LD.

The state diagram shows the state flow but does not clearly show the inputs. The inputs to the FSM are the 46 output lines of the instruction decoder, timer IRQ, external IRQ, skip request and branch request. Branch request is generated by the branch evaluation unit when the condition of the conditional branch instruction is fulfilled.

We now assume all instructions are single cycle and there are no IRQ, skip request and branch request. The state machine will have no state change in this case and remain at EXE state. All instructions have a fetch cycle and an execute cycle and are pipelined together as discussed in chapter 5. When the first instruction is fetched, its corresponding output line of the instruction decoder will become active. It happens in the fetch stage. The next state combinational logic finds that the next state is unchanged. However, the output combinational logic has prepared the control signals based on the decoder's active line. On the next clock transition, the instruction enter the execute stage and the control signals is asserted (latch into the output flip-flops). The ALU then executes the instruction. Because of pipeline processing, the next instruction has been fetched at the same clock transition. The instruction decoder decodes it and asserts another output line. Again, the output logic will prepare the correct control signals and asserts it on the next clock transition. So the FSM can perform the pipeline processing without any difficulty.

We now consider the one of the unconditional branch instruction - RJMP. When RJMP is fetched, the RJMP output line of the decoder is active. The next state logic determined that there would be a state change to BRANCH1 state on the next cycle. The output logic also prepared the control signals for RJMP, which will load the PC with the destination address. On the next clock transition, state changes to BRANCH1 and the control signals are asserted. At BRACH1, the next state must be BRANCH2. Although the pre-fetched instructions asserts one of the decoder output line, the output logic does

not prepared any control signals for the next cycle. So this instruction is being flushed from the pipeline, as discussed in chapter 5. So on the following clock transition, state changes to BRANCH2 and at the same time, PC is loaded with the new value. The next state will be returned to EXE state. Again, no output signal is asserted based on the fetched instruction because it is flushed. On the next clock transition, the FSM enters EXE state and the destination instruction has been fetched. The decoder's destination instruction output line is active and will be executed on the next cycle.

The discussion above is for the RJMP instruction. The same concept can be applied to RCALL, RET, RETI instructions as well as serving an IRQ. An IRQ (timer or external) is sent by the timer or external interrupt module in the datapath. An IRQ can only be served if the I-flag is set, else it will be ignored. To make sure all instruction s are completely executed, an IRQ can be only be served in the EXE state. On EXE state, the FSM first check for any IRQ (must have the I-flag set). If there is any, it will ignored the pre-fetched instruction and determines the next state to be BRANCH1. The output logic prepare control signal to load PC with the interrupt vector and to clear the I-flag. I-flag is cleared so that if there is a new IRQ occurred while serving the current one, it will not be served. After loading the interrupt vector to the PC, execution continues as normal but there will not be any IRQ served until the RETI instruction is fetched and executed. It will then set back the I-flag and allowed another IRQ to be served. All conditional branch instruction will take 3 cycles to complete. This can be count from the transitions make to complete the execution from EXE state back to EXE state. (EXE → BRANCH1 → BRANCH2 → EXE)

The next case to consider is the execution of conditional branch instructions – BRBC and BRBS. Different from conditional branch instruction, the branch may or may not be taken. They test a bit in the SR to determine whether the branch should be taken. The branch evaluation unit will do the job on testing the SR flags base on the condition specified. If the condition is fulfilled, it will immediately generate a branch request to the FSM.

When either BRBC or BRBS is fetched, the shared instruction decoder output line become active. Different from unconditional branch instructions, there will be no state chance on the next cycle. The FSM will assert the branch test signal on the next cycle to request the branch evaluation unit to perform a branch test. If the condition is not fulfilled, no branch request is generated. The pre-fetched instruction is not flushed from the pipeline and is executed. So it takes only one cycle for a conditional branch instruction if the branch if not taken.

If the condition is fulfilled, the branch evaluation unit will send back a branch request to the control unit immediately. At the same time, the control unit will also instruct the PC to loads the PC with the destination address. With the branch request, the FSM will transfer to BRANCH2 state on the next clock and the pre-fetched instruction is flushed. On the next clock, the second pre-fetched instruction is also flushed but the FSM now return to EXE state. The next instruction is the destination instruction and will be executed on next cycle. So it takes 3 execution cycles if the branch is taken for conditional branch instructions. Note that the control signal to load the PC is not asserted according to clock transition. It is asserted only after the branch evaluation unit has received the branch test signal and performs the test successfully. So there is delay for the PC to receive the signal in this case.

When the FSM sees the SLEEP instruction, it will jump to the SLEEP state. When in the SLEEP state, the PC is stopped and no instruction is executed. Only when there is an IRQ (with the I-flag set), the FSM jumps to BRANCH1 state to serve the interrupt request. The process is exactly the same as serving an IRQ from the EXES.

For single cycle instruction, the instruction will not need to be remembered after the control signals is asserted because it is completed in one cycle. When enter the execute cycle, the next instruction is fetched and the current instruction is lost. However, instructions that require 2 cycles to complete must have some way to remember the instruction in order to assert the correct control signals at the second cycle. So, the FSM

provides the second state to remember the instruction. Control signals are based on the state itself without considering the decoder's output line.

If the second cycle of the instructions asserts the same control signals, then the state can be shared, else it will require another one. There are 4 states of all for executing 2 cycles instruction – LD, ST, CBISBI and SBICS. The FSM jump to LD state when LD Z, LD +Z or LD –Z is seen; ST if ST Z, ST +Z, ST –Z; CBISBI if CBI or SBI; SBICS if SBIC or SBIS. When one of these instructions is found, the control unit will need to hold the pipeline (Chapter 4). The EN signal send to the PC module and IR module will not be asserted for one cycle. So the PC is not incremented and the IR is still holding the pre-fetched instruction.

Skip instructions executes in a similar way to unconditional branch instructions. When the FSM sees a skip instruction, it will send control signals to the ALU to perform the skip test. The ALU will send a skip request back to the FSM if the skip condition fulfilled. The skip request will not generate a state chance as branch instructions. However, it will ignore the pre-fetched instruction (the instruction to be skipped). No control signal is asserted to execute it. So it takes 1 cycle if the skip is not taken but 2 if the skip is taken.

SBIC and SBIS is a combination of 2 cycles instruction and skip instruction. It requires an extra cycle to fetch the I/O register before the skip can be test by the ALU. The skip test signal is asserted on the transition from SBICS to EXE. If a skip is taken, it takes 3 cycles and it takes 2 is the skip is not taken.

After the long discussion, we should notice when in the EXE state, it will first check to see if there are any branch request or skip request to processed (two of them will never occurred at the same time). If none, it will then check the IRQ. The IRQ must be enabled by the I-flag in order to be served. Only after then it checks the instruction decoder's output to execute an instruction.

8.5 Finite State Machine Output

The finite state machine (FSM) output are the control signals send to control the datapath. The datapath and their control signals have been discussed in Chapter 7. The FSM will generate these control signals at the correct timing. Table 8.1 lists the control signals and the instructions/ state/ condition that assert them.

Table 8.1 Control Signals

Module	Control Signal	Instruction/ State/ Condition
PC	ADDOFFSET	RJMP, RCALL, Branch Request
	PUSH	RCALL, Timer IRQ, External IRQ
	PULL	RET, RETI
	VEC2	External IRQ
	VEC4	Timer IRQ
PC & IR	EN	Other than (CBI, SBI, SBIC, SBIS, LD Z, LD Z+, LD –Z, ST Z, ST Z+, ST –Z)
General Purpose Register File	WR_REG	ADD, ADC, INC, SUB, SUBI, SBC, SBCI, DEC, NEG, AND, ANDI, OR, ORI, EOR, COM, LSR, ROR, ASR, LDI, MOV, SWAP, IN, LD State
	INC_ZP	LD Z+, ST Z+
	DEC_ZP	LD –Z, LD -Z
ALU	ADD	ADD, ADC, INC
	SUBCP	SUB, SUBI, SBC, SBCI, CP, CPC, CPI
	LOGIC	AND, ANDI, OR, ORI, EOR, COM
	RIGHT	LSR, ROR, ASR
	DIR	LDI, MOV, SWAP
	BLD	BLD
	CBISBI	CBISBI state
	PASSA	OUT, ST Z, ST +Z, ST –Z
	CPSE	CPSE
	SKIPTEST	SBRC, SBRS, SBICS State
	LOGICSEL	Refer to Table 7.3
	DIRSEL	Refer to Table 7.3
	RIGHTSEL	Refer to Table 7.3
SR	BCLR	BCLR
	BSET	BSET
	EN for C-flag	ADD, ADC, SUB, SUBI, SBC, SBCI, CP, CPC, CPI, NEG, COM, LSR, ROR, ASR

	EN for S,V,N,Z-flag	ADD, ADC, INC, SUB, SUBI, SBC, SBCI, CP, CPC, CPI, DEC, NEG, AND, ANDI, OR, ORI, EOR, COM, LSR, ROR, ASR
	EN for H-flag	ADD, ADC, SUB, SUBI, SBC, SBCI, CP, CPC, CPI, NEG
	EN for T-flag	BST
	CLR_I	Timer IRQ, External IRQ
	SET_I	RETI
Data RAM	LD_MAR	LD Z, LD +Z, LD –Z, ST Z, ST +Z, ST –Z
	LD_MBR	ST Z, ST +Z, ST –Z
	RD_RAM	LD state
	WR_RAM	ST state
Timer	CLR_TOV0	Timer IRQ
External Interrupt	CLR_INTF	External IRQ
I/O Decoder	RD_IO	IN, CBI, SBI, SBIC, SBIS
	WR_IO	OUT, CBISBI state, SBICS state
Branch Evaluation Unit	BRANCH_ TEST	BRBC, BRBS

8.6 Fetch Stage Signals

Signals discussed so far are execute stage signals, which means they are asserted at the execute stage of an instruction. But there are also fetch stage signals, which are asserted at the fetch stage of the instruction.

The C2A and C2B signals are the operand-forwarding signals. There are logics in the control unit that compare the Rd bits of the current executing instruction (in IBR) with the Rd and Rr bits of the newly fetched instruction (in IR). If it is found to be the same as either, or both, C2A or C2B will be asserted immediately. So on the next clock, the operand register will loads the results of the ALU to the operand register instead of the general register.

As discussed in the ALU operand fetch unit section is Chapter 7, ASEL and BSEL control signals are used to select what should be loaded into the operand registers. They are generated directly by the instruction decoder's output. Table 8.2 shows the value of ASEL and BSEL with the corresponding instructions and operands.

Table 8.2 C2A and C2B Operand Fetching Signals

ASEL	ORA	Instruction
0	Rd	Default
1	0000 0000	NEG

BSEL	ORB	Instruction
0	Rr	Default
1	Rd	NEG
2	Immediate Value	SUBI, SBCI, CPI, ANDI, ORI, LDI
3	000 0001	INC, DEC

8.7 Instruction Backup Register (IBR)

IR is always loaded with the next instruction, then IBR will always loaded with the currently executing instruction. So it is actually loading the contents of the last IR. Bits in the IBR are used to form the destination register address for the register file; the bit select signal (BSEL) for the ALU (select one of the 8 bits in a register); the SET signal for the ALU (for bit loading, bit test); the flag select signal (SRSEL) for SR; and the OFFSET for the PC.

8.8 I/O Decoder

When either the RD_IO or WR_IO is asserted, the I/O decoder will decode the I/O address to know exactly which I/O register are to be read of write. Then it sends out the specific read or write control signal for that I/O. In the instruction format section in chapter 4, it is shown that there are two instruction formats for instructions that accessed the I/O. So the bits location for the I/O address is different. The I/O decoder must be able to know which bits are to be used as the I/O address.

8.9 Branch Evaluation Unit

A conditional branch instruction will test one of the 8 bits in the SR. BRBC will take the branch if the specific bit is cleared while BRBS will take the branch if that bit is set. The branch evaluation unit is enabled when the BRANCH_TEST signal is active. It will then test whether the specific bit meets the branch condition (clear/set). If it does meet the condition, a branch request is generated immediately to the control unit to generate the ADDOFFSET control signal, the next state will now be BRANCH2 state. If the condition is not fulfilled, nothing happens and the CPU will execute the next instruction.

CHAPTER IX

HARDWARE IMPLEMENTATION

9.1 Altera UP1 Educational Board

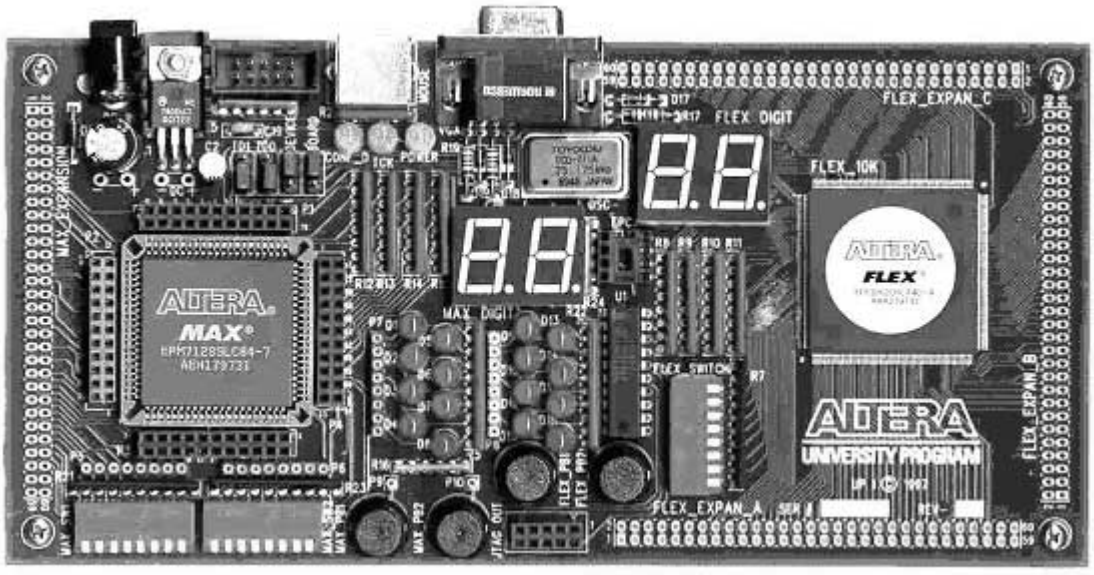


Figure 9.1 Altera UP1 Educational Board

The Altera UP1 (University Program) Educational Board as shown in Figure 9.1 is the only FPGA device available in the LAB. It has two FPGAs on it for developing complex programmable logic applications. The MAX7000 device on the left side of the board typically supports 2,500 gates for simple designs. The FLEX10K20 on the right

supports 20,000 gates, and includes connections to a DB25 VGA connector, as well as a PS/2 mouse port. The system is programmable via a PC parallel port, using the included MAX+PLUS II Student Edition.

This design is targeting the FLEX10K20 device. The exact device code is EPF10K20RC240-4. 240 means the package has 240 external pins; while –4 is the speed grade of the device. This device is the second smallest in the FLEX10K series. Designing larger digital system might be challenging if it is the only device available for implementation. Imagine that implementing 32x8 bit register with two 16-1 8-bit multiplexers will swallow up 52% of the logic cells! There are 4 speed grades for FLEX devices, -4, -3, -2 and –1. Unfortunately, -4 is the slowest grade. Although the area and speed constraints may lead to harder design process, however it will be more challenging and require more knowledge on the device architecture in order to minimized the area used and maximized the speed performance.

9.2 Pin Assignments

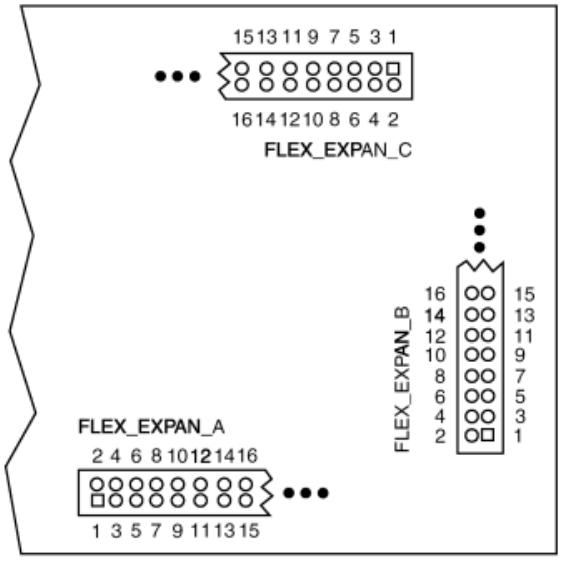


Figure 9.2 FLEX10K Pins Arrangement on UP1 Board

Figure 9.2 shows the pin arrangement of the FLEX10K device on the UP1 board. Before the design is programmed into the device, pin assignments must be made to map all the pins of the design to the physical pin on the UP1 board. Table 9.1 lists the pin assignments used.

Table 9.1 Pin Assignments

Design Pin	Map To	Design Pin	Map To
RESET	29	PINC3	73
CLK	91	PINC4	75
PINB0	45	PINC5	79
PINB1	48	PINC6	80
PINB2	50	PINC7	82
PINB3	53	PIND0	79
PINB4	55	PIND1	81
PINB5	61	PIND2	83
PINB6	63	PIND3	86
PINB7	65	PIND4	88
PINC0	66	PIND5	95
PINC1	68	PIND6	98
PINC2	71	PIND7	100

RESET pin is mapped to one of the onboard switch. The CLK is mapped to the build-in clock source (25.9MHz). All the I/O pins (Ports) are connected to the FLEX_EPAN_A pins. Since the clock source is faster than the design maximum speed (around 12 MHz). A frequency divider must be used to divide the clock source by 4 (6.5 MHz) before driving the whole system.

9.3 External Circuitry

The board itself is not sufficient to test the design. If the board is now a microcontroller, then the external circuitry for the control applications must be prepared. The external circuit will be connected to the 24 I/O pins of the microcontroller.

Port B is configured as output and is used to control two 7-segments LED display through the used of two BCD to 7-segments decoder. The 4 lower bits of the port will drive the right digit while the 4 upper bits of the port drive the left digit.

Lower 4 bits of port C is configured as output and is connected to 4 common VCC red LEDs. The LED will on when the pin output a LOW logic. Upper bits of port C is configured as input and is connected to 4 momentary normally open push buttons. The other end of the push buttons is connected to GND. The input pin will sense a LOW logic is the button is pressed.

PINB0	1	2	3
PINB1	4	5	6
PINB2	7	8	9
PINB3	*	0	#
	PINB4	PINB5	PINB6

Figure 9.3 Keypad Interfacing

7 pins of port D is used to interface a 4 x 3 keypad. Figure 9.3 shows the connection between he keypad and the pins. The lower 4 bits are configured as input and are connected to the 4 rows of the keypad. These 4 bits are also connected to 4 pull-up resistors. The following 3 bits are configured as output and is connected to the 3 columns of the keypad.

The last pin (Pin D7) is the external interrupt request pin and is connected to a push button. The configuration of the push button is the same as the push buttons for port C. Pin D7 is also connected to a green LED. So this pin will be configured both as input and output depends on the program. It will sense the push-button when configured as input and it will on/off the LED when configured as output.

9.4 Fitting Report

```
***** Project compilation was successful



** DEVICE SUMMARY **

Chip/                    Input Output Bidir  Memory  Memory            LCs
POF       Device         Pins  Pins   Pins   Bits % Utilized  LCs  % Utilized

v_riscmcu
      EPF10K20RC240-4     2      0      24   10240     83 %    1068     92 %

User Pins:                2      0      24
```

Figure 9.4 Fitting Report

Figure 9.4 shows the fitting report of the whole design. The first line tells us that the project has been compiled successfully. There are 2 input pins (CLK and RESET) and 24 I/O pins (Port B, C and D). 83% of the memory is utilized. Memory is implemented in the embedded cells (EC) in the device. The program ROM and data RAM uses EC. 92% of the logic cells (LC) are utilized. LC is the most basic logic building block in the device.

9.5 Control Applications

We have got a microcontroller in the FPGA and the external circuitry. Now we need the have the program for the control application. 2 control applications are used to test the microcontroller. The programs for the applications are listed in Appendix B. The program must be assembled and changed to MIF format. Maxplus2 then compiled it along with the designed microcontroller.

9.5.1 Simple Calculator

The first application is a simple calculator that can only perform add and minus operations. The keypad is the input of the calculator and the two 7-segments digits are the output. The # key is used to represent the add (+) key while the * key is used to represent the minus (-) key. There are no equal (=) key, the results is automatically shows is the results changed after an operation (add or minus). To clear the result, external interrupt is used. The external interrupt will clear all the saved data when requested.

All operations are done with BCD numbers. The microcontroller detect a key pressed on the keypad and changed it to the BCD number it represents. Operations are done in BCD directly so the C-flag and the H-flag of the status register are used. Then the results is shown and saved as BCD.

If an overflow occurred after an operation, interrupt is temporary disabled and pin D7 (the external interrupt pin) is configured as output to on the green LEDs for a short delay to indicate an overflow has occurred.

The timer is also tested in this application. The timer is enabled and the interrupt mask bit is set. The interrupt service routine will on a red LED and rotate it through the 4 red LEDs. Since it is controlled by the timer interrupt, it does not affect the main program (the calculator) and thus the microcontroller is multitasking, detecting keys and generating running lights at the same time.

When a key is pressed and is holding, the microcontroller will take only one data and it will only detect another key when the current key is released. The application is also software de-bounced.

9.5.2 Simple Memory Game

The simple memory game will display random red LEDs blink (one at a time). The player will need to remember the sequence of the LEDs blink and tell the microcontroller by using the push buttons. A player is given 3 lifes for the whole game. If the player gets it right, the green LED blinks once and the game proceed to the next level. If the player gets it wrong, all the red LEDs blink once and the blink sequence is shown again. One life will be deducted.

The first level will have only 1 LED in the blink sequence and the second level increased to 2. The higher the level, the longer the blink sequence. The two digits 7-segments display will always shown the current level of the game. If the player entered a wrong sequence, the life is deducted and the remaining life is shown before the game shows the sequence again.

Random numbers are used to determine which red LEDs should be on next. Random numbers are generated by the used of timer. Every time a random number is needed, the microcontroller read the timer and get a value, then it processed the random number to decide which of the 4 LEDs should be on.

This program must have complicated software for input detection. It must be able to detect and count the key press very accurately. Let say in level 8, there will be 8 LEDs blink in sequence. When the player keying in the result, the microcontroller must takes in 8 inputs, and check them will the saved value. So it must have very accurate detection on whether the current key has been released before taking the next one. It must also have software de-bounced.

CHAPTER X

SUGGESTIONS AND CONCLUSION

10.1 Recommendation on Future Works

At first, the microcontroller does not contain any data RAM. So the stack is implemented using hardware just like AT90S1200 and is only 4-level deep. At the end of the design process, data RAM has been included due to the extra time the author have. Future works should have the stack implemented in the data RAM using a stack pointer. This will save up some area and more important, the stack will be able to keep a few times more entry then the original hardware stack.

There is only one indirect pointer, the Z-pointer in this design. If memory access is frequent, more indirect pointers would make the job easier. Future works should also include the X-pointer and Y-pointer.

There are many more extra features available in the AVR RISC microcontroller family, such as the UART serial interface, SPI serial interface, the 16-bit timer (with output compare and input capture), etc. This works from this project should be used as a platform to implement these features in.

10.2 Conclusion

As a conclusion, this project has been completed successfully fulfilling are the objectives and scopes specified. The author has used his extra time to optimized the speed of the design until 12 MHz. The data RAM that is not specified in the scope of the project has also been included. Hardware stack is enlarged to 4-level instead of 3 and a total of 24 I/O lines are available. Since the project now occupies 92% of the FPGA device (FLEX10K20), the author recommends that the laboratory provides a larger FPGA device. Table 10.1 is the comparison chart between AT90S1200 and the current design.

Table 10.1 AT90S1200 VS Current Design

Specification	AT90S1200	Current Design
Instructions	89	92
G.P Registers	32	16
Program ROM	512 words	512 words
SRAM	None	128 bytes
Hardware Stack	3 Level Deep	4 Level Deep
I/O Ports	2 (15 pins)	3 (24 pins)
Addressing Modes	5	7
Speed	4 MHz / 12 MHz	12 MHz
8-bit Timer	1	1
External Interrupt	1	1
Implementation	CMOS	FPGA
Others	Analog Comparator, Watch Dog Reset, EEPROM, Internal Pull Up Resistors	None

Reference

[1] Daniel Tabak, *RISC Systems*, Research Studies Press Ltd.: Taunton, Somerset, England TA1 1HD, 1990

[2] M.Morris Mano, *Computer System Architecture*, Prentice Hall inc.: Englewood Cliffs, New Jersey 07632, 1993.

[3] *AVR 8-bit RISC Microcontrollers Data Book*, Atmel Corporation, San Jose, California 95131, August 1999.

[4] Randy H. Katz, *Contemporary Logic Design*, The Benjamin/Cummings Publishing Company, Inc.: Redwood City, California 94065, 1994.

[5] Douglas L. Perry, *VHDL*, McGraw-Hill Companies, Inc.: Singapore, 1999.

[6] Jan Gray, *Building a RISC system in an FPGA: Part 1,2 & 3*, Circuit Cellar Magazine (http://www.circuitcellar.com), 2000.