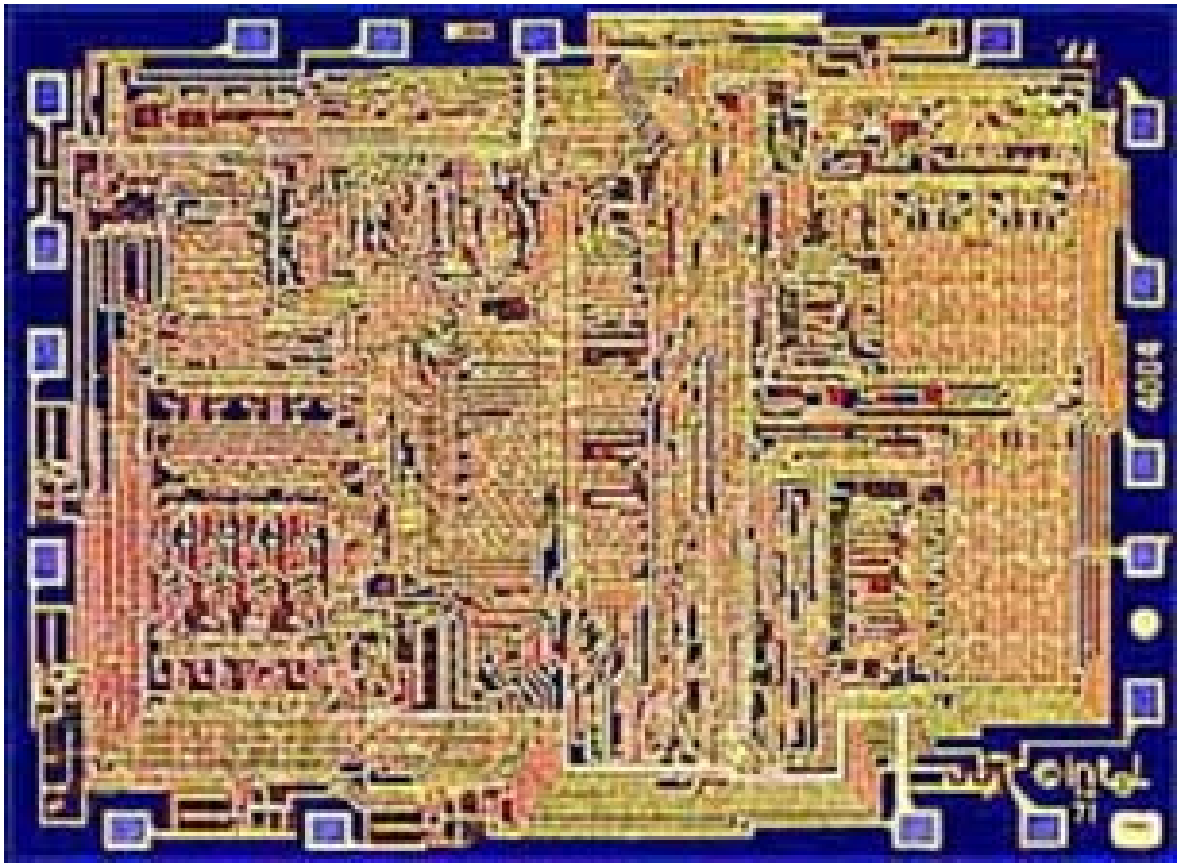
ASICS Coursework

CPU Design using the Altera MAX2PLUS package to design, analyse and prove the system.



UNIVERSITY of ULSTER — school of electrical and mechanical engineering

COURSE: BEng (HONS) Electronic Systems
MODULE: EEE515J2 - ASICS & VLSI Design
BY: Colin K McCord
DATE: Thursday, 17 July 2003

TABLE OF CONTENTS

1.0. INTRODUCTION

The propose of this assignment is to use the Altera Max2Plus package to design, analyse and prove working a simple CPU that can execute a few instructions.

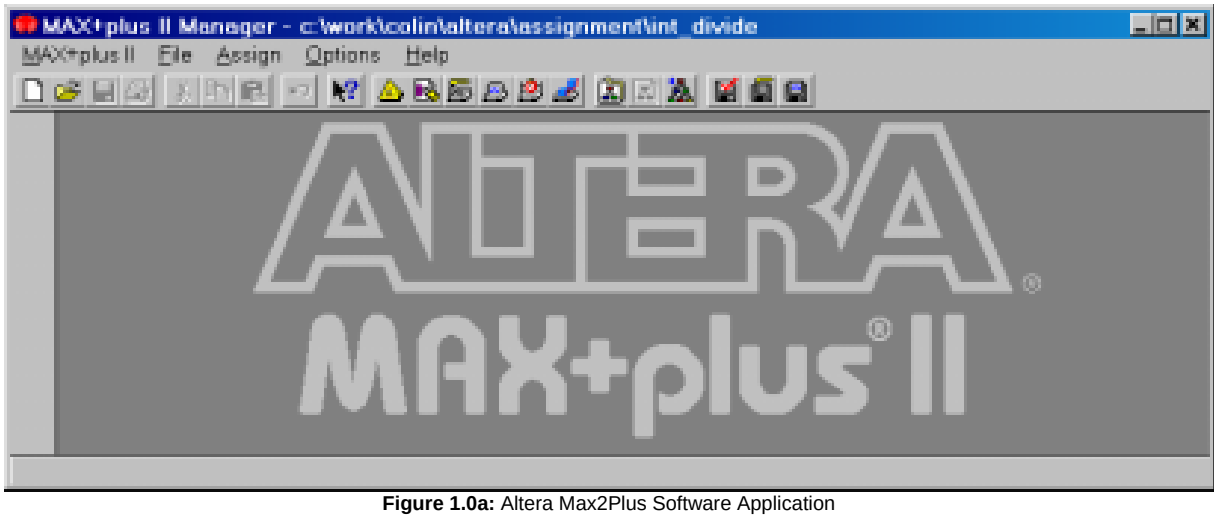


Figure 1.0a: Altera Max2Plus Software Application

The MAX+PLUS®II development software is a fully integrated programmable logic design environment. This easy-to-use tool supports the Altera® ACEX™, FLEX® and MAX® programmable device families and works in both PC and UNIX environments. The MAX+PLUS II software offers unmatched flexibility and performance and allows for seamless integration with industry-standard design entry, synthesis, and verification tools. By giving the designer entry freedom and the ability to mix and match design entry methodologies, the MAX+PLUS II software minimizes re-design work.

This report includes schematic capture and other design details, as well as an English description on how the design works. The design is partitioned into functional blocks, each block does one task that is easy to describe. As well as the design details, the simulation details are included for most functional blocks within the design. The simulation results are explained, this is done by annotating the waveforms produced by the system.

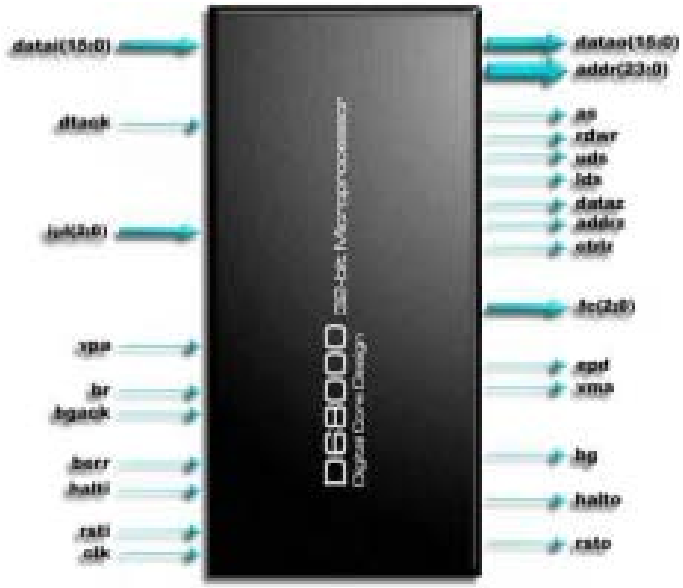


Figure 1.0b: "Digital Core Design" D68000 32-bit Microprocessor

The report is split into 7 sections for clarity (including this introduction): -

1.0. Introduction,
2.0. Background Information
3.0. Simple Hypothetical Computer
4.0. CPU Design
5.0. CPU Simulation
6.0. Conclusions
7.0. Appendixes

Extremely powerful commercial CPU cores exist; "Digital Core Design" D68000 is available for purchase and is compatible with Max2Plus (see Appendix 2, for details).

The CD-ROM attached to appendix 7, contains all the Max2Plus design, symbol and simulation files for the designed CPU.

2.0 BACKGROUND INFORMATION

2.1. Brief History of Microprocessors

"The first microprocessor was developed by what was then a small company called Intel (short for Integrated Electronics) in the early 1970s. The client, a Japanese company called Busicon, declined to buy the chipset and Intel, faced with a development cost and no customer, decided to market the chipset as a "general purpose" micro processing system for use in applications where digital logic chips would have been used. The chipset was a success and within a short while Intel developed a general purpose 4 bit microprocessor called the 4004." [W4]

1971: Intel introduces the 4-bit bus, 108-KHz 4004 chip - the first microprocessor. It operates at 60,000 operations per second. It has 2300 transistors and can address 640 bytes.

1972: Intel introduces the 200-KHz 8008 chip, the first 8-bit microprocessor. It can access 16KB of memory, uses 3500 transistors, and has a speed of 60,000 instructions per second. Texas Instruments introduces the TMS1000 one-chip microcomputer. It integrates 1KB ROM and 32 bytes of RAM with a simple 4-bit processor.

1974: Intel releases its 2-MHz 8080 chip, an 8-bit microprocessor. It can access 64KB of memory. Motorola introduces its 6800 chip, an early 8-bit microprocessor that can access 64KB.

1976: Intel introduces the 5-MHz 8085 microprocessor. It supports an 8-bit bus. Texas Instruments introduces the TMS9900, the first 16-bit microprocessor. Zilog releases the 2.5-MHz Z80, an 8-bit microprocessor whose instruction set is a superset of the Intel 8080.

1978: Intel introduces the 4.77-MHz 8086 microprocessor. It uses 16-bit registers, a 16-bit data bus and can access 1 MB of memory.

1983: Intel introduces the 6-MHz 80286 microprocessor. It uses a 16-bit data bus and offers protected mode operation. It can access 16 MB of memory, or 1 GB of virtual memory. Later versions operate at 10-MHz and 12-MHz.

1984: Apple Computer officially unveils the Lisa computer. It features a 5-MHz 68000 microprocessor,1MB RAM, 2MB ROM, a 12-inch B/W monitor, 720x364 graphics, dual 5.25-inch 860KB floppy drives, and a 5MB Profile hard drive. It is slow, but innovative. It is the first personal computer graphical user interface (GUI). Its initial price is £10,000. NEC introduces the 8-MHz V20 microprocessor, the first clone of Intel's 8088 and the 8-MHz V30 microprocessor, the first clone of Intel's 8086. Motorola introduces the 68020, a 32-bit microprocessor. IBM announces the PC AT, a 6MHz 80286 computer using PC-DOS 3.0, a 5.25-inch 1.2MB floppy drive, with 256KB RAM, for £4000, which doesn't include hard drive or monitor/card. With a 20MB hard drive, colour card and monitor: £6700.

1985: Motorola introduces the MC68HC11 Intel introduces the 16-MHz 80386DX microprocessor. It uses 32-bit registers and a 32-bit data bus. Initial price is £299. It can access 4 gigabytes of physical memory, or up to 64 terabytes of virtual memory.

1987: Intel introduces the 20-MHz 80386DX microprocessor. IBM introduces the IBM Personal System/2 (PS/2) line, with IBM's first 386 PC, and 3.5-inch floppy drives as standard. The PS/2 Model 30 uses a 8-MHz 8086, the Model 50 and 60 use the 10-MHz 80286, and the Model 80 uses a 20-MHz 80386. Zilog introduces its Z-280 16-bit version of the Z-80 CPU.

1991: Intel introduces the 20-MHz i486SX microprocessor, the 486DX, but no math coprocessor. Intel introduces the 50-MHz 486 microprocessor.

1993: Intel introduces the Pentium processor. It uses 32-bit registers, with a 64-bit data bus, giving it an address space of 4 GB. It incorporates 3.1 million transistors. Speeds are 60-MHz (>£800) and 66-MHz (>£900). IBM and Motorola introduce the 80-MHz version of the PowerPC 601 processor. IBM and Motorola introduce the 66-MHz and 80-MHz version of the PowerPC 603 processor.

The development of more recent microprocessor architectures such as the Harvard architecture and the use of Reduced Instruction Set Computers (RISC) have led to the development of microcontrollers such as the Microchip PIC.

2.2. Central Processing Unit (CPU)

"The central processing unit is the central component of a digital computer. Its purpose is to interpret instruction codes received from memory and perform arithmetic, logic, and control operations with data stored in internal register, memory words, or I/O interface units. Externally, the CPU provides a bus system for transferring instructions, data, and control information to and from the modules connected to it.

A typical CPU is usually divided in two parts: the processor unit and the control unit. The processor unit consists of an arithmetic logic unit, a number of registers, and internal buses that provide the data paths for the transfer of information between the registers and the arithmetic logic unit.

The control unit consists of a program counter, an instruction register, and timing and control logic. The control logic may be either hardwired or micoprogrammed. If it is hardwired, registers, decoders, and a random set of gates are connected to provide the logic that determines the action required to execute the various instructions. A microprogrammed control unit uses a control memory to store microinstructions and a sequencer to determine the order by which the microinstructions are read from control memory." [B1]

2.3. Von Neumann CPU Architecture

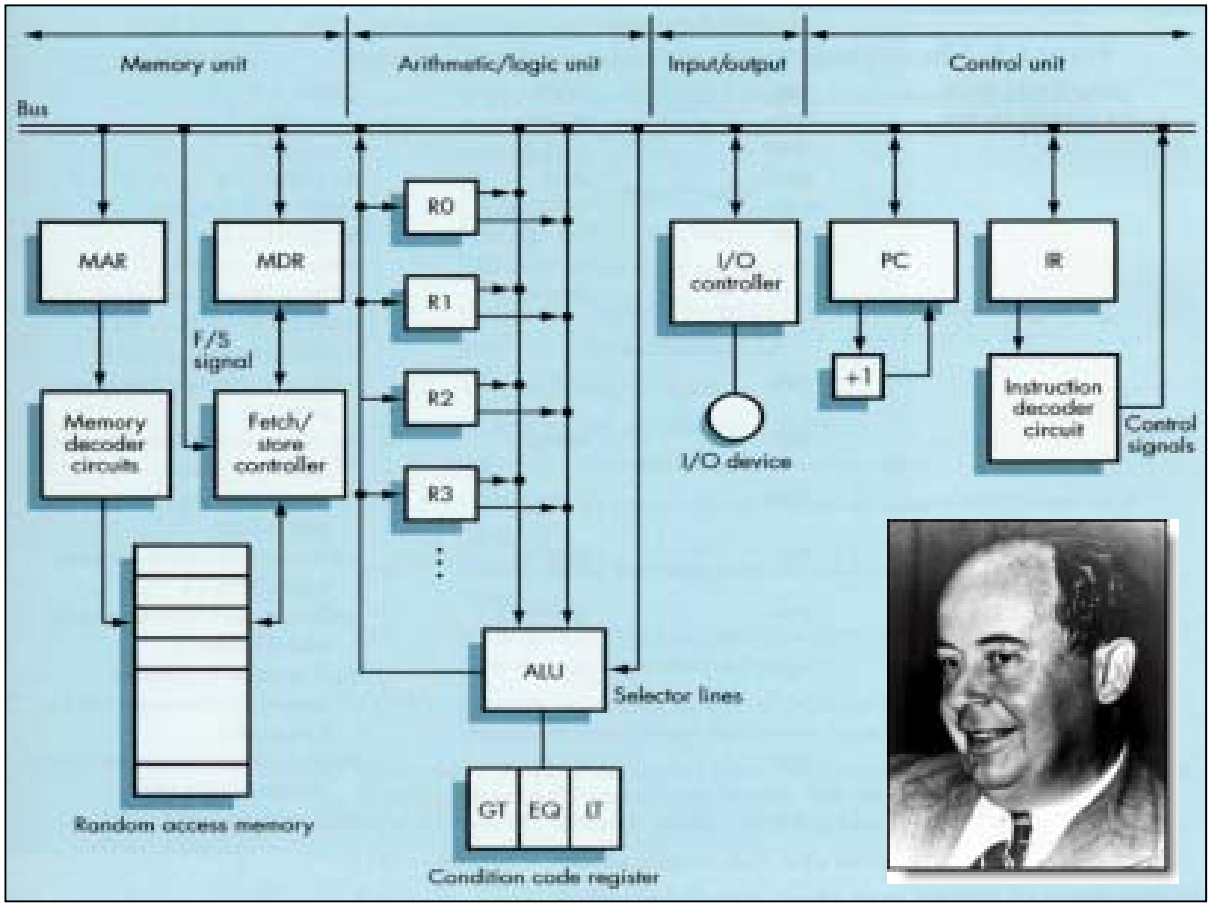


Figure 2.3a. The classic Von Neumann CPU Architecture

Classic model for designing and building computers, (shown in figure 2.3a) which is based on three characteristics: The computer consists of four main sub-systems (memory, ALU, Control Unit, Input/Output

System (I/O), program is stored in memory during execution and program instructions are executed sequentially.

2.4. Machine code

"The sequence of micro operations that are incorporated in a CPU is determined from the instructions that are defined for the computer. The instructions are stored in memory as part of a program and are presented to the control unit in a binary form. Each binary coded instruction contains a number of fields that provide the pertinent information needed by the control to execute of fields that provide the pertinent information needed by the control to execute the instruction in the CPU. The binary coded form of the instruction is defined by its instruction format. Different computer have different instruction formats; this is one aspect that gives each computer its special hardware characteristic." [B1]

Acc. & memory		Immed			Direct			Index			Extend			Inher			CCR					Description
Operation	Mnemonic	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	H	N	Z	V	C	
ADD	adda	8B	2	2	9B	3	2	AB	5	2	BB	4	3				√	√	√	√	√	[A] <- [A] + [M]
	addb	CB	2	2	DB	3	2	EB	5	2	FB	4	3				√	√	√	√	√	[B] <- [B] + [M]
Add accums	aba													1B	2	1	√	√	√	√	√	[A] <- [A] + [B]
ADd with Carry	adca	89	2	2	99	3	2	A9	5	2	B9	4	3				√	√	√	√	√	[A] <- [A] + [M] + C
	adcb	C9	2	2	D9	3	2	E9	5	2	F9	4	3				√	√	√	√	√	[B] <- [B] + [M] + C
AND	anda	84	2	2	94	3	2	A4	5	2	B4	4	3				•	√	√	0	•	[A] <- [A] · [M]
	andb	C4	2	2	D4	3	2	E4	5	2	F4	4	3				•	√	√	0	•	[B] <- [B] · [M]
BIT test	bita	85	2	2	95	3	2	A5	5	2	B5	4	3				•	√	√	0	•	[A] · [M]
	bitb	C4	2	2	D4	3	2	E4	5	2	F4	4	3				•	√	√	0	•	[B] · [M]
CLeaR	clr							6F	7	2	7F	6	3				•	0	1	0	0	[M] <- #00
	clra													4F	2	1	•	0	1	0	0	[A] <- #00
	clrb													5F	2	1	•	0	1	0	0	[M] <- #00
CoMPare	cmpa	81	2	2	91	3	2	A1	5	2	B1	4	3				•	√	√	√	√	[A] - [M]
	cmpb	C1	2	2	D1	3	2	E1	5	2	F1	4	3				•	√	√	√	√	[B] - [M]
Compare accums	cba													11	2	1	•	√	√	√	√	[A] - [B]
COMplement (1's)	com							63	7	2	73	6	3				•	√	√	0	1	[M] <- [$\overline{M}$]
	coma													43	2	1	•	√	√	0	1	[A] <- [$\overline{A}$]
	comb													53	2	1	•	√	√	0	1	[B] <- [$\overline{B}$]
Complement (2's)	neg							60	7	2	70	6	3				•	√	√	1	2	[M] <- #00 - [M]
	nega													40	2	1	•	√	√	1	2	[A] <- #00 - [A]
	negb													50	2	1	•	√	√	1	2	[B] <- #00 - [B]
Decimal Adjust A	daa													19	2	1	•	√	√	√	3	Adjusts sum of BCD bytes to BCD formatted byte
DECrement	dec							6A	7	2	7A	6	3				•	√	√	4	•	[M] <- [M] - #1

Figure 2.4a. Some of the machine code instructions available on the 6800 microprocessor [W1]

A machine-code program is a sequence of machine-code instructions. To be executed, a machine-code program must be stored in main memory, execution taken place within the CPU. Each instruction is executed in a number of small steps (controlled by control unit, e.g. microcode): -

- Fetch the next instruction from memory into the instruction register.
- Change the program counter to point to the next instruction.
- Decode the current instruction.
- If the instruction uses a word in memory, determine where it is.
- Fetch the word.
- Execute the instruction.
- … and repeat until the instruction is 'sleep'

This execution follows the 'fetch-decode-execute' cycle

2.5. Microcode

The sequence of micro-operations in the CPU is controlled by means of a micro-program stored in control memory. The micro-program consists of microinstructions that control the data paths and operations of the CPU. The micro-program can control not only the micro-operations in the processor unit but also all other data paths in the CPU. It is also capable of decoding computer instructions and obtaining the operands according to the specified addressing code.

Basically each microinstruction specifies which control signals are to be generated.

2.5.1. Horizontal Microcode

Horizontal microcode has one bit for each possible control signal, execution is very simple and underlying hardware is very simple. But microinstructions are very wide as there is one bit for each control signal.

Horizontal microcode example: -

L[3..0]	ACCS	MDRS	RESS	RESE	IRS	PCS	PCI	PCE	MARS	MARE	ALUC2	ALUC1	ALUC0	ROME	RAMS	RAME	INPUT	OUTS
0001	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

2.5.2. Vertical Microcode

Controls signals are grouped into sets, two can be in the same set only if never used at the same time. Separate field for each set and each field is encoded. Each combination of bits in each field represents either one control signal, or “no signal”. A decoder for each field is used to execute a microinstruction. Microinstructions are short with a minimal number of fields and a minimal number of bits per field.

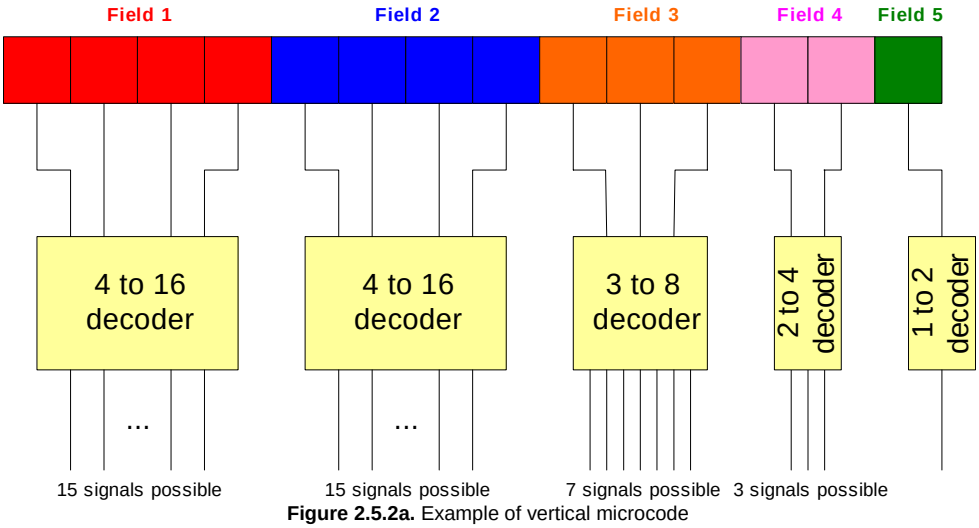


Figure 2.5.2a. Example of vertical microcode

At most 1 signal can be generate for each field at one time. A total of up to 5 signals can be generated at one time.

2.5.3. Comparisons & Trade-Offs

Horizontal microcode requires a larger micro-memory because the microinstructions are longer, simpler hardware can be used execute it, e.g. just generate a signal for each bit. Vertical microcode needs less storage in micro-memory and requires more complex hardware for execution (e.g. several decoders).

2.6. Hard-wired Control Logic Vs Microcode

Hard-wired logic is faster than microcode, but has a more complex circuit design and layout and small changes may require major revisions, e.g. layout and implementation may have to be reworked. Microcode is easier to test and debug while prototyping, easier to change e.g. only microcode memory needs to be altered, but its slower because it depends on memory technology.

2.7. Microprocessors Vs Microcontrollers

The term, microprocessor, commonly refers to a general-purpose Central Processing Unit (CPU). They are powerful, suitable for all types of computations and require additional hardware components to support communications and storage.

The term, microcontroller, commonly refers to a Central Processing Unit (CPU) that has been specialised to control the operation of a mechanical or electronic system. They are small and cost-effective, built-in memory and specialised built-in interface support for some of the following: -

- High-speed communication.
- Parallel devices.
- Serial devices.
- Analogue devices.

2.8. The PIC Microcontroller

A PIC (**P**eripheral **I**nterface **C**ontroller) microcontroller is an IC manufactured by Microchip.

These ICs are complete computers in a single package. The only external components necessary are whatever is required by the I/O devices that are connected to the PIC.

The traditional Von-Neumann Architecture (Used in: 80X86, 8051, 6800, 68000, etc…) is illustrated in Figure 2.8a. Data and program memory share the same memory and must be the same width.

"All the elements of the von Neumann computer are wired together with the one common data highway or bus. With the CPU acting as the master controller, all information flow is back and forward along these shared wires. Although this is efficient, it does mean that only one thing can happen at any time. This phenomenon is sometimes known as the von Neumann bottleneck." [B2]

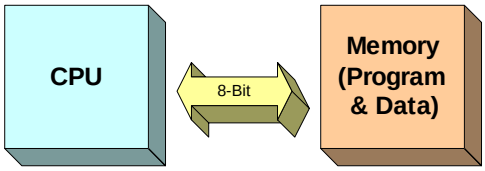


Figure 2.8a. Simplified illustration of the von Neumann architecture

PICs use the Harvard architecture. The Harvard architecture (Figure 2.8b) is an adaptation of the standard von Neumann structure with separate program and data memory: data memory is made up by a small number of 8-bit registers and program memory is 12 to 16-bits wide EPROM, FLASH or ROM.

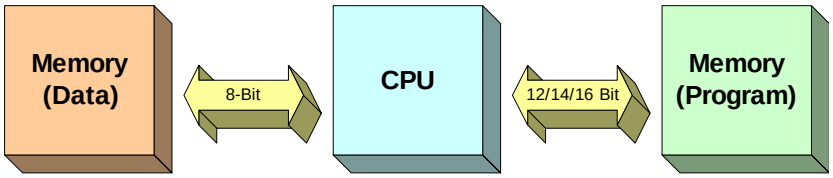


Figure 2.8b. Simplified illustration of the Harvard architecture

Traditional CISC (**C**omplex **I**nstruction **S**et **C**omputer) machines (Used in: 80X86, 8051, 6800, 68000, etc…) have many instructions (usually > 100), many addressing modes and it usually takes more than 1 internal clock cycle to execute. PIC microcontrollers are RISC (**R**educed **I**nstruction **S**et **C**omputer) machines, which

have 33 (12-bit) to 58 (15-bit) instructions, reduced addressing modes (PICs have only direct and indirect), each instruction does less, but usually executes in one internal clock.

"The combination of single-word instructions, the simplified instruction decoder implicit with the RISC paradigm and the Harvard separate program and data buses gives a fast, efficient and cost effective processor implementation." [B2]

2.8.1. Summary of the PICs Built-in Peripherals

SPI (**S**erial **P**eripheral **I**nterface) uses 3 wires (data in, data out, clock), Master/Slave (can have multiple masters), very high speed (1.6 Mbps), and full speed simultaneous send and receive (full duplex).

I^2C (**I**nter **IC**) uses 2 wires (data and clock), Master/Slave. There are lots of cheap I^2C chips available; typically < 100kbps.

UART (**U**niversal **A**synchronous **R**eceiver/**T**ransmitter) with baud rates of 300bps to 115kbps, 8 or 9 bits, parity, start and stop bits, etc. Outputs 5V hence an RS232 level converter (e.g. MAX232) is required.

Timers, both 8 and 16 bits, many have prescalers and some have postscalers. In 14 bit cores they generate interrupts. External pins (clock in/clock out) can be used for counting events.

Ports have two control registers: TRIS sets whether each pin is an input or an output and PORT sets their output bit levels. Note: Other peripherals may steal pins, so in this respect peripheral registers control ports as well. Most pints have 25mA source/sink (LED enabled), but not all pins, it is important to look up the datasheet. Floating input pints must be tied off (or set to outputs).

ADCs (**A**nalogue to **D**igital **C**onverter) are currently slow, less than 54 KHz sampling rate (8, 10 or 12 bits), theoretically higher accuracy when PIC is in sleep mode (less digital noise) once the sample is complete the ADC sends an interrupt waking the PIC. Note that the PIC must wait until the sampling capacitor is charged; see datasheets.

2.9. RSIC VS CISC

CISC (Complex Instruction Set Computer) have many instructions, some of them being very complex, this makes programming easier (Typical PC architecture). Historically, machines tend to add features over time, for example IBM 70X, 70X0 series went from 24 opcodes to 185 in 10 years (performance also increased 30 times), with additional addressing modes and special purpose registers. The motivations for this complex instruction set are to improve efficiency, since complex instructions can be implemented in hardware and executed faster, make life easier for compiler writers and support more complex higher-level languages.

RISC (Reduced Instruction Set Computer), have few instructions, reduced addressing modes and each instruction does less, but usually executes within one internal clock pulse (e.g. a PIC microcontroller). Simple, limited instruction set with a large number of general purpose registers and optimised instruction pipeline. RISC offer faster execution of instructions commonly used (e.g. 20% of the instructions do 80% of the work) and faster design and implementation.

	Year	No. of Instructions	Instruction Size	Address Modes	Registers
IBM 370/168	1973	208	2 – 6	4	16
VAX 11/780	1978	303	2 - 57	22	16
M 88000	1988	51	4	1	32
I 80486	1989	235	1 - 11	11	8
IBM 6000	1990	184	4	2	32
MIPS R4000	1991	94	4	1	32

Figure 2.9a. Comparison of some architectures

Typically, RISC takes about 1/5 the design time, but CISC have adopted RISC techniques.

2.10. Superscalar Architectures

Superscalar machines issues a variable number of instructions during each clock cycle, up to some maximum. Instructions must satisfy some criteria of independences, for example a simple choice could be a maximum of one fp and one integer instruction per clock, need separate execution paths for each possible simultaneous instruction issue. Compiled code from non-superscalar implementation of same architecture runs unchanged, but slower.

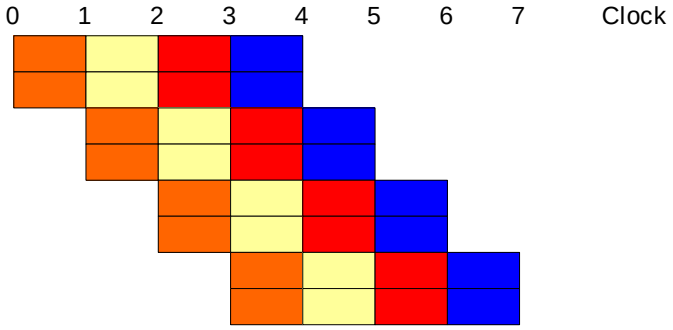


Figure 2.10a. Superscalar Example, Note each instruction path may be pipelined

A common superscalar problem is instruction-level parallelism, for example what if two successive instructions can't be executed in parallel (data dependencies, or two instructions of slow type).

2.11. VLIW Architectures

Very Long Instruction Word (VLIW) architectures store several simple instructions in one long instruction fetched from memory: -

- Number and type are fixed, e.g. 2 memory reference, 2 floating point, one integer.
- Need one functional unit for each possible instruction, e.g. 2 fp units, 1 interger unit, 2 MBRs all run synchronised.
- Each instruction is stored in a single word, e.g. required wider memory communication paths and many instructions may be empty, meaning wasted code space.

2.12. Instruction Level Parallelism

Success of superscalar and VLIW machines depends on a number of instructions that occur together that can be issued in parallel (i.e. no dependencies, no branches). Compliers can help create parallelism and speculation techniques try to overcome branch problems, e.g. assume branch is taken, execute instructions but don't let them store results until status of branch is know.

3.0. SIMPLE HYPOTHETICAL COMPUTER

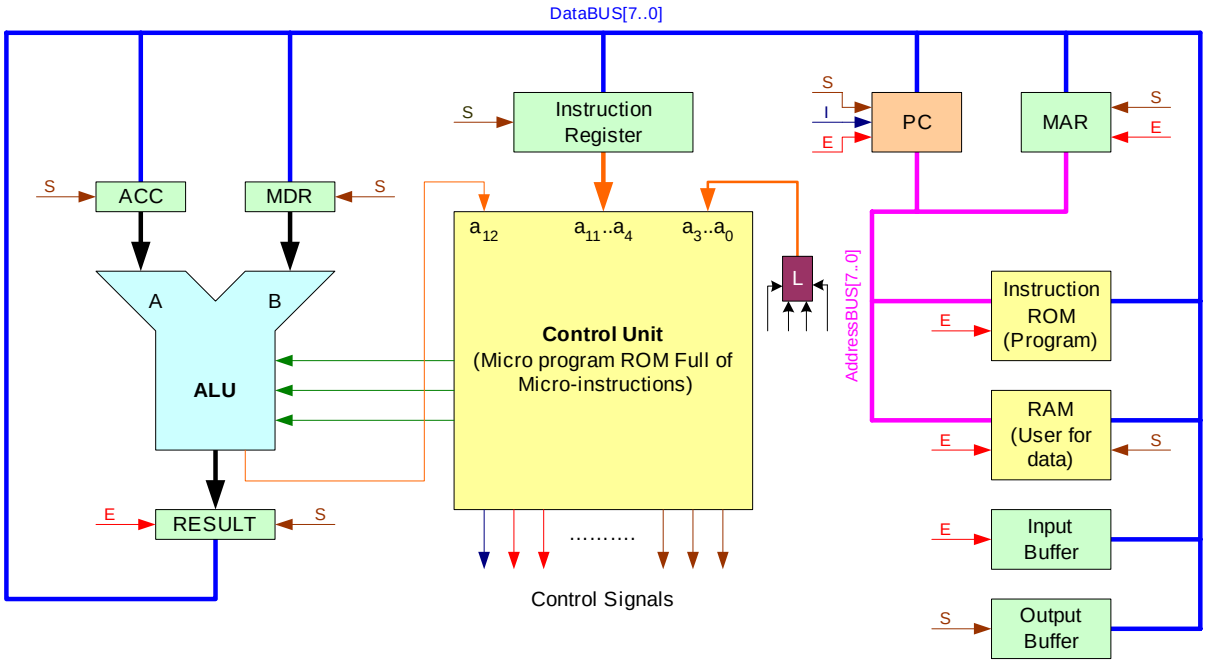


Figure 3.0a. Simple hypothetical computer block diagram

The "Control Unit must emit signals on 22 different bits for each microinstruction. Most registers require an 'S' strobe signal to latch data into the register; some required an 'E' enable signal to allow the register contents to appear on a BUS.

No BUS can have more than one register enabled at the same time (for instance the PC and MAR outputs). The ALU has 3 control lines to allow addition, subtraction (A-B), inversion (NOT A and NOT B) as well as logical operations of AND and OR also PASS function, (Output = A and Output = B). The PC (program Counter) has an 'I' control to increment it, the PC is a loadable counter.

4 of the 22 control bits are fed back to the address inputs of the CU ROM via a latch L to give up to 16 microinstructions per opcode. The CU ROM (Control Unit ROM) is 2^{13} x 22 bits in size (bits).

Other Control bits are labelled: -
ACC_S, MDR_S, $RESULT_S$, $RESULT_E$, IR_S, PC_S, PC_I, PC_E, MAR_S, MAR_E, ALU_{C0}, ALU_{C1}, ALU_{C2}, ROM_E, RAM_S, RAM_E, INP_E, OUT_S.

The Fetch-Execute table for "immediate add": -

Step	Action	Result	Comment
0	$PC_E = 1$	(PC) → AB	Put program counter contents onto address bus
1	$ROM_E = 1$	(ROM → DB	Read the program ROM. Opcode now on data bus
2	$PC_I = 1$, $PC_E = 1$, $IR_S = 1$	(PC)+1 → PC, (PC) → AB, (DB) → IR	Point PC at operand, and read the ROM: Its contents go into the IR
3	$ROM_E = 1$	(ROM) → DB	Address bus settles with new value; the address of the operand.
4	$MDR_S = 1$	(DB) → MDR	Put it in the MDR.
5	ALU = ADD	ALU = (ACC) + (MDR)	Execute the instruction.
6	$RESULT_S = 1$	(ALU) → RESULT	
7	$RESULT_E = 1$	(RESULT) → DB	Put answer onto data bus.
8	$ACC_S = 1$	(DB) → ACC	And into ACC.

9	$PC_I = 1$	These three lines are essential “housekeeping” and must occur at the end of every micro program to allow the “fetch” of the next opcode
10	$PC_E = 1$, $ROM_E = 1$	
11	$IR_S = 1$, $PC_I = 1$, $PC_E = 1$	

Microcode ROM contents “Immediate Add” instruction, assuming an opcode of 0E: -

a12	a[11..4]	a[3..0]	L[3..0]	ACCS	MDRS	RESS	RESE	IRS	PCS	PCI	PCE	MARS	MARE	ALUC2	ALUC1	ALUC0	ROME	RAMS	RAME	INPUT	OUTS	Step
X	00	0000	0001	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
X	00	0001	0010	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1
X	00	0010	0000	0	0	0	0	1	0	1	1	0	0	0	0	0	1	0	0	0	0	2
X	0E	0000	0001	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	3
X	0E	0001	0010	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	4
X	0E	0010	0011	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	5
X	0E	0011	0100	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	6
X	0E	0100	0101	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7
X	0E	0101	0110	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8
X	0E	0110	0111	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	9
X	0E	0111	1000	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	10
X	0E	1000	0000	0	0	0	0	1	0	1	1	0	0	0	0	0	1	0	0	0	0	11
X	??	0000	0001	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	X
X	??	0001	0010	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	X

3.1. Additional Accumulators

The simple hypothetical computer only has one accumulator, clearly this is a drawback. Figure 3.1a, and figure 3.1b shows how the design could be modified to accommodate additional accumulators.

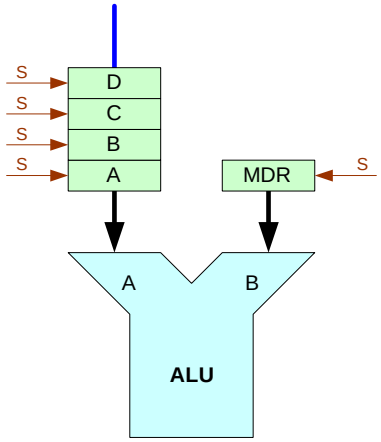


Figure 3.1a. Add register bank

or

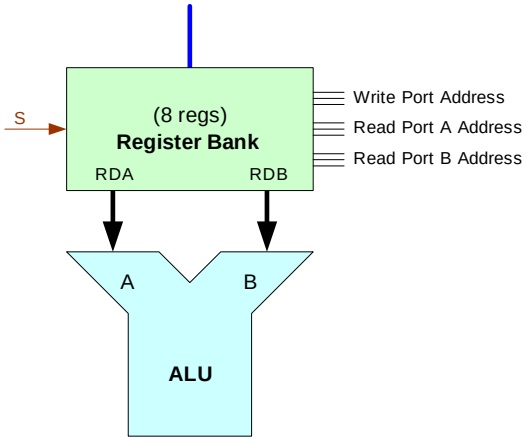


Figure 3.1b. Add register bank version 2

3.2. Stacks

Stacks (last in first out memory) are used to save the return address when a subroutine or interrupt routine is called. There are 2 common methods of implementing a stack: -

1. Keep a variable sized area of RAM free along with a stack pointer (SP).
2. Keep a second (& third) shadow PC and limit subroutine nesting to one or two.

3.3. Eight Bidirectional 8-bit Ports

The simple hypothetical computer only has one input and output port, clearly this is a drawback. Figure 3.3a shows how the design could be modified to accommodate additional input / output ports.

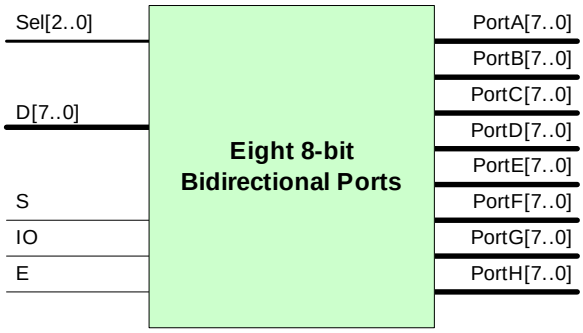


Figure 3.3a. Add a bank of 8-bit directional ports

3.4. 16-bit CPU

A 16-bit CPU design is more powerful, e.g. the data bus, address bus, ALU, registers, and latches are 16-bits.

4.0. CPU DESIGN

Several versions (e.g. mark 1, mark 2, etc…) of the CPU design have been included in this report; basically each new version is an improvement of the last. This ensures that operational simulations are achieved, as the more complex the design becomes the less likelihood of it simulating (e.g. design errors, does not fit on chip, simulation results difficult to understand).

4.1. Mark 1 – Simple CPU

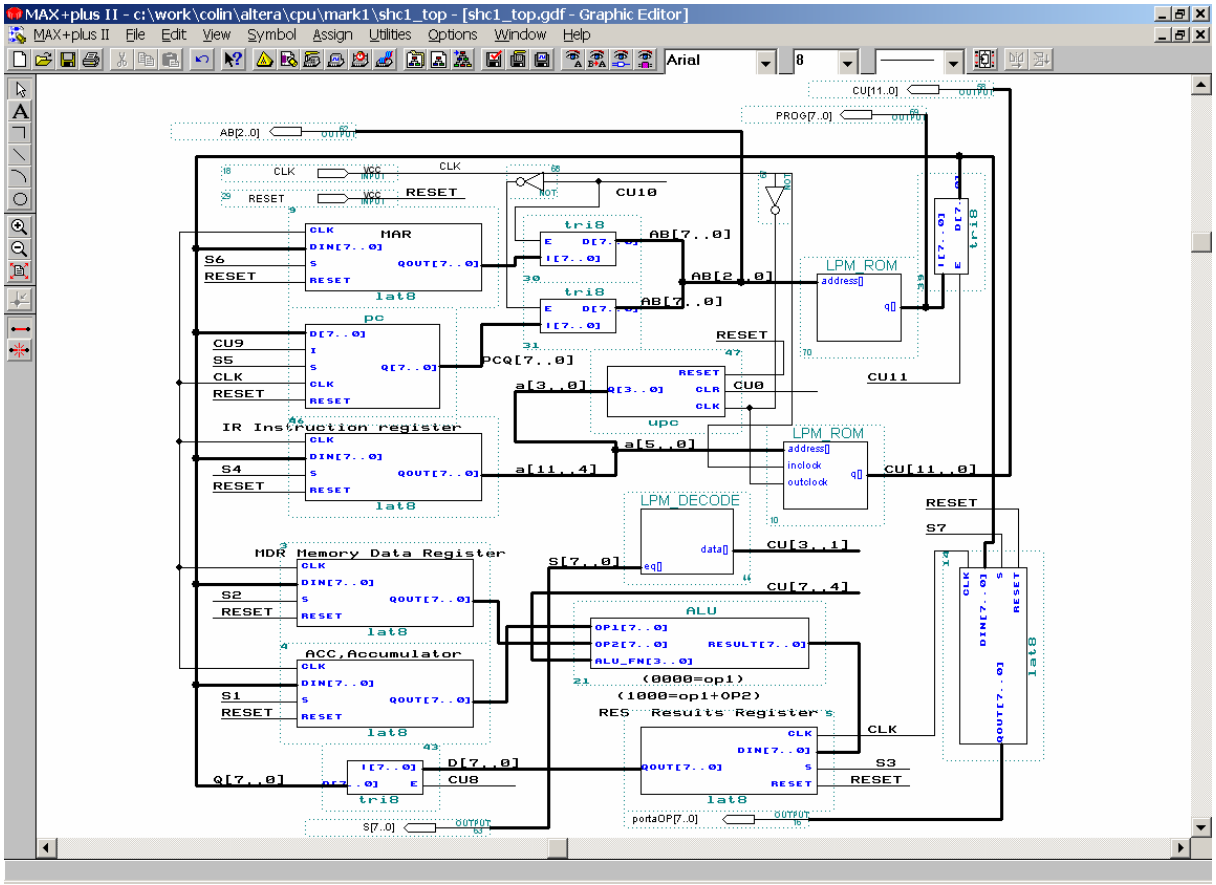


Figure 4.1a Screen dump of sch1_top.gdf

Notice the design has been partitioned into a number of functional blocks, each block does one task that can be easily described. This partitioned method of design makes the design and testing of large complex machines easy. Each functional block is tested using simulation, then when all blocks are working as designed the entree system is tested. Note that the design of a functional block can be made-up by other functional blocks, which in turn may have blocks within them; this cycle continues until there are no more functional blocks.

The top layout (figure 4.1a) is a little cluttered, and it makes sense to improve the situation in future versions. This could be achieved by splitting the design into four main functional blocks (ALU, memory, control unit and input/output), simplifying the top layout.

This design is based on the “Simple Hypothetical Computer” (chapter 3). In order to make simulation easier the control unit ROM and program ROM have been made much smaller than practical, but of a sufficient size to demonstrate a simple program. By restricting the program space to 8 locations, the address bus was reduced to 3 bits. By only supporting 3 opcodes it was possible to make the output of the Instruction register (IR) 2 bits wide, hence this reduces the microcode ROM to a more manageable 64 locations.

Notice functional block ‘lat8’ is used 6 times (MAR, IR, MDR, ACC, RES, portaOP), ‘tri8’ is used 4 times, and LP_ROM is used twice (microcode, program memory). Notice an ‘LPM_DECODE’ block is used to generate the S signals for the various registers; this saves a few microcode outputs.

A LPM_ROM block is used to store the program; functional block ‘pc’ points to the program memory locations. Only the MAR or ‘pc’ can change the program address, only one is enabled at anyone time as the output of each are sent into tri state buzzers (tri8) which are enabled/disable using CU10. CU10 is fed directly into the MARE but NOT CU10 is fed into PCE, making sure that only one is enabled at any one time. PC is incremented by the control unit using CU9 to move to the next instruction, while the MAR is used for direct or indirect addressing modes.

Functional block ‘upc’ is the microcode program counter, e.g. each opcode instruction could have up to 16 lines of microcode (upc 4-bits = 16). The ‘upc’ specifies the microcode location of an instruction, which is incremented using CU0. A 12-bit bit wide LPM_ROM is used for the control unit (microcode) note this is a non-standard ROM size, two 8-bit ROMs instead may make life easier for the auto router.

The ‘ALU’ block carries out the arithmetic and logic of the CPU (e.g. A+B, not A, A AND B), note there is a drawback, there is only one accumulator, clearly multiple accumulators could be added by adding a register bank to the input of the ALU.

The design has one output port, clearly there is no reason why more ports cannot be added except for increasing the number of S lines required from the control unit.

4.1.1. upc.gdf (4-bit Synchronous Counter)

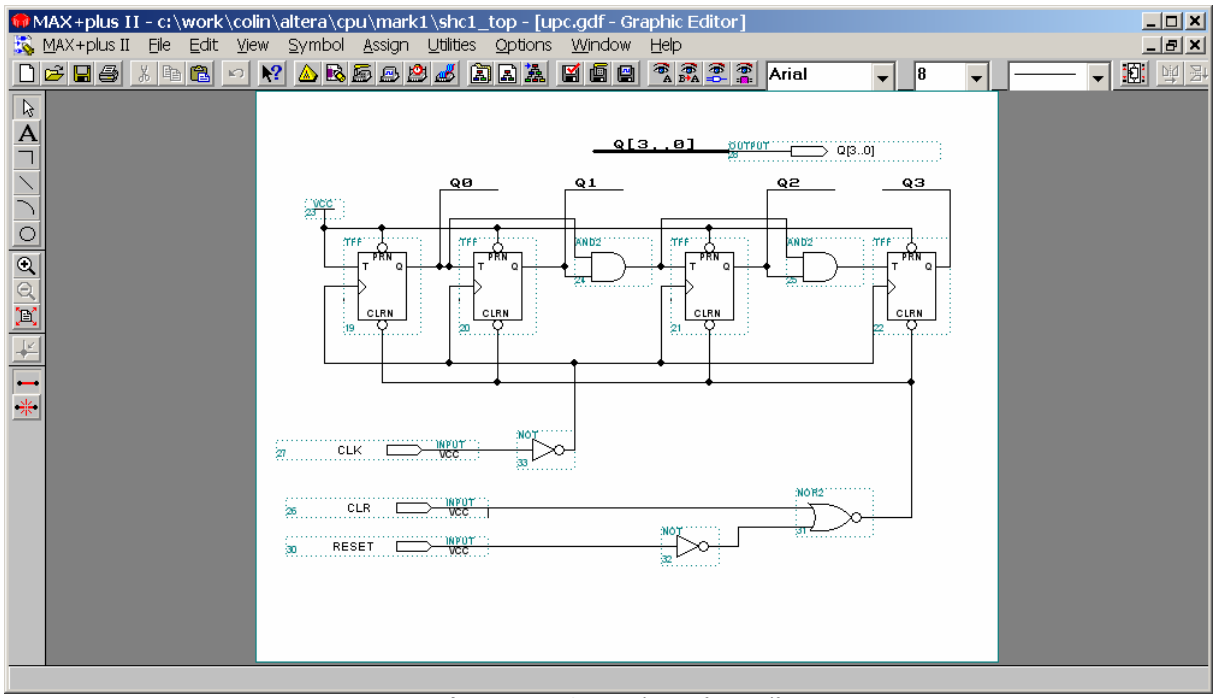


Figure 4.1.1a Screen dump of upc.gdf

Basically a simple four bit counter that is synchronised with the clock, which is reset when input CLR goes high or input RESET goes low.

Ripple counters have a problem due to the accumulated propagation delay of the clock from flip-flop to flip-flop. Synchronous counters eliminate that problem, because all the clock inputs are tied to a common clock, so each flip-flop will be triggered at the same time (thus any Q output transitions will occur at the same time).

To design a 4-bit synchronous counter four flip-flops (T type in this case) are required. Keep in mind that, because all the clock inputs receive a trigger at the same time, certain flip-flops must be stopped from making output transitions until it is their turn.

From figure 4.1.1a it is clear that the same clock input is driving all four flip-flops. The Q1 flip-flop will be in the hold mode until Q0 goes HIGH, which will force T_1 HIGH, allowing the Q1 flip-flop to toggle when the next clock edge comes in.

Flip-flop Q2 cannot toggle until Q0 and Q1 are both HIGH. Flip-flop Q3 cannot toggle until Q0, Q1 and Q2 are all HIGH.

RESET pin is connected to the CLRN of all the flip-flops, and VCC is connected to the PRN of all the flip-flops. If the count reaches 1111 the counter returns to zero on the next clock plus and starts the counting all over again.

Clearly the circuit is more complicated then a standard ripple counter, but the cumulative effect of propagation delays through the flip-flops is not a problem because all output transitions will occur at the same time. There is a propagation delay through the AND gates, but it will not affect the Q outputs of the flip-flops.

4.1.2. lat8.gdf (8-bit Latch)

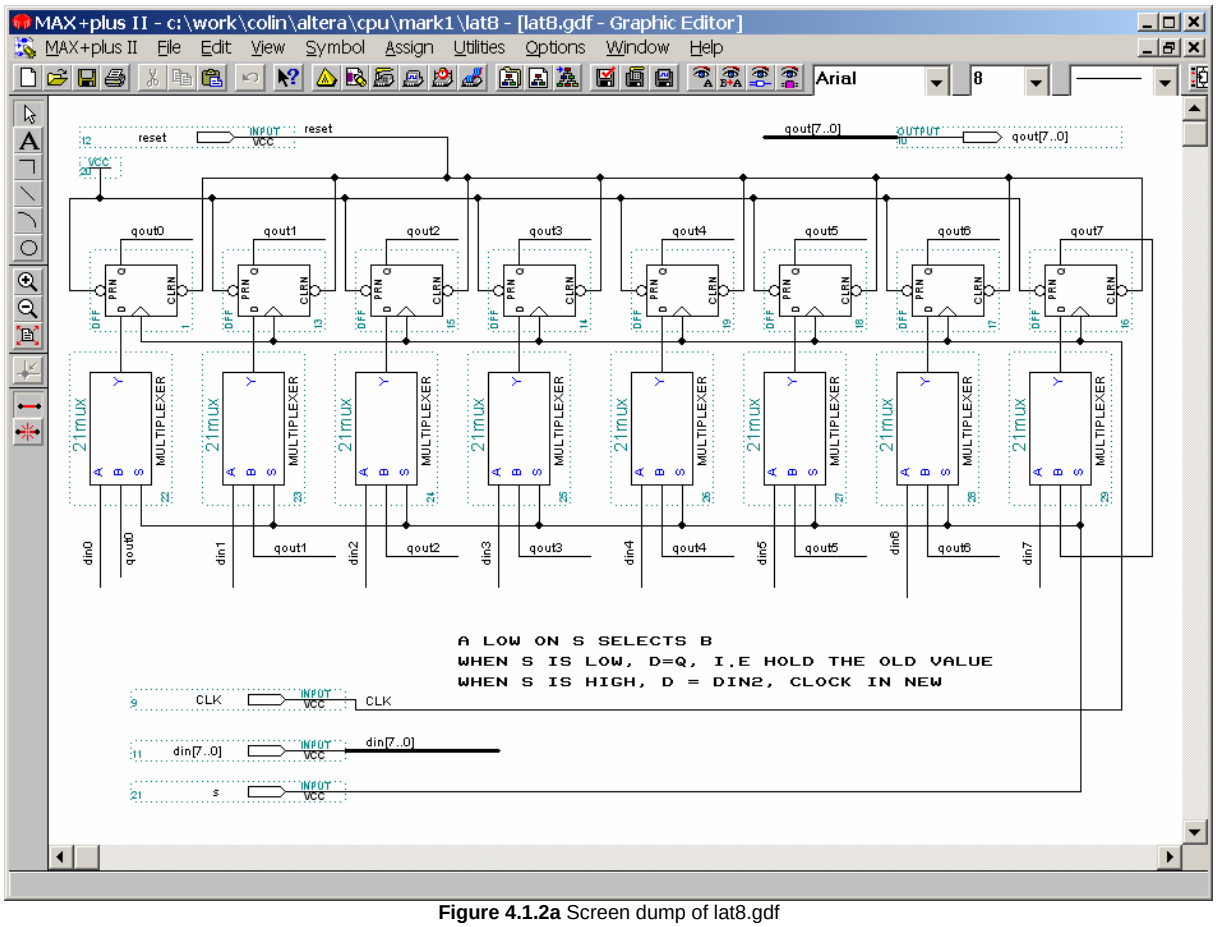


Figure 4.1.2a Screen dump of lat8.gdf

8 d-type flip-flops are use to store 8-bits of data, a 2-to-1 line multiplexer is connected to the input of each flip-flop. Input A of each multiplexer is connected to one of the din [7..0] lines, hence when S is high the new data is stored into the latch. Input B of each multiplexer is connected to the output (Q) of the flip-flops hence when S is low the latch will hold the old value (e.g. old value looped back into the latch).

4.1.3. tri8.gdf (8-bit Tri-State Buffer)

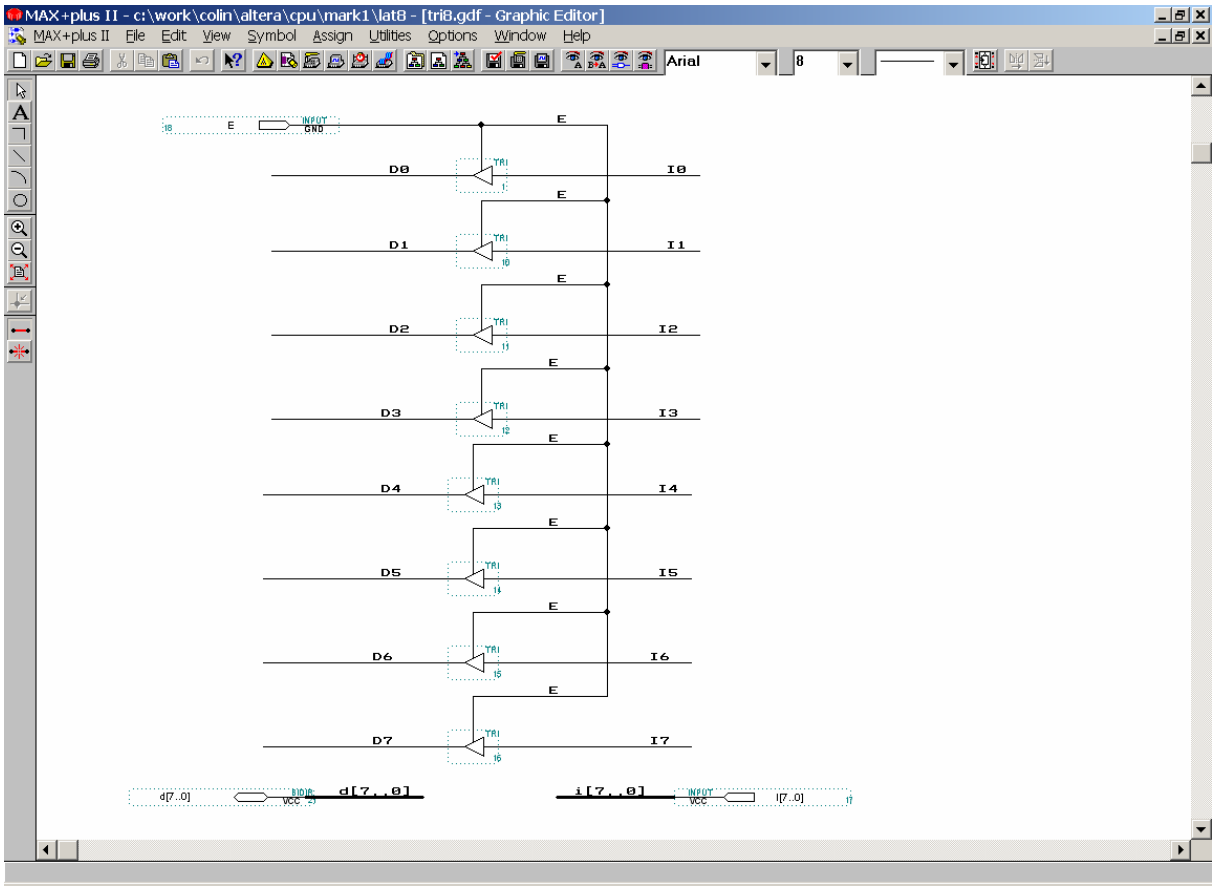


Figure 4.1.3a Screen dump of tri8.gdf

This functional block is extremely simple, basically 8 tri-state buffers are used to enable or disable a 8-bit bus. When E is high (enabled) d[7..0] equals i[7..0] and when E is low (disabled) d[7..0] is Z (high impedance). Obviously this functional block is useful and used many times throughout the CPU, as the CPU has a shared data bus, and there must be some way of disconnecting devices from the data bus because only one device can use the data bus at any one time.

4.1.4. pc.gdf (Program Counter)

Figure 4.1.4a shows the circuit diagram of this block, a functional block called “t_cell” is used. When input ‘I’ is high the “t_cell” blocks function like T type flip-flops, hence the T input of each block is connected as a T-type flip-flop would be connected to create an 8-bit binary up counter, as was done for upd.gdf (4-bit counter).

When input ‘S’ is high the “t_cell” functional block acts like a 1-bit latch, where the d input is latched into the flip-flop.

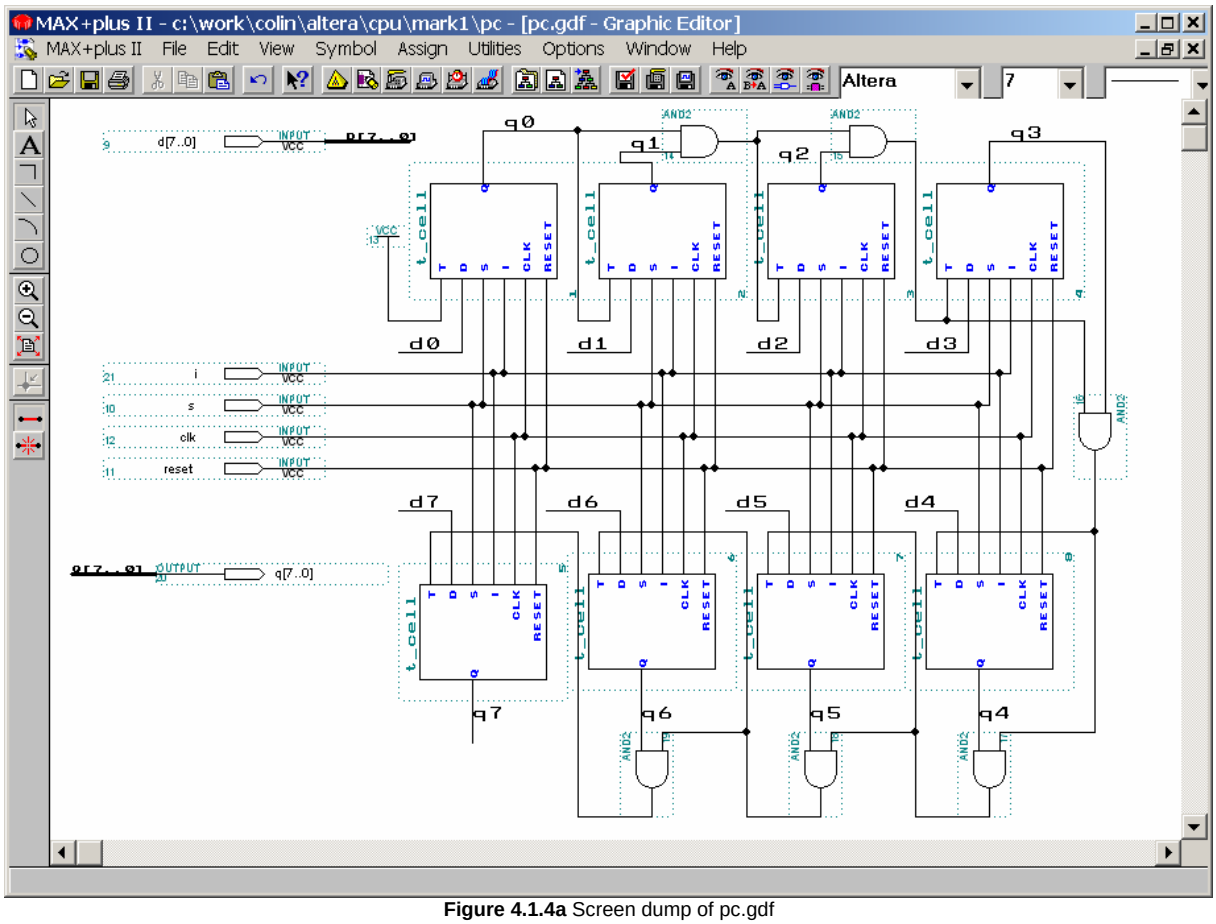


Figure 4.1.4a Screen dump of pc.gdf

4.1.5. t_cell.gdf (T-type Flip-Flop and Latch)

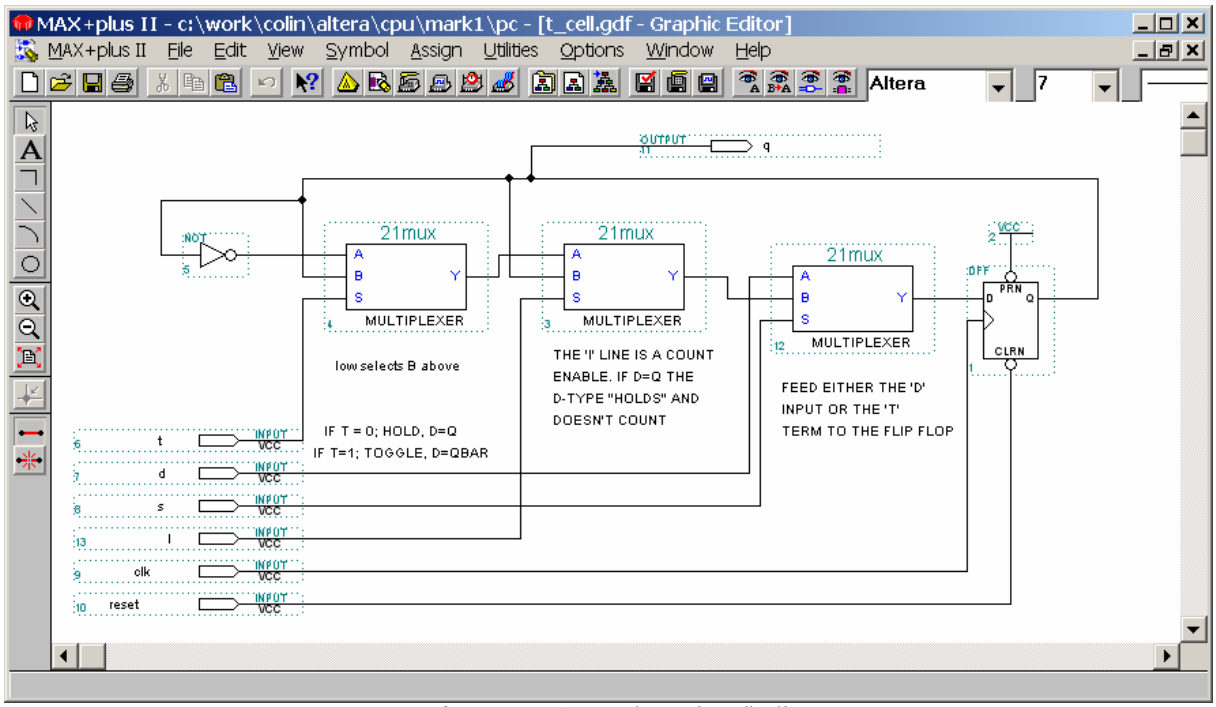


Figure 4.1.5a Screen dump of t_cell.gdf

3 two input multiplexer are used: The first multiplexer will hold if T = 0 (y=q), and toggle if T =1 (y=/Q), the middle multiplexer enables or disables the count, and the right most multiplexer feeds either the 'D' input or the 'T' term to the d-type flip-flop.

4.1.6. alu.vhd (Arithmetic and Logic Unit)

The ALU can carry out and one of 14 mathematical operations, input line alu_fn[3..0] is used to specify which ALU function to use. There are two 8-bit input lines op1[7..0] and op2[7..0] which are used in the mathematical calculation of the result[7..0].

```
-------------------------------
-- Arithmetic and Logic Unit --
-- By:  Ian McCrum            --
-- Modified by: Colin McCord --
-- Date: 20/04/2002          --
-- Version: 1.1              --
-------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY alu IS
       PORT (op1, op2         : IN  std_logic_vector(7 downto 0);
             alu_fn           : IN  std_logic_vector(3 downto 0);
             result           : OUT std_logic_vector(7 downto 0));
END alu;


ARCHITECTURE a OF alu IS
BEGIN
PROCESS(alu_fn)
 BEGIN
  CASE alu_fn IS
    WHEN "0000" => result <= op1;
    WHEN "0001" => result <= op2;
    WHEN "0010" => result <= not op1;
    WHEN "0011" => result <= not op2;

    WHEN "0100" => result <= op1 and op2;
    WHEN "0101" => result <= op1 or  op2;
    WHEN "0110" => result <= op1 xor op2;

    WHEN "0111" => result <= op1 + "00000001";
    WHEN "1000" => result <= op1 + op2 ;
    WHEN "1001" => result <= op1 + op2 + "00000001";

    WHEN "1010" => result <= op1 - op2;
    WHEN "1011" => result <= op2 - op1;

    WHEN "1100" => result <= op1 - "00000001";
    WHEN "1101" => result <= op2 - "00000001";
    WHEN OTHERS => result <= op1;
  END CASE;
 END PROCESS;
END a;
```

alu_fn[3..0]	result[7..0]	alu_fn[3..0]	result[7..0]
0 (0000)	op1	7 (0111)	op1 + 1
1 (0001)	op2	8 (1000)	op1 + op2
2 (0010)	not op1	9 (1001)	op1 + op2 + 1
3 (0011)	not op2	10 (1010)	op1 – op2
4 (0100)	op1 and op2	11 (1011)	op2 – op1
5 (0101)	op1 or op2	12 (1100)	op1 - 1
6 (0110)	op1 xor op2	13 (1101)	op2 - 1

Good selection of operations perhaps functions like divide and multiply could be added using lpm_divide and lpm_multiply.

4.1.7. Microcode

The instruction register has been limited (two address lines); hence this CPU has a maximum of three instructions. Clearly this is not ideal and this needs to be improved.

First initialised the system on power up: -

Address Radix		Data Radix							
5..4	3..0	11	10	9	8	7..4	3..1	0	
OP ADDRESS	UPC	ROME	PCE	PCI	RESE	SEL ALU FUN	SEL S	UPCCL	Comments
00	0000	1	0	0	0	0000	000	0	Enable (PC) → AB & Enable ROM (early)
00	0001	1	0	0	0	0000	000	0	Enable ROM in earnest, (ROM)→DB
00	0010	1	0	1	0	0000	100	1	Strobe IR and Clear UPC. Jump to New

PCE is active low

S1: ACCS = 001, S2: MDRS = 010, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101, S6: MARS = 110
S7: OUTS = 111, leave at 00 for unused

ALU functions, various…e.g.
0000 = pass OP1, 1000 = PO1 + OP2

Add immediate opcode (address 03): -

Address Radix		Data Radix							
5..4	3..0	11	10	9	8	7..4	3..1	0	
OP ADDRESS	UPC	ROME	PCE	PCI	RESE	SEL ALU FUN	SEL S	UPCCL	Comments
11	0000	0	0	0	0	0000	000	0	Enable (PC) → AB & Enable ROM (early)
11	0001	1	0	0	0	0000	000	0	Enable ROM in earnest, (ROM) → DB
11	0010	1	0	1	0	0000	010	0	Strobe MDR got second byte, also inc PC
11	0011	0	0	0	0	1000	000	0	Add OP1 and OP2
11	0100	0	0	0	0	1000	011	0	Strobe the result register, got answer
11	0101	0	0	0	1	0000	000	0	RESE, answer → DB
11	0110	0	0	0	1	0000	001	0	ACCS, Got answer in the accumulator
11	0111	0	0	0	0	0000	000	0	Enable (PC) → AB & Enable ROM
11	1000	1	0	0	0	0000	000	0	Enable ROM In Earnest, (ROM) → DB
11	1001	1	0	1	0	0000	100	1	Strobe IR and clear UPC. Jump to new

PCE is active low

S1: ACCS = 001, S2: MDRS = 010, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101, S6: MARS = 110
S7: OUTS = 111, leave at 00 for unused

ALU functions, various…e.g.
0000 = pass OP1, 1000 = OP1 + OP2

Out A opcode (address 02): -

Address Radix		Data Radix							
5..4	3..0	11	10	9	8	7..4	3..1	0	
OP ADDRESS	UPC	ROME	PCE	PCI	RESE	SEL ALU FUN	SEL S	UPCCL	Comments
10	0000	0	0	0	0	0000	011	0	ALU is passing OP1, Inc RES
10	0001	0	0	0	1	0000	000	0	Enable results → DB
10	0010	0	0	0	1	0000	111	0	Keep enabled, then strobe output port
10	0011	0	0	0	0	0000	000	0	Enable (PC) → AB & Enable ROM
10	0100	1	0	0	0	0000	000	0	Enable ROM In Earnest, (ROM) → DB
10	0101	1	0	1	0	0000	100	1	Strobe IR and clear UPC. Jump to new

PCE is active low

S1: ACCS = 001, S2: MDRS = 010, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101, S6: MARS = 110
S7: OUTS = 111, leave at 00 for unused

ALU functions, various…e.g.
0000 = pass OP1, 1000 = OP1 + OP2

Jump Immediate opcode (address 01): -

PCE is active low

S1: ACCS = 001, S2: MDRS = 010, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101, S6: MARS = 110
S7: OUTS = 111, leave at 00 for unused

ALU functions, various…e.g.
0000 = pass OP1, 1000 = OP1 + OP2

Address Radix		Data Radix							
5..4	3..0	11	10	9	8	7..4	3..1	0	
OP ADDRESS	UPC	ROME	PCE	PCI	RESE	SEL ALU FUN	SEL S	UPCCL	Comments
01	0000	0	0	0	0	0000	000	0	Enable PC, get next byte
01	0001	1	0	0	0	0000	000	0	ROM, second byte → DB
01	0010	1	0	0	0	0000	101	0	PC, new value in PC
01	0011	0	0	0	0	0000	000	0	Enable (PC) → AB & Enable ROM
01	0100	1	0	0	0	0000	000	0	Enable ROM In Earnest, (ROM) → DB
01	0101	1	0	1	0	0000	100	1	Strobe IR and clear UPC. Jump to new

This version of the CPU is limited to three opcodes; hence to add additional functions another address line is required from the IR. For example say there were 3 OP address lines, the following opcodes could be added.

Subtract immediate opcode (address 04): -

PCE is active low

S1: ACCS = 001, S2: MDRS = 010, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101, S6: MARS = 110
S7: OUTS = 111, leave at 00 for unused

ALU functions, various…e.g.
0000 = pass OP1, 1011 = OP1 - OP2

Address Radix		Data Radix							
5..4	3..0	11	10	9	8	7..4	3..1	0	
OP ADDRESS	UPC	ROME	PCE	PCI	RESE	SEL ALU FUN	SEL S	UPCCL	Comments
100	0000	0	0	0	0	0000	000	0	Enable (PC) → AB & Enable ROM (early)
100	0001	1	0	0	0	0000	000	0	Enable ROM in earnest, (ROM) → DB
100	0010	1	0	1	0	0000	010	0	Strobe MDR got second byte, also inc PC
100	0011	0	0	0	0	1011	000	0	Subtract OP1 and OP2
100	0100	0	0	0	0	1011	011	0	Strobe the result register, got answer
100	0101	0	0	0	1	0000	000	0	RESE, answer → DB
100	0110	0	0	0	1	0000	001	0	ACCS, Got answer in the accumulator
100	0111	0	0	0	0	0000	000	0	Enable (PC) → AB & Enable ROM
100	1000	1	0	0	0	0000	000	0	Enable ROM In Earnest, (ROM) → DB
100	1001	1	0	1	0	0000	100	1	Strobe IR and clear UPC. Jump to new

LoadA ## opcode (address 05): -

PCE is active low

S1: ACCS = 001, S2: MDRS = 010, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101, S6: MARS = 110
S7: OUTS = 111, leave at 00 for unused

ALU functions, various…e.g.
0000 = pass OP1, 0001 = OP2

Address Radix		Data Radix							
5..4	3..0	11	10	9	8	7..4	3..1	0	
OP ADDRESS	UPC	ROME	PCE	PCI	RESE	SEL ALU FUN	SEL S	UPCCL	Comments
101	0000	0	0	0	0	0000	000	0	Enable (PC) → AB & Enable ROM (early)
101	0001	1	0	0	0	0000	000	0	Enable ROM in earnest, (ROM) → DB
101	0010	1	0	1	0	0000	010	0	Strobe MDR got second byte, also inc PC
101	0011	0	0	0	0	0001	000	0	OP2
101	0100	0	0	0	0	0001	011	0	Strobe the result register, got answer
101	0101	0	0	0	1	0000	000	0	RESE, answer → DB
101	0110	0	0	0	1	0000	001	0	ACCS, Got answer in the accumulator
101	0111	0	0	0	0	0000	000	0	Enable (PC) → AB & Enable ROM
101	1000	1	0	0	0	0000	000	0	Enable ROM In Earnest, (ROM) → DB
101	1001	1	0	1	0	0000	100	1	Strobe IR and clear UPC. Jump to new

4.2. Mark 2 – Improved CPU

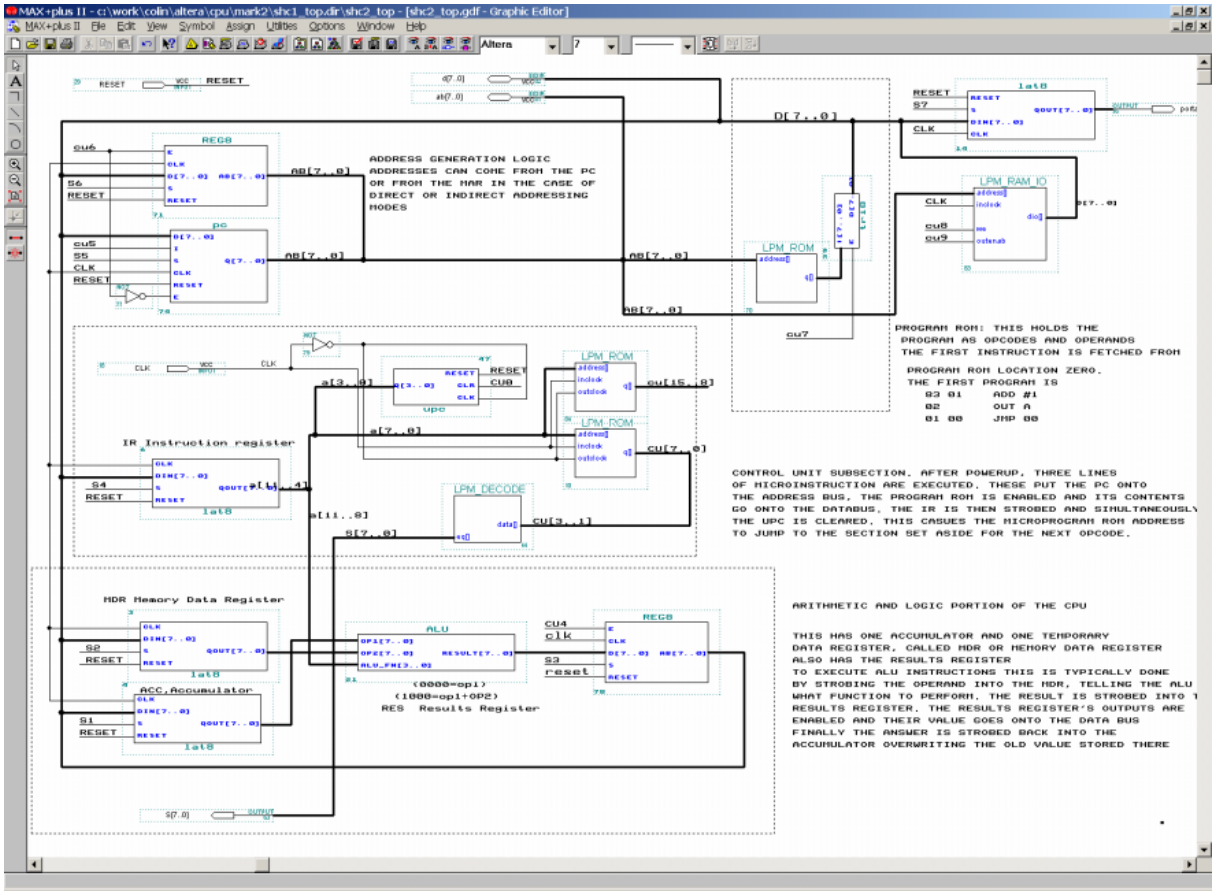


Figure 4.2a. Screen dump of sch2_top.gdf

This CPU is more powerful than mark1; it can support up to 15 opcodes, and has many unused control lines (CU15..CU10) for future expansion. Notice that the control unit (microcode) has now been split into two 8-bit ROMs, this is because the auto router finds it easier to fit 8-bit ROM than a non-standard sized ROM.

A RAM block has now been added, which uses two controls line (CU8 & CU9), the data bus and the address bus. This feature makes the CPU much more powerful as complex calculations can be carried out using the RAM for storage of intermediate results. Also a portion of the RAM is could be reserved for us as a software stack, for storing the program counter, when calling subroutines or interrupt routines.

Something very smart has been done with the ALU control lines, instead of the control unit controlling the ALU function, this is set in the program ROM, and fid directly into the ALU from the instruction register. Therefore for example the microcode for add immediate is exactly the same for any ALU function, this dramatically reduces the amount of microcode. Therefore this CPU design can actually support many more than 15 opcode instructions, as up to 16 instructions (16 ALU functions) could be achieved per microcode instruction.

For example the address of the immediate opcode is 03 (11), hence these opcodes can be used: -

0x13 = 0b00010011 (ALU function, opcode address) = LDA #
0x33 = 0b00110011 (ALU function, opcode address) = NOTA #
0x73 = 0b01110011 (ALU function, opcode address) = INCA
0x83 = 0b10000011 (ALU function, opcode address) = ADDA #
0xA3 = 0b10100011 (ALU function, opcode address) = SUBA #
0xC3 = 0b11000011 (ALU function, opcode address) = DECA
Note: This is just a summary, as every ALU function 0000 to 1111 can be used.

Notice the design uses all of the functional blocks as used for mark1, with the addition of one new block called 'REG8', this block is extremely simple basically it uses the 'lat8' block and the 'tri8' block (see figure 4.2b), this reduces clutter on the top layout (clearly top layer still too complex, still needs simplified further).

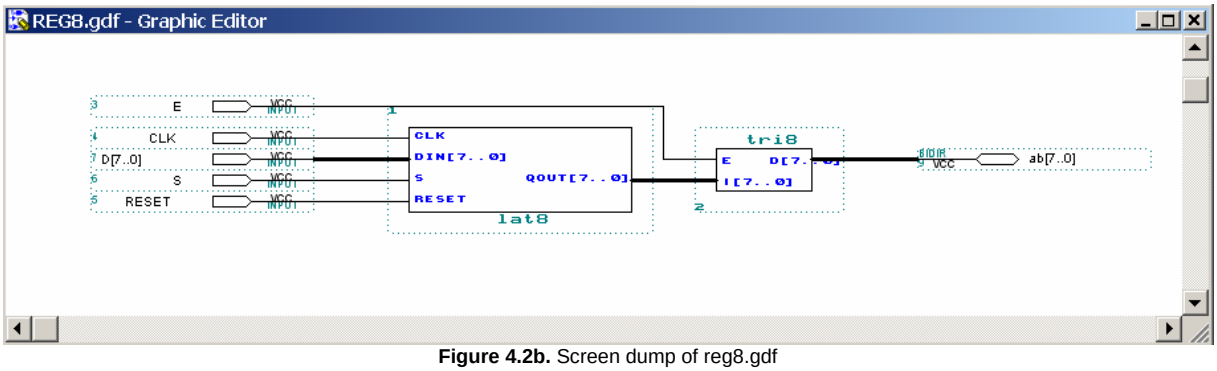


Figure 4.2b. Screen dump of reg8.gdf

4.2.1. Microcode

The instruction register has been limited (four address lines); hence this CPU can have a maximum of 16 instructions. But each instruction can implement up to 16 different ALU functions, hence providing up to 256 different opcodes (extremely powerful).

First initialised the system on power up: -

Address Radix		Data Radix						
7..4	3..0	7	6	5	4	3..1	0	
OP ADDRESS	UPC	ROME	PCE	PCI	RESE	SEL S	UPCCL	Comments
0000	0000	1	0	0	0	000	0	Enable (PC) → AB & Enable ROM (early)
0000	0001	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
0000	0010	1	0	1	0	100	1	Strobe IR and Clear UPC. Jump to New

PCE is active low

S1: ACCS = 001, S2: MDRS = 010, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101, S6: MARS = 110
S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

General immediate opcode (address #3): -

Address Radix		Data Radix						
5..4	3..0	7	6	5	4	3..1	0	
OP ADDRESS	UPC	ROME	PCE	PCI	RESE	SEL S	UPCCL	Comments
0011	0000	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM (early)
0011	0001	1	0	0	0	000	0	Enable ROM in earnest, (ROM) → DB
0011	0010	1	0	1	0	010	0	Strobe MDR got second byte, also inc PC
0011	0011	0	0	0	0	000	0	ALU operation (e.g. add OP1 and OP2)
0011	0100	0	0	0	0	011	0	Strobe the result register, got answer
0011	0101	0	0	0	1	000	0	RESE, answer → DB
0011	0110	0	0	0	1	001	0	ACCS, Got answer in the accumulator
0011	0111	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0011	1000	1	0	0	0	000	0	Enable ROM In Earnest, (ROM) → DB
0011	1001	1	0	1	0	100	1	Strobe IR and clear UPC. Jump to new

PCE is active low

S1: ACCS = 001, S2: MDRS = 010, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101, S6: MARS = 110
S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

General purpose output opcode: (address #2): -

Address Radix		Data Radix						
5..4	3..0	7	6	5	4	3..1	0	
OP ADDRESS	UPC	ROME	PCE	PCI	RESE	SEL S	UPCCL	Comments
0010	0000	0	0	0	0	011	0	ALU function (e.g. passing OP1, Inc RES)
0010	0001	0	0	0	1	000	0	Enable results → DB
0010	0010	0	0	0	1	111	0	Keep enabled, then strobe output port
0010	0011	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0010	0100	1	0	0	0	000	0	Enable ROM In Earnest, (ROM) → DB
0010	0101	1	0	1	0	100	1	Strobe IR and clear UPC. Jump to new

PCE is active low

S1: ACCS = 001, S2: MDRS = 010, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101, S6: MARS = 110
S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

Jump Immediate opcode (address 01): -

Address Radix		Data Radix						
5..4	3..0	7	6	5	4	3..1	0	
OP ADDRESS	UPC	ROME	PCE	PCI	RESE	SEL S	UPCCL	Comments
0001	0000	0	0	0	0	000	0	Enable PC, get next byte
0001	0001	1	0	0	0	000	0	ROM, second byte → DB
0001	0010	1	0	0	0	101	0	PC, new value in PC
0001	0011	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0001	0100	1	0	0	0	000	0	Enable ROM In Earnest, (ROM) → DB
0001	0101	1	0	1	0	100	1	Strobe IR and clear UPC. Jump to new

PCE is active low

S1: ACCS = 001, S2: MDRS = 010, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101, S6: MARS = 110
S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

4.2.2. Supported Opcodes

Machine Code	Opcode	Function
0x13 0x##	LDA #	ACC = #
0x33 0x##	NOTA #	ACC = /#
0x43 0x##	ANDA #	ACC = ACC and #
0x53 0x##	ORA #	ACC = ACC or #
0x63 0x##	XORA #	ACC = ACC xor #
0x73 0x##	INCA #	ACC = ACC + 1
0x83 0x##	ADDA #	ACC = ACC + #
0x93 0x##	ADDAI #	ACC = ACC + # + 1
0xA3 0x##	SUBA #	ACC = ACC - #
0xB3 0x##	SUBA #,A	ACC = # - ACC
0xC3 0x##	DECA	ACC = ACC -1
0xD3 0x##	DECA #,A	ACC = # - 1
0x02	OUTA, A	Output port A = ACC
0x12	OUTA, B	Output port A = MDR
0x22	OUTA, NOTA	Output port A = not ACC
0x32	OUTA, NOTB	Output port A = not MDR
0x42	OUTA, AandB	Output port A = ACC and MDR
0x52	OUTA, AorB	Output port A = ACC or MDR
0x62	OUTA, AxorB	Output port A = ACC xor MDR
0x72	OUTA, INCA	Output port A = ACC +1
0x82	OUTA, A+B	Output port A = ACC + MDR
0x92	OUTA, A+B+1	Output port A = ACC + MDR + 1

0xA2	OUTA, A-B	Output port A = ACC - MDR
0xB2	OUTA, B-A	Output port A = MDR - ACC
0xC2	OUTA, A-1	Output port A = ACC - 1
0xD2	OUTA, B-1	Output port B = MDR - 1
0x01 0x##	JMP ##	PC = ##

4.3. Mark 3 – Superior CPU

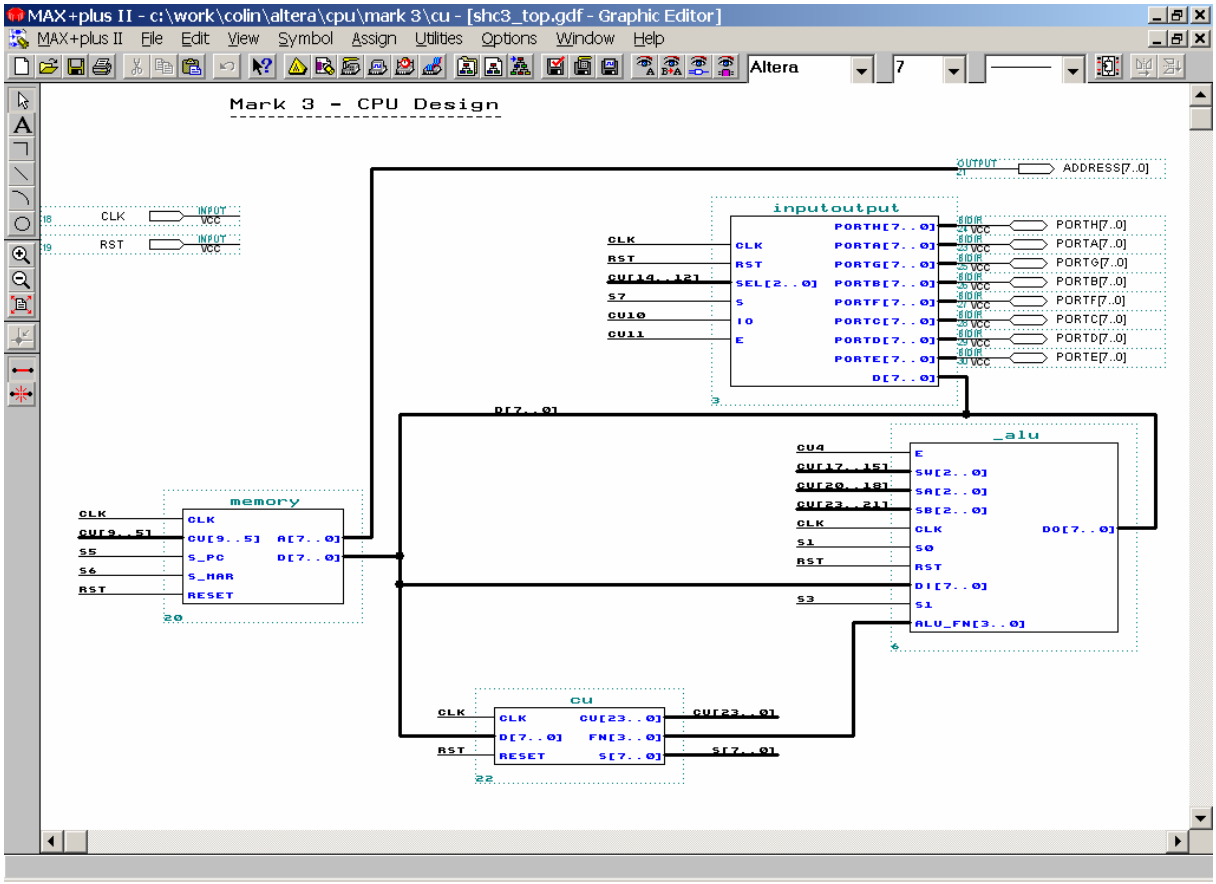


Figure 4.3a. Screen dump of sch3_top.gdf

The top layout has four functional blocks as suggested by 'Von Neumann' (memory, ALU, Control Unit and Input/Output). Clearly this simplifies the design dramatically; as it is clear 'straight away' how the CPU works, unlike the mark 1 and mark 2 were some time had to be spent tracing wires. This demonstrates the power of the partitioning method, as this CPU looks simpler than the mark 1 and mark 2, but it is actually much more complicated (e.g. if everything was on the top layout, it would be extremely difficult to understand how the CPU worked). Clearly even the most complex of designs can be simplified dramatically if partitioned into small enough blocks and this is the key to the design of extremely complex CPUs (different design teams work on different blocks).

Functional block 'inputoutput' contains eight 8-bit bidirectional input/output ports. CU[14..12] selects a port (e.g. 000 = PORTA, 111 = PORTH), s7 strobe data into the latch of the selected port, and CU10 specifies the direction of the select port (0 = Output, 1 = Input). Note unselected ports are automatically set as outputs so they don't interfere with the data bus.

Functional block 'memory' contains the program ROM, RAM, PC and MAR. Controls lines CU[9..5] are fed into this block, S5 strobes the program counter and S6 strobes the MAR. Functional block 'CU' is the control unit that controls the CPU, CU[23..0] are general purpose controls lines, S[7..0] are strobe lines, and FN[3..0] set ALU functions (directly from IR).

The '_ALU' block is powerful, recall that mark 1 and mark 2 had only one ACC (and a MDR), well this ALU has 8 accumulators any of which can be connected to the op1 or op2 lines of the ALU. CU4 is used to enable the result of an ALU operation to appear on the data bus. CU[17..15] specifies which accumulator to write to (e.g. 000 = ACC A, 111 = ACC H), CU[20..18] specifies which accumulator is connected to op1 of the ALU and CU[23..21] specifies which accumulator is connected to op2 of the ALU. S1 strobes the selected accumulator, S3 strobes the result register and FN[3..0] selects the ALU function.

4.3.1. inputoutput.gdf (Eight 8-bit Bidirectional Ports)

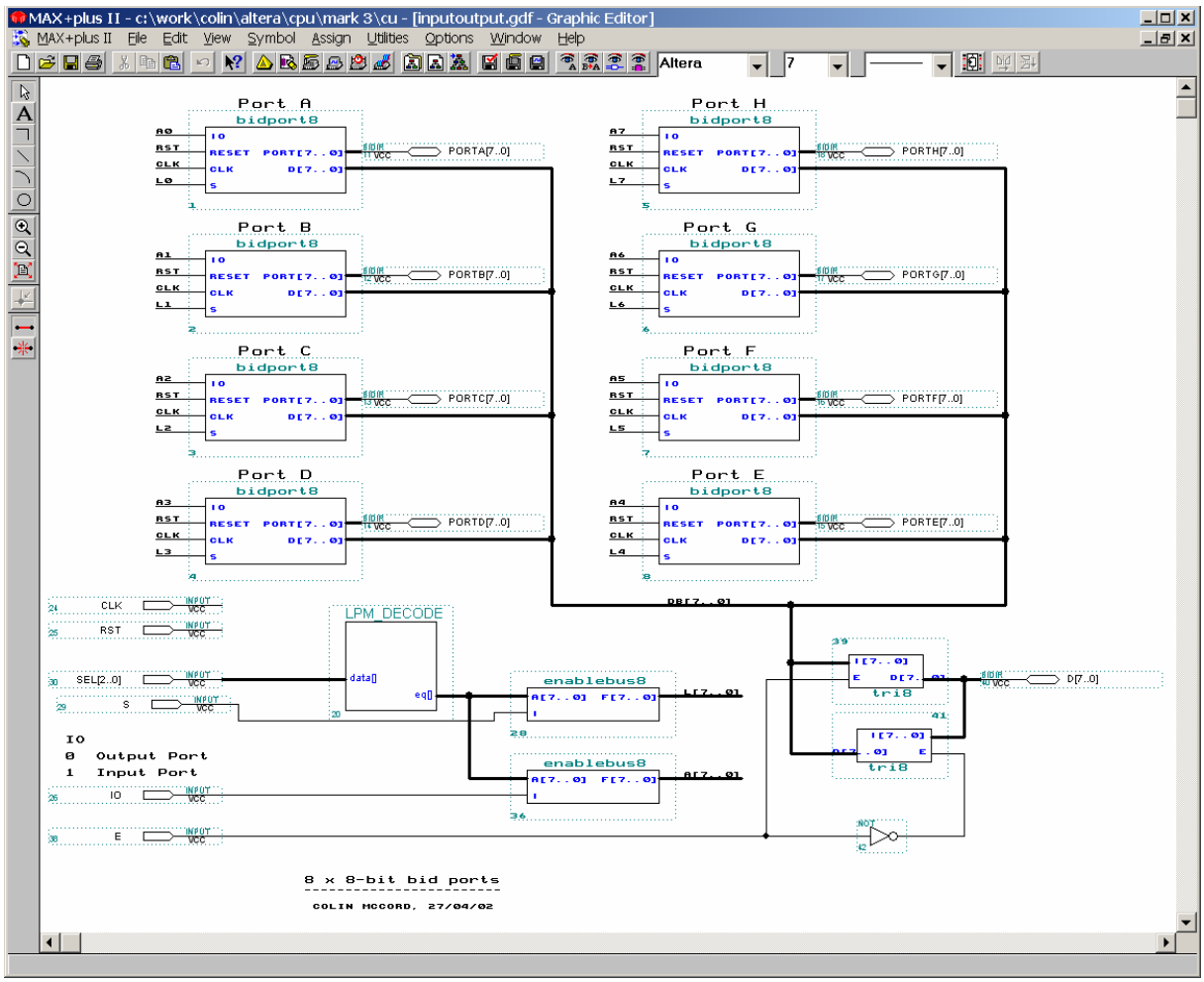


Figure 4.3.1a. Screen dump of inputoutput.gdf

There are two new functional blocks within this block, 'bidport8' and 'enablebus8'.

'bidport8' is used 8 times (one for each port); this block is basically an 8-bit latch that can be operated in two different modes. The first mode is the input mode; this means that the latch (on strobe) will get its new value from the port and put it onto the data bus. The second mode is the output mode; this means that the latch (on strobe) will get its new value from the data bus and put it onto the port.

Notice the select lines (SEL[2..0]) are sent through a decoder, the output of which is connected to two 'enablebus8' blocks. The functional block 'enablebus8' operation is extremely simple basically every bit in A[7..0] is ANDed with I, hence if I is low the output F[7..0] will be 0b00000000, else it will equal A[7..0]. The first 'enablebus8' block has S connected to it's I input, this means that L[7..0] which strobes the selected port, only operates when the S input is high. The second 'enablebus8' block has IO connected to it's I input, this means all ports are set to an output when IO is low, and the selected port is set as an input when IO is high (rest set as an output).

The E input is connected to a 'tri8' functional block and via a not gate another 'tri8 functional block. Basically when E equals '0' the data bus is set as an input, and when E equal '1' the data bus is set as an output. Note when the data bus is set as an input this means that the data bus (shared resource) can be used by other devices.

4.3.2. bidport8.gdf (8-bit Bidirectional Port)

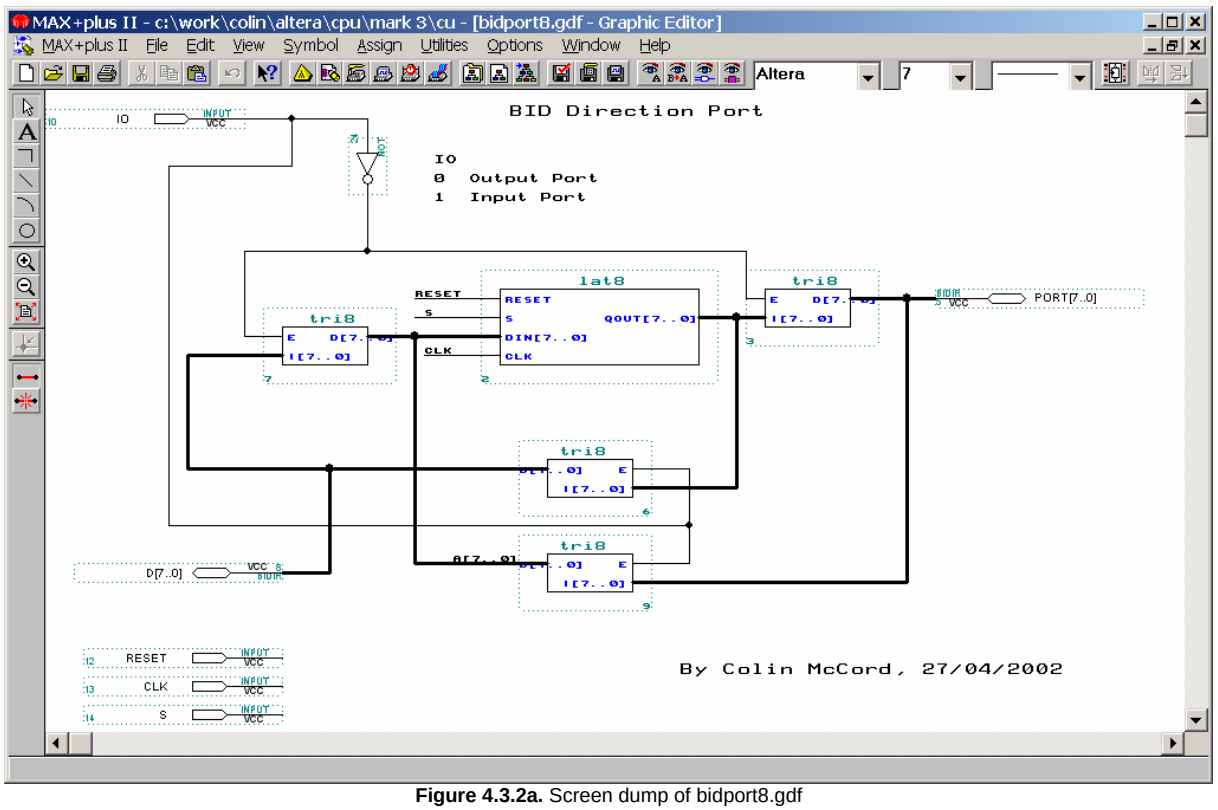


Figure 4.3.2a. Screen dump of bidport8.gdf

This block is simple; basically four 'tri8' blocks are used to set the direction of the port. When IO equals '0' the 'tri8' blocks on the left and right hand side of 'lat8' are enabled, hence the input of the latch is connected to the data bus and the output of the latch is connected to port. When IO equals '1' the 'tri8' blocks directly below 'lat8' are enabled, hence the input of the latch is connected to port and the output of the latch is connected to the data bus.

4.3.3. enablebus8.gdf (Enable 8-bit Bus)

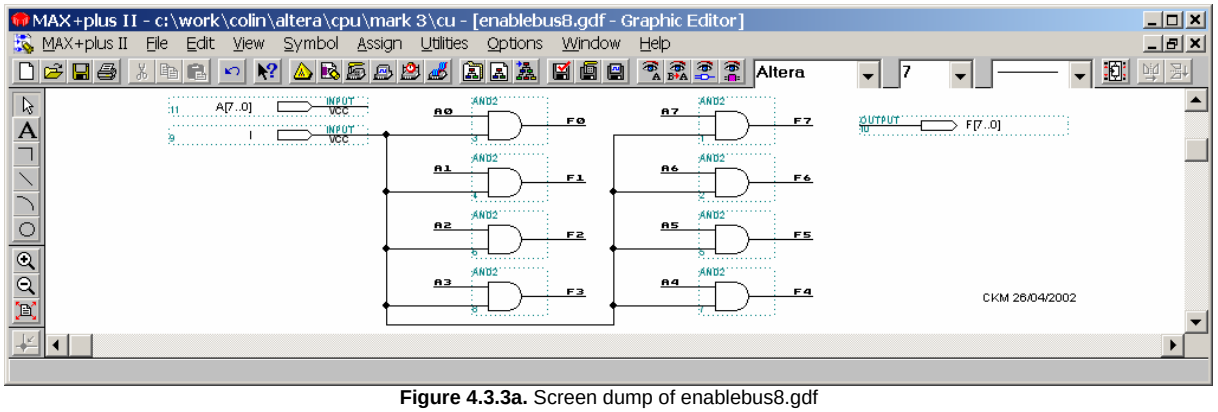


Figure 4.3.3a. Screen dump of enablebus8.gdf

This block is extremely simple, basically input I is ANDed with every bit in A[7..0], hence if I equals ‘0’ the output F[7..0] equals zero (bus disabled) and when I equals ‘1’ the output F[7..0] equals the input A[7..0].

4.3.4. _alu.gdf (Arithmetic and Logic Portion of CPU)

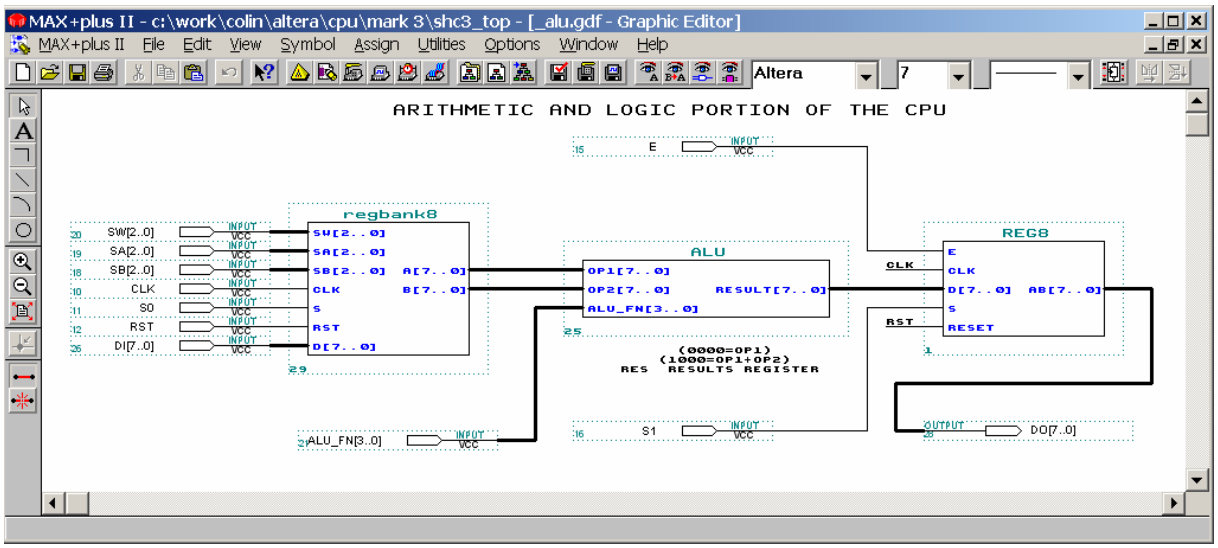


Figure 4.3.4a. Screen dump of _alu.gdf.gdf

The arithmetic and logic portion of the CPU is the same as mark 1 and mark 2 except that the input of the ALU (op1 and op2) is determined by a new functional block called ‘regbank8’. This functional block contains 8 accumulators, where SW[2..0] selects a accumulator to write to (on strobe), SA[2..0] selects a accumulator for OP1, SB[2..0] selects a accumulator for OP2 and S0 strobes the selected accumulator.

4.3.5. regbank8.gdf (Register Bank of 8 Accumulators)

See figure 4.3.5a for a screen dump of the circuit.

A new functional block ‘ACC’ is used 8-times (one of each accumulator), basically this new functional block contains an 8-bit latch which is connected to output A[7..0] when input ENABLE_A is 1 and connected to output B[7..0] when input ENABLE_B is 1.

Notice the write select lines (SW[2..0]) are connected to a decoder which is fed into the functional block ‘enablebus8’. The I input of the ‘enablebus8’ block is connected to input S, hence W[7..0] will strobe the selected accumulator when the S input is high, and all accumulators will hold there old value when the S input is low.

Notice the OP2 select lines (SB[2..0]) are connected to a decoder which (EB[7..0]) is connected to the ENABLE_B input of each accumulator. Hence the selected accumulator is connected to B[7..0], the rest are not.

Notice the OP1 select lines (SA[2..0]) are connected to a decoder which (EA[7..0]) is connected to the ENABLE_A input of each accumulator. Hence the selected accumulator is connected to A[7..0], the rest are not.

All accumulators share the data bus.

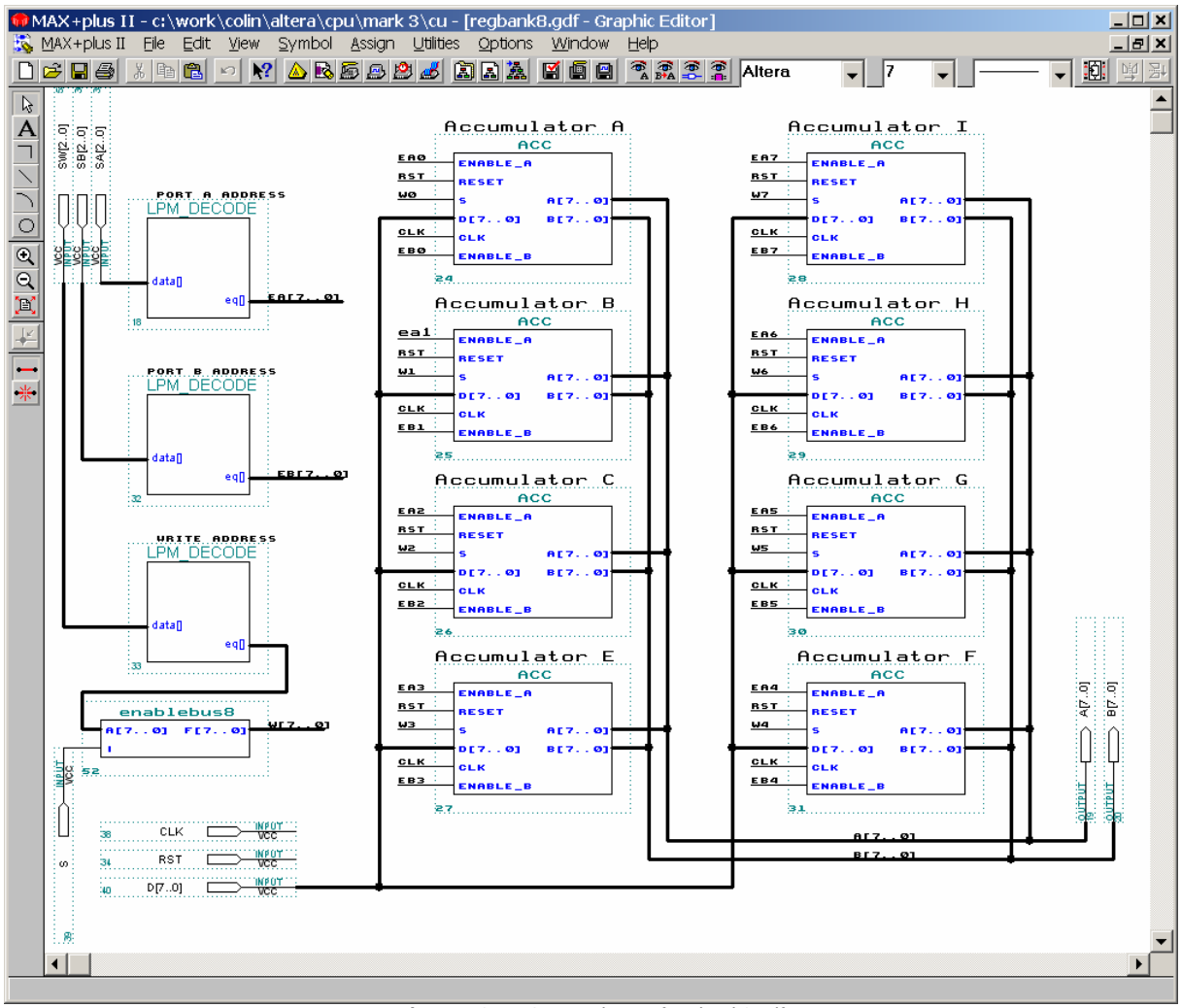


Figure 4.3.5a. Screen dump of regbank8.gdf

4.3.6. acc.gdf (8-bit Accumulator)

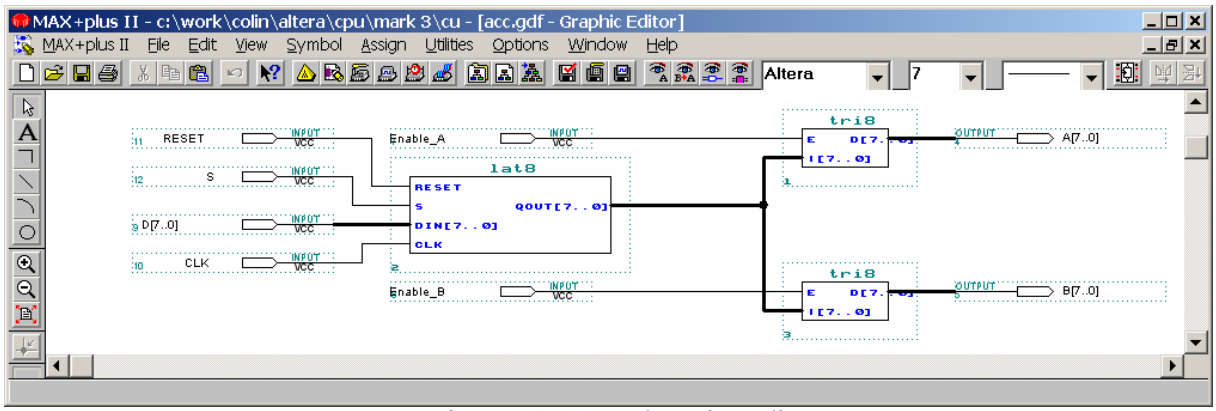


Figure 4.3.6a. Screen dump of acc.gdf

Very simple; basically Enable_A is connected to the E input of a 'tri8' block, which connects the output of 'lat8' to A[7..0] when Enable_A is a logic '1', else A[7..0] is high impedance. Enable B is connect to the E input of another 'tri8' block, which connects the output of 'lat8' to B[7..0] when Enable_B is a logic '1', else B[7..0] is high impendence. Notice that it is possible for the output of the latch to be connected to both A & B.

4.3.7. cu.gdf (Control Unit)

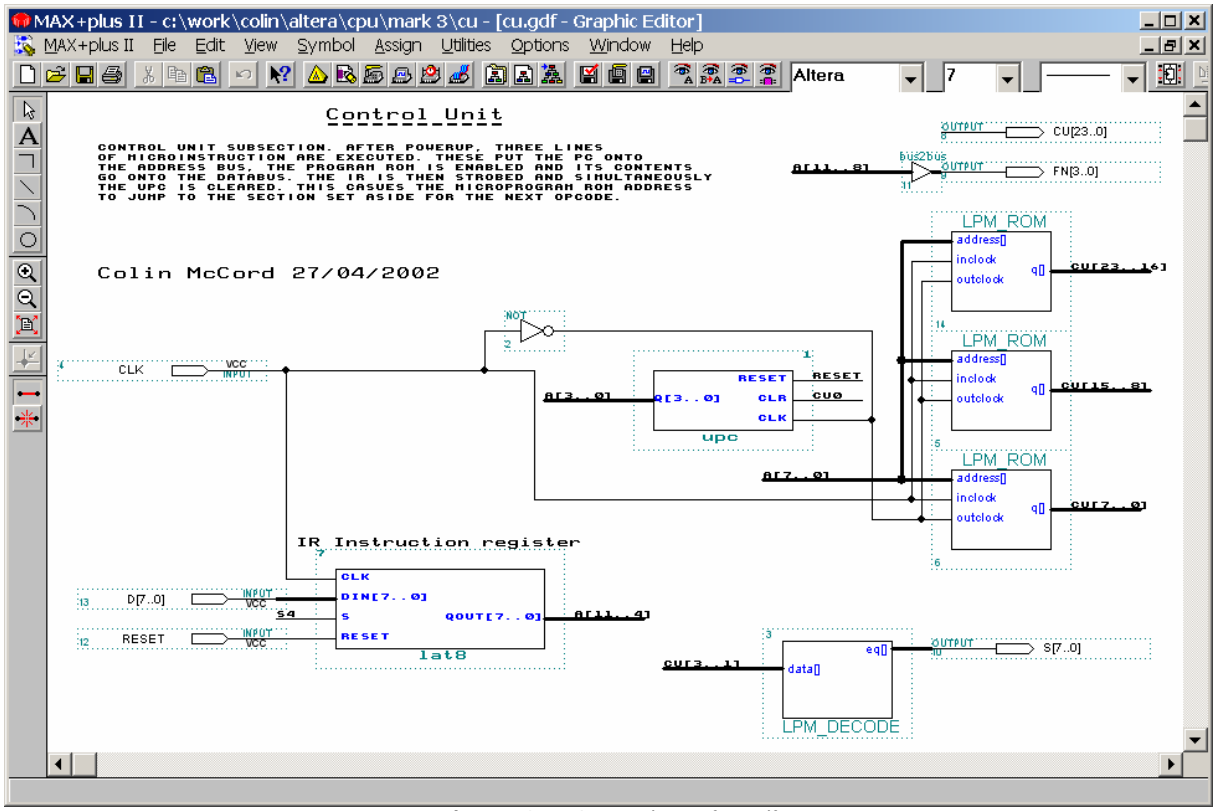


Figure 4.3.7a. Screen dump of cu.gdf

Same as mark 2 except for the additional ROM block. Basically this CPU has multipliable input/output ports and multipliable accumulators additional control lines are required to control these devices. Hence the reason why an additional microcode ROM block is required.

Note that each ROM block must have it own set of microcode. For example the microcode for all three ROMs use the same addresses, but configure different devices; hence each one gives different simultaneous outputs controlling their connected devices. Basically the microcode can be written as before with 23 columns, but once its time to write the text files for each of the ROM blocks, the microcode is spilt into three columns of 8 (one for each block) with the address duplicated in all three blocks.

4.3.8. memory.gdf (Memory / Program ROM)

Screen dump of the circuit is shown in figure 4.3.8a.

This block contains the program counter, MAR, program ROM and RAM as used for mark 2, except that in the mark 2 these items were on the top layout. Putting them into this function block simplifies the top layout.

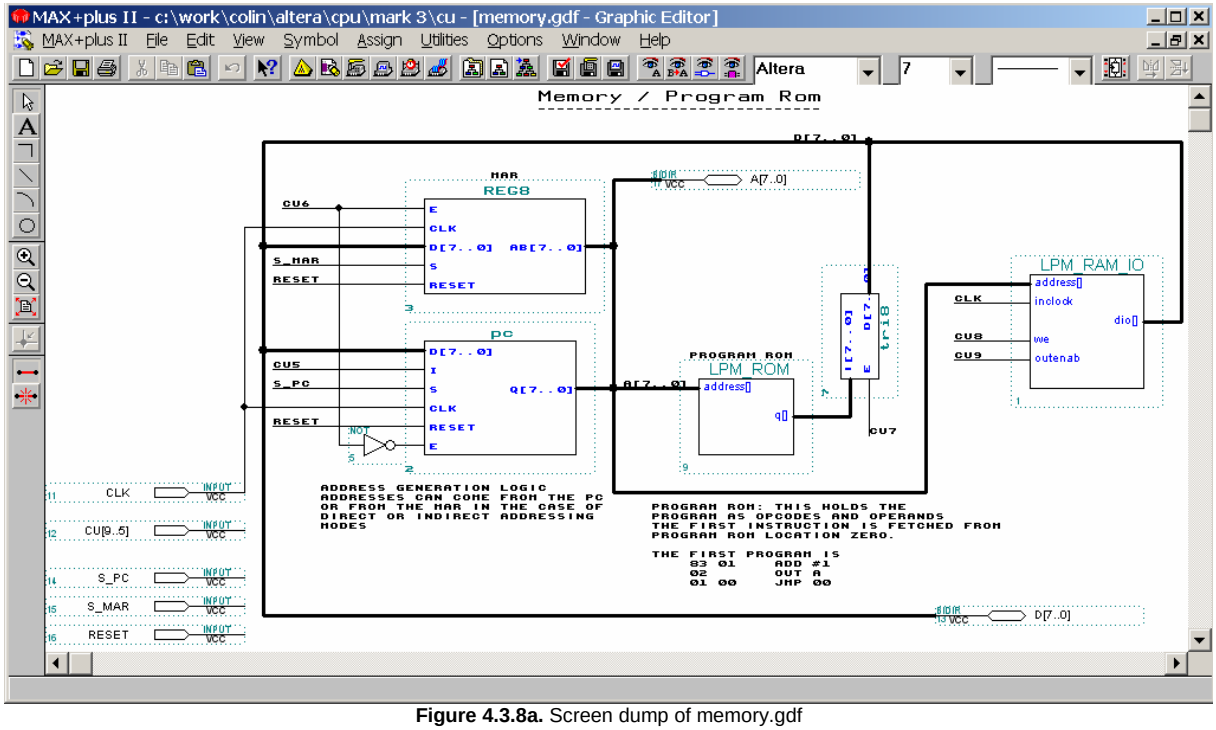


Figure 4.3.8a. Screen dump of memory.gdf

4.3.9. Microcode

The instruction register has been limited (four address lines); hence this CPU can have a maximum of 16 instructions. But each instruction can implement up to 16 different ALU functions, hence providing up to 256 different opcodes (extremely powerful). But more address are required for a full functionally of opcodes, or the accumulator bank and input/output could be configured using a latch connected to the data bus, rather than direct control by the microcode. This would be slower, but would reduce the amount of microcode required, for example at present a set of microcode is required to write or read each of the 8 ports (that's 16 instructions, e.g. OUTA #, OUTB #, OUTC #, INA #, INB # ,INC #), if the selection of the port is set by a latch connected to data bus, then only two sets of microcode are required (e.g. OUT I,#, IN I,#, where I specifies the port, hence this opcode is a 3 byte instruction). Hence this CPU still needs to be improved, to be a practical/useful CPU.

Another option to solve the configuration of the ports, accumulators and increase number of microcode sets, is to use a 24-bit instruction register, where 8-bits are used for the microcode address (up to 256 micro instructions) and the top 16-bits are used to configure the ALU function (4-bits), select output port (3-bits), select accumulator for OP1 (3-bits), select accumulator for OP2 (3-bits), select accumulator to write to (3-bits). This will reduce the amount of microcode required, as each opcode can carry out many operations, but this makes the CPU 3-time slower as the data bus is only 8-bits wide, hence IR must be filled in three stages, another drawback is that programs will require more space (24-bit operand instead of 8-bit).

First initialised the system on power up: -

Address Radix		Data Radix																
		Microcode ROM 3					Microcode ROM 2					Microcode ROM 1						
7..4	3..0	23..21	20..18	17	16	15	14..12	11	10	9	8	7	6	5	4	3..1	0	
OP ADDRESS	UPC	SB Acc	SA Acc	SW Acc			Select Port	PortE	Port_IO	RAM_OUTE	RAM_WE	ROME	PCE	PCI	RESE	SEL S	UPCCL	Comments
0000	0000	001	000	000			000	0	0	0	0	1	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0000	0001	001	000	000			000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
0000	0010	001	000	000			000	0	0	0	0	1	0	1	0	100	1	Strobe IR and Clear UPC. Jump

PCE is active low

S1: RBS = 001, S2: RESS = 010
S4: IRS = 100, S5: PCS = 101,
S6: MARS = 110, S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

ACCA immediate opcode (address #3, e.g. ADDA #, LDA #, # uses acc H): -

PCE is active low

S1: RBS = 001, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101,
S6: MARS = 110, S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

Address Radix		Data Radix																
		Microcode ROM 3					Microcode ROM 2					Microcode ROM 1						
7..4	3..0	23..21	20..18	17	16	15	14..12	11	10	9	8	7	6	5	4	3..1	0	
OP ADDRESS	UPC	SB Acc	SA Acc	SW Acc			Select Port	PortE	Port_IO	RAM_OUTE	RAM_WE	ROME	PCE	PCI	RESE	SEL S	UPCCL	Comments
0011	0000	111	000	111			000	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0011	0001	111	000	111			000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
0011	0010	111	000	111			000	0	0	0	0	1	0	1	0	001	0	Strobe ACCH got second byte, also inc PC
0011	0011	111	000	111			000	0	0	0	0	0	0	0	0	000	0	ALU operation (e.g. add ACCA and ACCH)
0011	0100	111	000	000			000	0	0	0	0	0	0	0	0	011	0	Strobe the result register, got answer
0011	0101	111	000	000			000	0	0	0	0	0	0	0	1	000	0	RSE, answer → DB
0011	0110	111	000	000			000	0	0	0	0	0	0	0	1	001	0	Strobe ACCA, put answer here
0011	0111	111	000	111			000	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0011	1000	111	000	111			000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
0011	1001	111	000	111			000	0	0	0	0	1	0	1	0	100	1	Strobe IR and clear UPC, Jump to new

General purpose OUTA,A opcode (address #2, e.g. OUTA A): -

PCE is active low

S1: RBS = 001, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101,
S6: MARS = 110, S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

Address Radix		Data Radix																
		Microcode ROM 3					Microcode ROM 2					Microcode ROM 1						
7..4	3..0	23..21	20..18	17	16	15	14..12	11	10	9	8	7	6	5	4	3..1	0	
OP ADDRESS	UPC	SB Acc	SA Acc	SW Acc			Select Port	PortE	Port_IO	RAM_OUTE	RAM_WE	ROME	PCE	PCI	RESE	SEL S	UPCCL	Comments
0010	0000	000	000	000			000	0	0	0	0	0	0	0	0	011	0	ALU function (e.g. passing OP1, Inc RES)
0010	0001	000	000	000			000	0	0	0	0	0	0	0	1	000	0	Enable Results → DB
0010	0010	000	000	000			000	0	0	0	0	0	0	0	1	111	0	Keep enabled then strobe output port A
0010	0011	000	000	000			000	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0010	0100	000	000	000			000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
0010	0101	000	000	000			000	0	0	0	0	1	0	1	0	100	1	Strobe IR and clear UPC, Jump to new

Jump Immediate opcode (address 01, e.g. JUMP #): -

PCE is active low

S1: RBS = 001, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101,
S6: MARS = 110, S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

Address Radix		Data Radix																
		Microcode ROM 3					Microcode ROM 2					Microcode ROM 1						
7..4	3..0	23..21	20..18	17	16	15	14..12	11	10	9	8	7	6	5	4	3..1	0	
OP ADDRESS	UPC	SB Acc	SA Acc	SW Acc			Select Port	PortE	Port_IO	RAM_OUTE	RAM_WE	ROME	PCE	PCI	RESE	SEL S	UPCCL	Comments
0001	0000	000	000	000			000	0	0	0	0	0	0	0	0	000	0	Enable PC, get next byte
0001	0001	000	000	000			000	0	0	0	0	1	0	0	0	000	0	ROM, second byte → DB
0001	0010	000	000	000			000	0	0	0	0	1	0	0	0	101	0	PC, new value in PC
0001	0011	000	000	000			000	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0001	0100	000	000	000			000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
0001	0101	000	000	000			000	0	0	0	0	1	0	1	0	100	1	Strobe IR and clear UPC, Jump to new

ACCB immediate opcode (address #4, e.g. ADDB #, LDB #, # uses acc H): -

PCE is active low

S1: RBS = 001, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101,
S6: MARS = 110, S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

Address Radix		Data Radix																
		Microcode ROM 3					Microcode ROM 2					Microcode ROM 1						
7..4	3..0	23..21	20..18	17	16	15	14..12	11	10	9	8	7	6	5	4	3..1	0	
OP ADDRESS	UPC	SB Acc	SA Acc	SW Acc			Select Port	PortE	Port_IO	RAM_OUTE	RAM_WE	ROME	PCE	PCI	RESE	SEL S	UPCCL	Comments
0100	0000	111	001	111			000	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0100	0001	111	001	111			000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
0100	0010	111	001	111			000	0	0	0	0	1	0	1	0	001	0	Strobe ACCH got second byte, also inc PC
0100	0011	111	001	111			000	0	0	0	0	0	0	0	0	000	0	ALU operation (e.g. add ACCB and ACCH)
0100	0100	111	001	001			000	0	0	0	0	0	0	0	0	011	0	Strobe the result register, got answer
0100	0101	111	001	001			000	0	0	0	0	0	0	0	1	000	0	RSE, answer → DB
0100	0110	111	001	001			000	0	0	0	0	0	0	0	1	001	0	Strobe ACCA, put answer here

0100	0111	111	001	111	000	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0100	1000	111	001	111	000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
0100	1001	111	001	111	000	0	0	0	0	1	0	1	0	100	1	Strobe IR and clear UPC, Jump to new

ACCC immediate opcode (address #5, e.g. ADDC #, LDC #, # uses acc H): -

Address Radix		Data Radix															
		Microcode ROM 3					Microcode ROM 2					Microcode ROM 1					
7..4	3..0	23..21	20..18	17	16	15	14..12	11	10	9	8	7	6	5	4	3..1	0
OP ADDRESS	UPC	SB Acc	SA Acc	SW Acc			Select Port	PortE	Port_IO	RAM_OUTE	RAM_WE	ROME	PCE	PCI	RESE	SEL S	UPCCL
																	Comments
0101	0000	111	010	111			000	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0101	0001	111	010	111			000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
0101	0010	111	010	111			000	0	0	0	0	1	0	1	0	001	0	Strobe ACCH got second byte, also inc PC
0101	0011	111	010	111			000	0	0	0	0	0	0	0	0	000	0	ALU operation (e.g. add ACCC and ACCH)
0101	0100	111	010	010			000	0	0	0	0	0	0	0	0	011	0	Strobe the result register, got answer
0101	0101	111	010	010			000	0	0	0	0	0	0	0	1	000	0	RSE, answer → DB
0101	0110	111	010	010			000	0	0	0	0	0	0	0	1	001	0	Strobe ACCA, put answer here
0101	0111	111	010	111			000	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0101	1000	111	010	111			000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
0101	1001	111	010	111			000	0	0	0	0	1	0	1	0	100	1	Strobe IR and clear UPC, Jump to new

PCE is active low

S1: RBS = 001, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101,
S6: MARS = 110, S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

ACCD immediate opcode (address #6, e.g. ADDD #, LDD #, # uses acc H): -

Address Radix		Data Radix															
		Microcode ROM 3					Microcode ROM 2					Microcode ROM 1					
7..4	3..0	23..21	20..18	17	16	15	14..12	11	10	9	8	7	6	5	4	3..1	0
OP ADDRESS	UPC	SB Acc	SA Acc	SW Acc			Select Port	PortE	Port_IO	RAM_OUTE	RAM_WE	ROME	PCE	PCI	RESE	SEL S	UPCCL
																	Comments
0110	0000	111	011	111			000	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0110	0001	111	011	111			000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
0110	0010	111	011	111			000	0	0	0	0	1	0	1	0	001	0	Strobe ACCH got second byte, also inc PC
0110	0011	111	011	111			000	0	0	0	0	0	0	0	0	000	0	ALU operation (e.g. add ACCD and ACCH)
0110	0100	111	011	011			000	0	0	0	0	0	0	0	0	011	0	Strobe the result register, got answer
0110	0101	111	011	011			000	0	0	0	0	0	0	0	1	000	0	RSE, answer → DB
0110	0110	111	011	011			000	0	0	0	0	0	0	0	1	001	0	Strobe ACCA, put answer here
0110	0111	111	011	111			000	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0110	1000	111	011	111			000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
0110	1001	111	011	111			000	0	0	0	0	1	0	1	0	100	1	Strobe IR and clear UPC, Jump to new

PCE is active low

S1: RBS = 001, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101,
S6: MARS = 110, S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

General purpose OUTB,B opcode (address #7, e.g. OUTB B): -

Address Radix		Data Radix															
		Microcode ROM 3					Microcode ROM 2					Microcode ROM 1					
7..4	3..0	23..21	20..18	17	16	15	14..12	11	10	9	8	7	6	5	4	3..1	0
OP ADDRESS	UPC	SB Acc	SA Acc	SW Acc			Select Port	PortE	Port_IO	RAM_OUTE	RAM_WE	ROME	PCE	PCI	RESE	SEL S	UPCCL
																	Comments
0111	0000	000	001	000			001	0	0	0	0	0	0	0	0	011	0	ALU function (e.g. passing OP1, Inc RES)
0111	0001	000	001	000			001	0	0	0	0	0	0	0	1	000	0	Enable Results → DB
0111	0010	000	001	000			001	0	0	0	0	0	0	0	1	111	0	Keep enabled then strobe output port B
0111	0011	000	001	000			001	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
0111	0100	000	001	000			001	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
0111	0101	000	001	000			001	0	0	0	0	1	0	1	0	100	1	Strobe IR and clear UPC, Jump to new

PCE is active low

S1: RBS = 001, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101,
S6: MARS = 110, S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

General purpose OUTC,C opcode (address #8, e.g. OUTC C): -

Address Radix		Data Radix															
		Microcode ROM 3					Microcode ROM 2					Microcode ROM 1					
7..4	3..0	23..21	20..18	17	16	15	14..12	11	10	9	8	7	6	5	4	3..1	0
OP ADDRESS	UPC	SB Acc	SA Acc	SW Acc			Select Port	PortE	Port_IO	RAM_OUTE	RAM_WE	ROME	PCE	PCI	RESE	SEL S	UPCCL

PCE is active low

S1: RBS = 001, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101,
S6: MARS = 110, S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

																Comments
1000	0000	000	010	000	010	0	0	0	0	0	0	0	0	011	0	ALU function (e.g. passing OP1, Inc RES)
1000	0001	000	010	000	010	0	0	0	0	0	0	0	1	000	0	Enable Results → DB
1000	0010	000	010	000	010	0	0	0	0	0	0	0	1	111	0	Keep enabled then strobe output port C
1000	0011	000	010	000	010	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
1000	0100	000	010	000	010	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
1000	0101	000	010	000	010	0	0	0	0	1	0	1	0	100	1	Strobe IR and clear UPC, Jump to new

General purpose OUTD,D opcode (address #9, e.g. OUTD D): -

Address Radix		Data Radix															
		Microcode ROM 3					Microcode ROM 2					Microcode ROM 1					
7..4	3..0	23..21	20..18	17	16	15	14..12	11	10	9	8	7	6	5	4	3..1	0
OP ADDRESS	UPC	SB Acc	SA Acc	SW Acc			Select Port	PortE	Port_IO	RAM_OUTE	RAM_WE	ROME	PCE	PCI	RESE	SEL S	UPCCL
																	Comments
1001	0000	000	011	000			011	0	0	0	0	0	0	0	0	011	0	ALU function (e.g. passing OP1, Inc RES)
1001	0001	000	011	000			011	0	0	0	0	0	0	0	1	000	0	Enable Results → DB
1001	0010	000	011	000			011	0	0	0	0	0	0	0	1	111	0	Keep enabled then strobe output port D
1001	0011	000	011	000			011	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
1001	0100	000	011	000			011	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
1001	0101	000	011	000			011	0	0	0	0	1	0	1	0	100	1	Strobe IR and clear UPC, Jump to new

PCE is active low

S1: RBS = 001, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101,
S6: MARS = 110, S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

General purpose OUTA,B opcode (address #A, e.g. OUTA B): -

Address Radix		Data Radix															
		Microcode ROM 3					Microcode ROM 2					Microcode ROM 1					
7..4	3..0	23..21	20..18	17	16	15	14..12	11	10	9	8	7	6	5	4	3..1	0
OP ADDRESS	UPC	SB Acc	SA Acc	SW Acc			Select Port	PortE	Port_IO	RAM_OUTE	RAM_WE	ROME	PCE	PCI	RESE	SEL S	UPCCL
																	Comments
1010	0000	000	001	000			000	0	0	0	0	0	0	0	0	011	0	ALU function (e.g. passing OP1, Inc RES)
1010	0001	000	001	000			000	0	0	0	0	0	0	0	1	000	0	Enable Results → DB
1010	0010	000	001	000			000	0	0	0	0	0	0	0	1	111	0	Keep enabled then strobe output port D
1010	0011	000	001	000			000	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
1010	0100	000	001	000			000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
1010	0101	000	001	000			000	0	0	0	0	1	0	1	0	100	1	Strobe IR and clear UPC, Jump to new

PCE is active low

S1: RBS = 001, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101,
S6: MARS = 110, S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

General purpose OUTA,C opcode (address #B, e.g. OUTA C): -

Address Radix		Data Radix															
		Microcode ROM 3					Microcode ROM 2					Microcode ROM 1					
7..4	3..0	23..21	20..18	17	16	15	14..12	11	10	9	8	7	6	5	4	3..1	0
OP ADDRESS	UPC	SB Acc	SA Acc	SW Acc			Select Port	PortE	Port_IO	RAM_OUTE	RAM_WE	ROME	PCE	PCI	RESE	SEL S	UPCCL
																	Comments
1011	0000	000	010	000			000	0	0	0	0	0	0	0	0	011	0	ALU function (e.g. passing OP1, Inc RES)
1011	0001	000	010	000			000	0	0	0	0	0	0	0	1	000	0	Enable Results → DB
1011	0010	000	010	000			000	0	0	0	0	0	0	0	1	111	0	Keep enabled then strobe output port D
1011	0011	000	010	000			000	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
1011	0100	000	010	000			000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
1011	0101	000	010	000			000	0	0	0	0	1	0	1	0	100	1	Strobe IR and clear UPC, Jump to new

PCE is active low

S1: RBS = 001, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101,
S6: MARS = 110, S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

General purpose OUTA,D opcode (address #C, e.g. OUTA D): -

Address Radix		Data Radix															
		Microcode ROM 3					Microcode ROM 2					Microcode ROM 1					
7..4	3..0	23..21	20..18	17	16	15	14..12	11	10	9	8	7	6	5	4	3..1	0
OP ADDRESS	UPC	SB Acc	SA Acc	SW Acc			Select Port	PortE	Port_IO	RAM_OUTE	RAM_WE	ROME	PCE	PCI	RESE	SEL S	UPCCL
																	Comments
1100	0000	000	011	000			000	0	0	0	0	0	0	0	0	011	0	ALU function (e.g. passing OP1, Inc RES)
1100	0001	000	011	000			000	0	0	0	0	0	0	0	1	000	0	Enable Results → DB

PCE is active low

S1: RBS = 001, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101,
S6: MARS = 110, S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

1100	0010	000	011	000	000	0	0	0	0	0	0	0	1	111	0	Keep enabled then strobe output port D
1100	0011	000	011	000	000	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
1100	0100	000	011	000	000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
1100	0101	000	011	000	000	0	0	0	0	1	0	1	0	100	1	Strobe IR and clear UPC, Jump to new

General purpose INA, A opcode (address #D, e.g. INA A): -

Address Radix		Data Radix																
		Microcode ROM 3					Microcode ROM 2					Microcode ROM 1						
7..4	3..0	23..21	20..18	17	16	15	14..12	11	10	9	8	7	6	5	4	3..1	0	
OP ADDRESS	UPC	SB Acc	SA Acc	SW Acc			Select Port	PortE	Port_IO	RAM_OUTE	RAM_WE	ROME	PCE	PCI	RESE	SEL S	UPCCL	Comments
1101	0000	000	000	000			000	0	1	0	0	0	0	0	0	111	0	Configure port A as a input
1101	0001	000	000	000			000	1	1	0	0	0	0	0	0	111	0	PortA → DB
1101	0010	000	000	000			000	0	0	0	0	0	0	0	0	001	0	DB → AccA
1101	0011	000	000	000			000	0	0	0	0	0	0	0	0	000	0	Enable (PC) → AB & Enable ROM
1101	0100	000	000	000			000	0	0	0	0	1	0	0	0	000	0	Enable ROM in earnest, (ROM)→DB
1101	0101	000	000	000			000	0	0	0	0	1	0	1	0	100	1	Strobe IR and clear UPC, Jump to new

PCE is active low

S1: RBS = 001, S3: RESS = 011
S4: IRS = 100, S5: PCS = 101,
S6: MARS = 110, S7: OUTS = 111, leave at 00 for unused

ALU function directly from IR

4.3.10. Supported Opcodes

Note: This list is not a complete list, there are more opcode available (e.g. address #A to #D not included).

Machine Code	Opcode	Function
0x13 0x##	LDA #	A = #
0x33 0x##	NOTA #	A = /#
0x43 0x##	ANDA #	A = A and #
0x53 0x##	ORA #	A = A or #
0x63 0x##	XORA #	A = A xor #
0x73 0x##	INCA #	A = A + 1
0x83 0x##	ADDA #	A = A + #
0x93 0x##	ADDAI #	A = A + # + 1
0xA3 0x##	SUBA #	A = A - #
0xB3 0x##	SUBA #,A	A = # - A
0xC3 0x##	DECA	A = A -1
0xD3 0x##	DECA #,A	A = # - 1
0x14 0x##	LDB #	B = #
0x34 0x##	NOTB #	B = /#
0x44 0x##	ANDB #	B = B and #
0x54 0x##	ORB #	B = B or #
0x64 0x##	XORB #	B = B xor #
0x74 0x##	INCB #	B = B + 1
0x84 0x##	ADDB #	B = B + #
0x94 0x##	ADDBI #	B = B + # + 1
0xA4 0x##	SUBB #	B = B - #
0xB4 0x##	SUBB #,A	B = # - B
0xC4 0x##	DECB	B = B -1
0xD4 0x##	DECB #,A	B = # - 1
0x15 0x##	LDC #	C = #
0x35 0x##	NOTC #	C = /#
0x45 0x##	ANDC #	C = C and #
0x55 0x##	ORC #	C = C or #
0x65 0x##	XORC #	C = C xor #
0x75 0x##	INCC #	C = C + 1
0x85 0x##	ADDC #	C = C + #
0x95 0x##	ADDCI #	C = C + # + 1
0xA5 0x##	SUBC #	C = C - #
0xB5 0x##	SUBC #,A	C = # - C

0xC5 0x##	DECC	C = C -1
0xD5 0x##	DECC #,A	C = # - 1
0x16 0x##	LDD #	D = #
0x36 0x##	NOTD #	D = /#
0x46 0x##	ANDD #	D = D and #
0x56 0x##	ORD #	D = D or #
0x66 0x##	XORD #	D = D xor #
0x76 0x##	INCD #	D = D + 1
0x86 0x##	ADDD #	D = D + #
0x96 0x##	ADDDI #	D = D + # + 1
0xA6 0x##	SUBD #	D = D - #
0xB6 0x##	SUBD #,A	D = # - D
0xC6 0x##	DECD	D = D -1
0xD6 0x##	DECD #,A	D = # - 1
0x02	OUTA, A	Output port A = A
0x12	OUTA, H	Output port A = H
0x22	OUTA, NOTA	Output port A = not A
0x32	OUTA, NOTH	Output port A = not H
0x42	OUTA, AandH	Output port A = A and H
0x52	OUTA, AorH	Output port A = A or H
0x62	OUTA, AxorH	Output port A = A xor H
0x72	OUTA, INCA	Output port A = A +1
0x82	OUTA, A+H	Output port A = A + H
0x92	OUTA, A+H+1	Output port A = A + H + 1
0xA2	OUTA, A-H	Output port A = A - H
0xB2	OUTA, H-A	Output port A = H - A
0xC2	OUTA, A-1	Output port A = A - 1
0xD2	OUTA, H-1	Output port A = H - 1
0x01 0x##	JMP ##	PC = ##
0x07	OUTB, B	Output port B = B
0x17	OUTB, H	Output port B = H
0x27	OUTB, NOTB	Output port B = not B
0x37	OUTB, NOTH	Output port B = not H
0x47	OUTB, BandH	Output port B = B and H
0x57	OUTB, BorH	Output port B = B or H
0x67	OUTB, BxorH	Output port B = B xor H
0x77	OUTB, INCB	Output port B = B +1
0x87	OUTB, B+H	Output port B = B + H
0x97	OUTB, B+H+1	Output port B = B + H + 1
0xA7	OUTB, B-H	Output port B = B - H
0xB7	OUTB, H-B	Output port B = H – B
0xC7	OUTB, B-1	Output port B = B - 1
0xD7	OUTB, H-1	Output port B = H – 1
0x08	OUTC, C	Output port C = C
0x18	OUTC, H	Output port C = H
0x28	OUTC, NOTC	Output port C = not C
0x38	OUTC, NOTH	Output port C = not H
0x48	OUTC, CandH	Output port C = C and H
0x58	OUTC, CorH	Output port C = C or H
0x68	OUTC, CxorH	Output port C = C xor H
0x78	OUTC, INCC	Output port C = C +1
0x88	OUTC, C+H	Output port C = C + H
0x98	OUTC, C+H+1	Output port C = C + H + 1
0xA8	OUTC, C-H	Output port C = C - H
0xB8	OUTC, H-C	Output port C = H – C
0xC8	OUTC, C-1	Output port C = C - 1
0xD8	OUTC, H-1	Output port C = H - 1
0x09	OUTD, D	Output port D = D

0x19	OUTD, H	Output port D = H
0x29	OUTD, NOTD	Output port D = not D
0x39	OUTD, NOTH	Output port D = not H
0x49	OUTD, DandH	Output port D = D and H
0x59	OUTD, DorH	Output port D = D or H
0x69	OUTD, DxorH	Output port D = D xor H
0x79	OUTD, INCD	Output port D = D +1
0x89	OUTD, D+H	Output port D = D + H
0x99	OUTD, D+H+1	Output port D = D + H + 1
0xA9	OUTD, D-H	Output port D = D - H
0xB9	OUTD, H-D	Output port D = H – D
0xC9	OUTD, D-1	Output port D = D - 1
0xD9	OUTD, H-1	Output port D = H - 1

4.4. Mark 4 - Advanced 16-bit CPU

The design of this CPU is not finished, but some key functional blocks were designed and simulated.

4.4.1. alu16.gdf (Powerful 16-bit ALU)

This ALU is much more powerful than the 8-bit ALU used in the marks 1-3. Obviously being 16-bits makes it more powerful, but the key aspect of the design is the additional flags which make this ALU extremely powerful, allowing the programmer to carry out complex calculations easily.

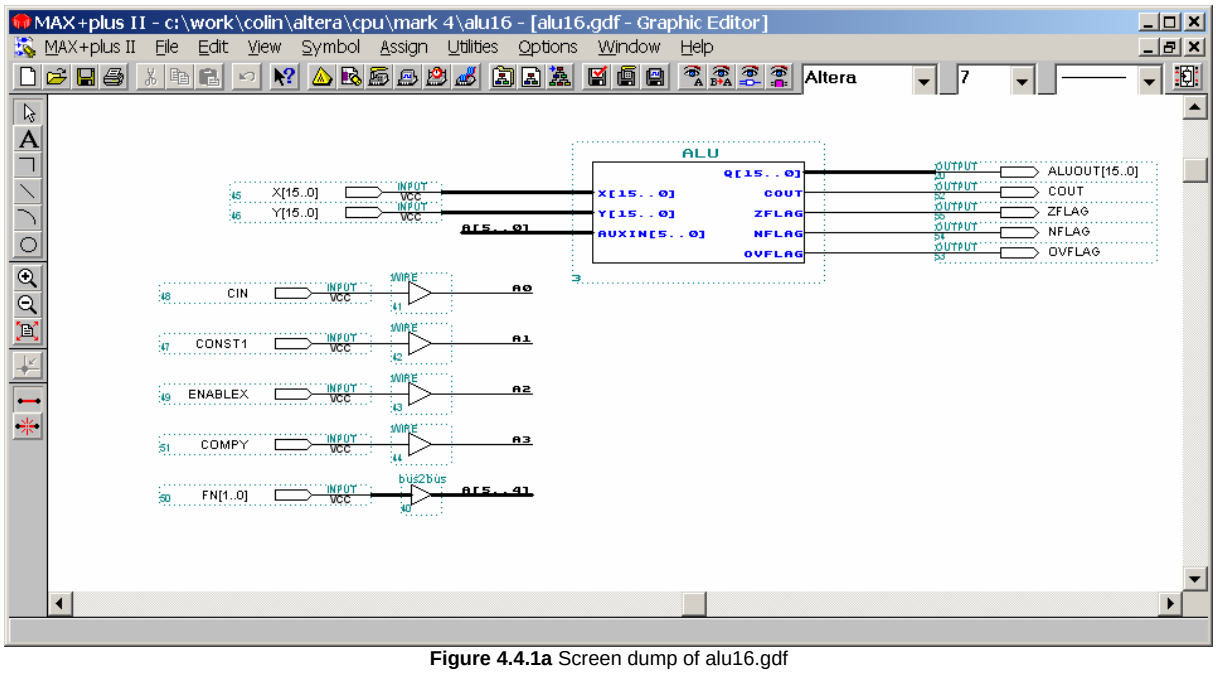


Figure 4.4.1a Screen dump of alu16.gdf

There are four output flags: -

COUT:	Carry out, e.g. if X + Y creates a 16-bit number the COUT file will be set.
ZFLAG:	Zero flag, e.g. if X – Y equals zero this flag will be set.
NFLAG:	Negative flag, e.g. if X – Y results in a negative number this flag will be set.
OVFLAG:	Overflow flag, e.g. if calculation overflows this flag is set.

There are four input flags: -

CIN	Carry In, e.g. X + Y +1.
CONST1	Sets input X to 0b1111111111111111111
ENABLEX	Enable X input, if X is disabled ‘0’ is used in the calculation (e.g. X + Y = 0 +Y)
COMPY	Complement Y.

FN selects ALU function (e.g. 00 = Add X + Y, 01 = X, 10 = Y, 11 = X – Y).

alu.vhd: -

```
-- MAX+plus II VHDL Template
-- Clearable loadable enablable counter

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
USE work.defines.all;
ENTITY onebitalu IS
               port(
                       x, y: std_logic;
```

```
                        fn: in std_logic_vector(1 downto 0);
                        cin: in std_logic;
                        const1, enablex, compy: std_logic;
                        Q: out std_logic;
                        cout: out std_logic
                );
END onebitalu;

architecture a of onebitalu is
        signal xsig, ysig, xory, xandy, xxory: std_logic;
begin
        xsig <= (x and enablex) or const1;
        ysig <= y xor compy;
        xory <= xsig or ysig;
        xandy <= xsig and ysig;
        xxory <= xsig xor ysig;
        Q <= xxory xor cin when fn=ADDFN else
                 xory when fn = ORFN else
                 xandy  when fn = ANDFN else
                 xxory;
        cout <= (xsig and cin ) or (ysig and cin) or xandy;
end a;
----------------------------
-----------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
USE work.defines.all;

ENTITY alu IS

        PORT
        (
                X, Y: in std_logic_vector(wordsize-1 downto 0);
                auxin: in std_logic_vector(5 downto 0);

                Q: out std_logic_vector(wordsize-1 downto 0);
                cout, zflag, nflag, ovflag: out std_logic
        );

END alu;

ARCHITECTURE a OF alu IS
        component onebitalu
                port(
                        x, y: std_logic;
                        fn: in std_logic_vector(1 downto 0);
                        cin: in std_logic;
                        const1, enablex, compy: std_logic;
                        Q: out std_logic;
                        cout: out std_logic
                );
        end component;
        signal coS, Qs: std_logic_vector(wordsize-1 downto 0);
        signal ovS: std_logic;
        signal cin,    const1, enablex, compy: std_logic;
        signal fn: std_logic_vector(1 downto 0);

BEGIN
        cin <= auxin(0);
        const1 <= auxin(1);
        enablex <= auxin(2);
        compy <= auxin(3);
        fn <= auxin(5 downto 4);

        bit0: onebitalu port map
                (X(0), Y(0), fn, cin, const1, enablex, compy,
                Qs(0), coS(0));
        gen_loop: for K in 1 to wordsize-1 generate
                bitK: onebitalu port map
                (X(K), Y(K), fn, coS(K-1), const1, enablex, compy,
                        Qs(K), coS(K));
        end generate;
        cout <= coS(wordsize-1) when fn=ADDFN else
                        '0';
        Q <= Qs;
```

```
        zflag <= '1' when Qs = ZERO_WORD else
                        '0';
        nflag <= Qs(wordsize-1);
        ovS <= X(wordsize-1) xor Y(wordsize-1) xor Qs(wordsize-1) xor coS(wordsize-1);
        ovflag <= ovS when fn=ADDFN else
                '0';
END a;
```

defines.vhd: -

```
-- MAX+plus II VHDL Template
-- Clearable flipflop with enable

LIBRARY ieee;
USE ieee.std_logic_1164.all;

PACKAGE defines IS
        constant wordsize: integer :=16;
        constant ZEROBYTE: std_logic_vector(7 downto 0) := "00000000";
        constant ZERO_DBL_BYTE: std_logic_vector( 15 downto 0) := ZEROBYTE&ZEROBYTE;
        constant ZERO_WORD: std_logic_vector( wordsize-1 downto 0) := ZERO_DBL_BYTE;


-- ALU
        constant ADDFN: std_logic_vector(1 downto 0) := "00";
        constant  ORFN: std_logic_vector(1 downto 0) := "01";
        constant ANDFN: std_logic_vector(1 downto 0) := "10";
        constant XORFN: std_logic_vector(1 downto 0) := "11";


    -- Selecting alufns
        --                                                fn    compy enx  cnst1  cin

        constant INCY: std_logic_vector(5 downto 0) := ADDFN & "0" & "0" & "0" & "1";
        constant NEGY: std_logic_vector(5 downto 0) := ADDFN & "1" & "0" & "0" & "1";
        constant COMY: std_logic_vector(5 downto 0) := ADDFN & "1" & "0" & "0" & "0";
        constant PSSY: std_logic_vector(5 downto 0) :=  ORFN & "0" & "0" & "0" & "0";
        constant ADD:  std_logic_vector(5 downto 0) := ADDFN & "0" & "1" & "0" & "0";
        constant ADDC: std_logic_vector(5 downto 0) := ADDFN & "0" & "1" & "0" & "1";

END defines;
```

One possible improvement to this design could be the use of LPM_DIVIDE, and LPM_MULTIPLY, to carry out multiply and divide operations.

4.4.2. register16.gdf (16-bit Register)

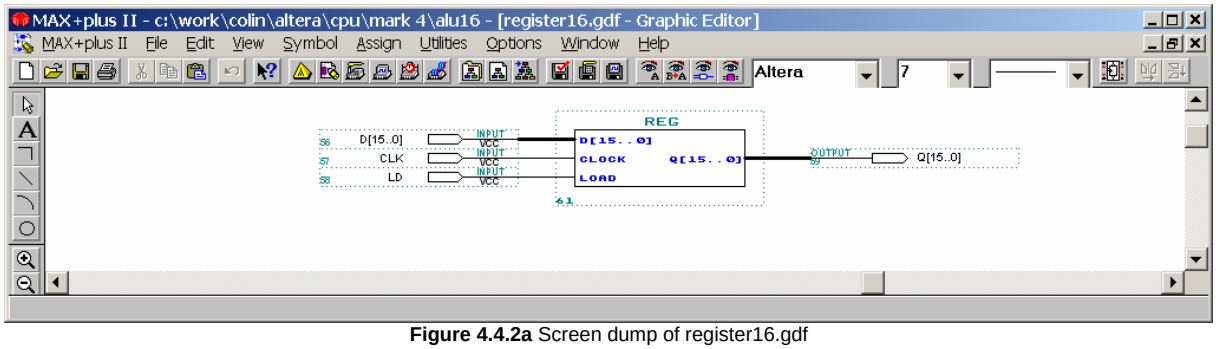


Figure 4.4.2a Screen dump of register16.gdf

reg.vhd: -

```
-- Loadable reg

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
USE work.defines.all;


ENTITY reg IS
```

```
        PORT
        (
                D               : IN    STD_LOGIC_VECTOR(wordsize-1 downto 0);
                clock   : IN    STD_LOGIC;
                load    : IN    STD_LOGIC;
                Q               : OUT   STD_LOGIC_VECTOR(wordsize-1 downto 0)
        );

END reg;

ARCHITECTURE a OF reg IS
        SIGNAL  QSignal : STD_LOGIC_VECTOR(wordsize-1 downto 0);
BEGIN
        PROCESS (clock)
        BEGIN
                IF (clock'EVENT AND clock = '1') THEN
                        IF load = '1' THEN
                                QSignal <= D;
                        ELSE
                                QSignal <= QSignal;
                        END IF;
                END IF;
        END PROCESS;
        Q <= QSignal;
END a;
```

4.4.3. regmux16.gdf (16-bit Register Multiplexer)

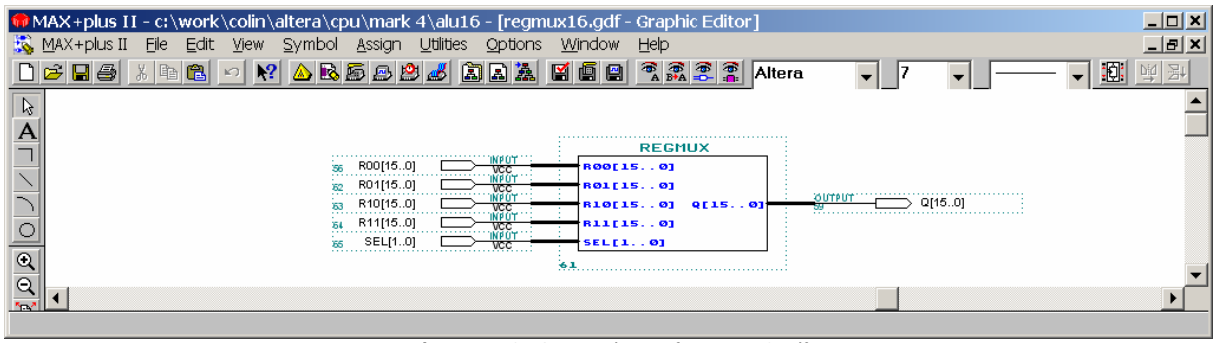


Figure 4.4.3a Screen dump of regmux16.gdf

Sel[1..0] specifies which of the input buses R00 to R11 are connected to the output Q.

regmux.vhd: -

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
USE work.defines.all;

entity regmux is
        port (
                R00, R01, R10, R11: std_logic_vector(wordsize-1 downto 0);
                sel: std_logic_vector(1 downto 0);
                Q: OUT std_logic_vector(wordsize-1 downto 0)
        );
end regmux;

architecture a of regmux is
begin
        Q <= R00 when sel = "00" else
             R01 when sel = "01" else
             R10 when sel = "10" else
             R11;
end a;
```

4.4.4. regmux16.gdf (Four 16-bit Accumulators)

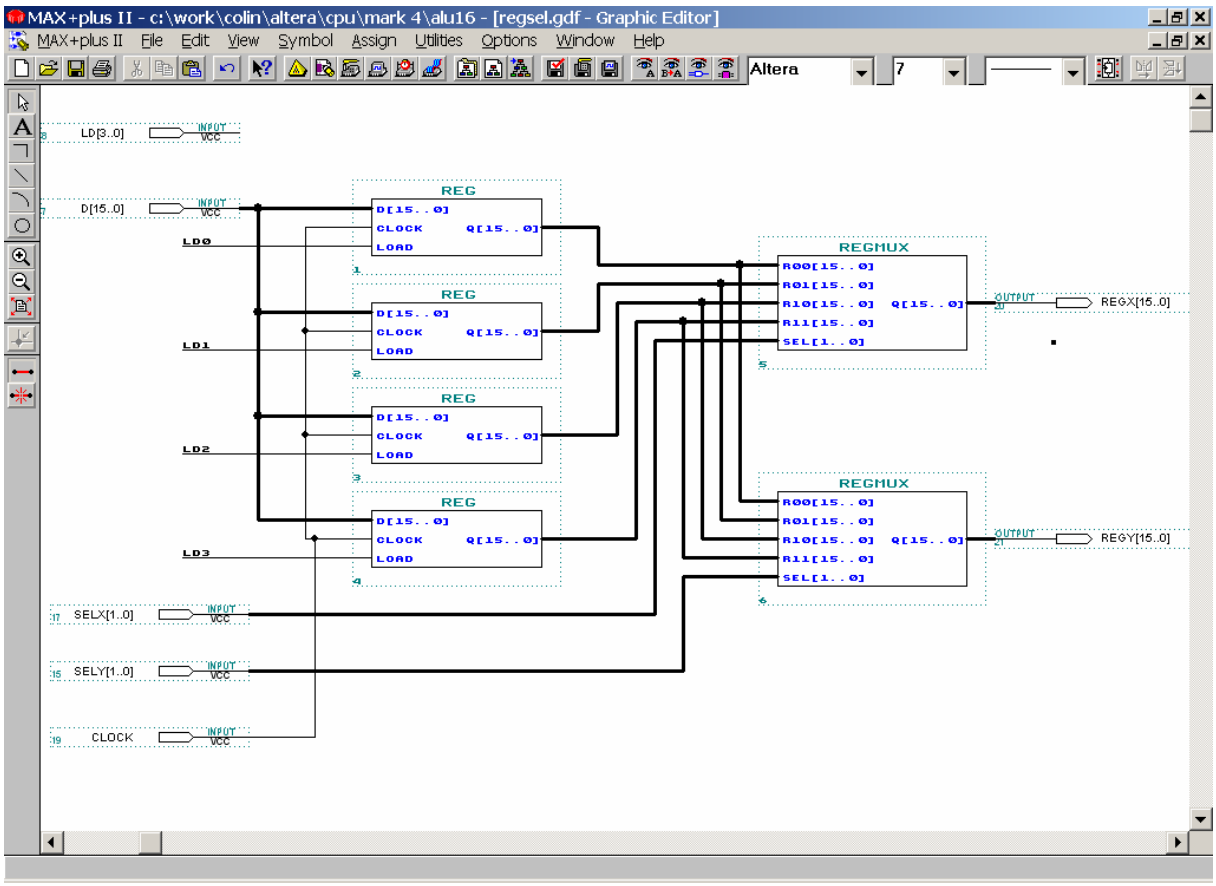


Figure 4.4.4a Screen dump of regsel.gdf

Extremely simple, basically there are four 16-bit register (one for each accumulator), LD[3..0] is used to strobe/load data into the register. Notice the output of all four registers are fed into two register multiplexers, SELX[1..0] is used to select a register to be outputted to REGX[15..0], and SELY[1..0] is used to select a register to be outputted to REGY[15..0].

5.0. CPU SIMULATION

5.1. Mark 1 – Simple CPU

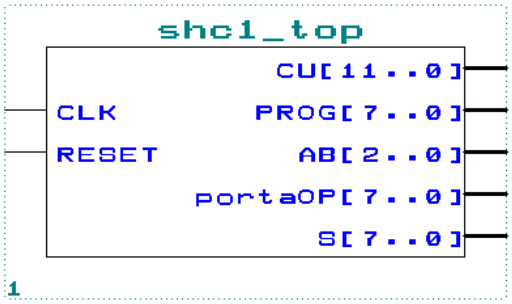


Figure 5.1a: shc1_top Symbol

Test 1: Increment output port continuously

The following program was entered into the program memory: -

```
000     :       03 01 ;         % ADDA #1 %
010     :       02    ;         % OUTA    %
011     :       01 00 ;         % JMP 00  %
```

Basically the output port should continuously increment.

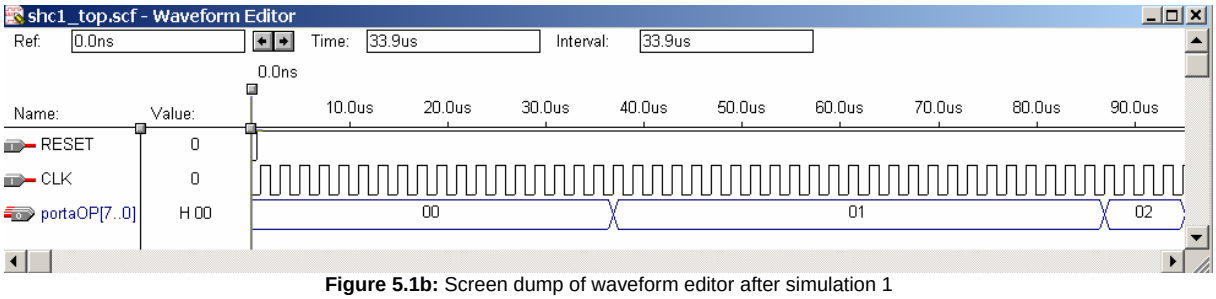


Figure 5.1b: Screen dump of waveform editor after simulation 1

Figure 5.1b clearly shows that portaOP[7..0] is incrementing, hence this proves that the simple 3 line program is working and the microcode for each instruction is correct.

Test 2: Increment output port by 8 continuously

Simulate again this time increment the output port by 8 continuously (make sure test1 was not a fluke): -

```
000     :       03 08 ;         % ADDA #8 %
010     :       02    ;         % OUTA    %
011     :       01 00 ;         % JMP 00  %
```

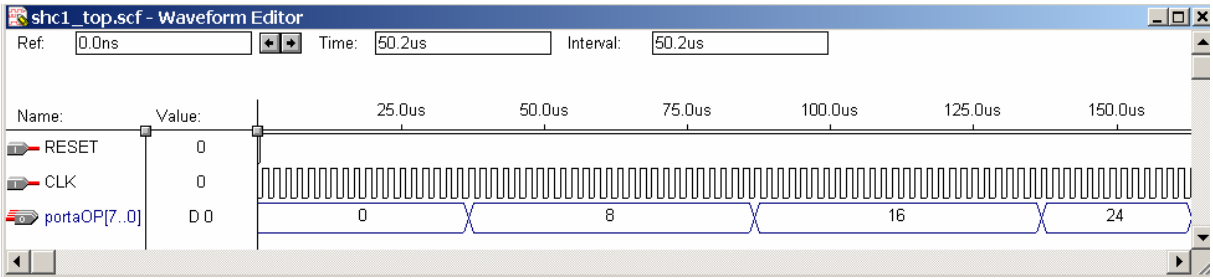


Figure 5.1c: Screen dump of waveform editor after simulation 2

Figure 5.1c clearly shows that portaOP[7..0] is incrementing by 8, as expected.

5.1.1. upc.gdf (4-bit Synchronous Counter)

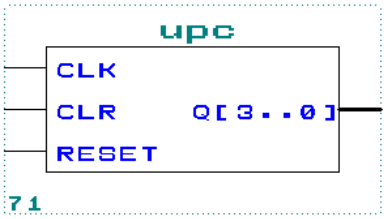


Figure 5.1.1a: UPC Symbol

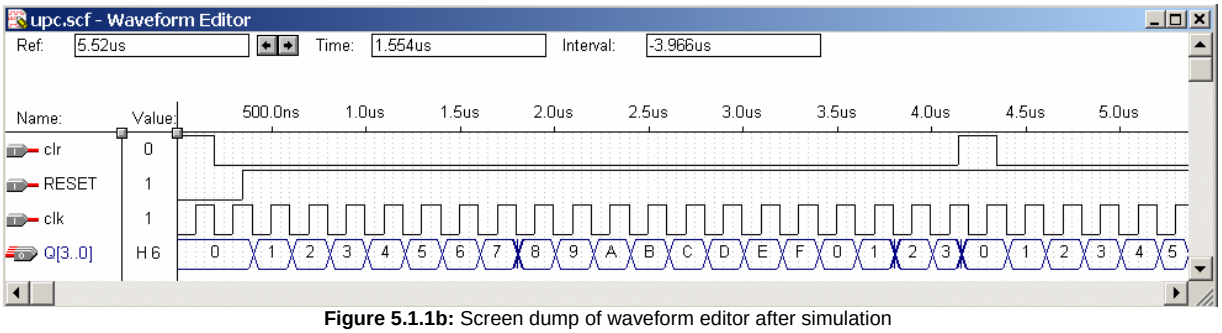


Figure 5.1.1b: Screen dump of waveform editor after simulation

Counts from 0 – 15 – 0, notice when CLR line went HIGH the counter reset.

5.1.2. lat8.gdf (8-bit Latch)

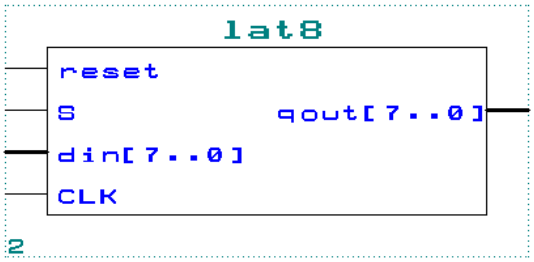


Figure 5.1.2a: lat8.gdf Symbol

din[7..0] count up, S line must be high to latch in new value, else old value should be held: -

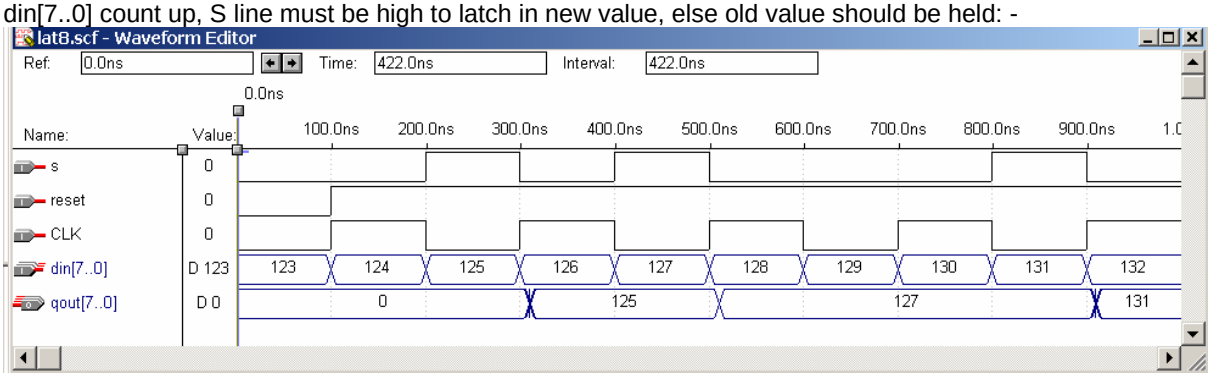


Figure 5.1.2b: Screen dump of waveform editor after simulation

Clearly the latch is working, as it stored new values (e.g. 125,127,131) when S was high, else the old values were held.

5.1.3. tri8.gdf (8-bit Tri-State Buffer)

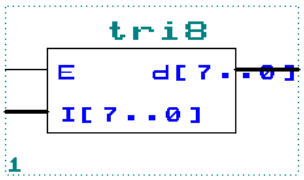


Figure 5.1.3a: tri8.gdf Symbol

I[7..0] counting up, d[7..0] should equal I[7..0] when E is 1, else d[7..0] should be Z: -

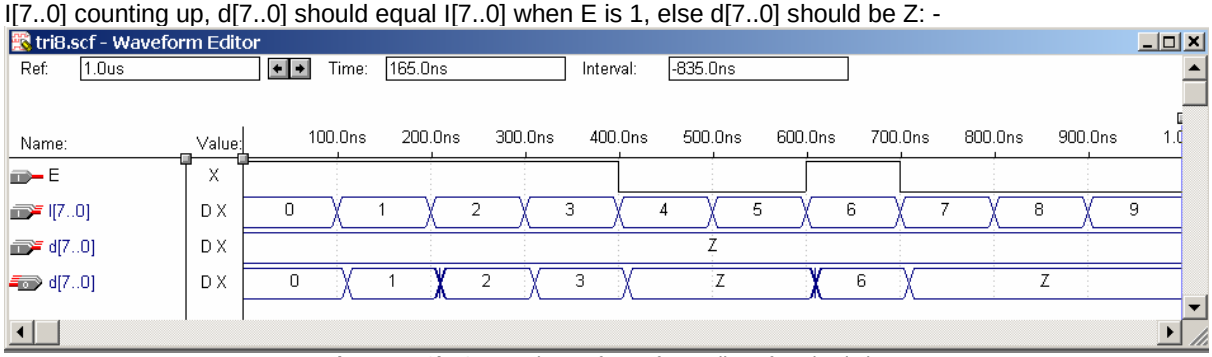


Figure 5.1.2b: Screen dump of waveform editor after simulation

Clearly this simulation was a success, when E equalled '1' I[7..0] passed through to d[7..0], and when E equalled '0' d[7..0] equalled Z (high impendence) therefore other devices can now use the data bus during this period.

5.1.4. pc.gdf (Program Counter)

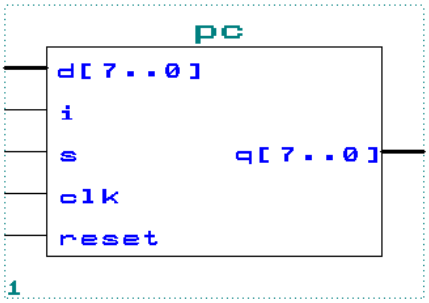


Figure 5.1.4a: pc.gdf Symbol

d[7..0] counting up, S line high, hence q[7..0] should equal d[7..0]: -

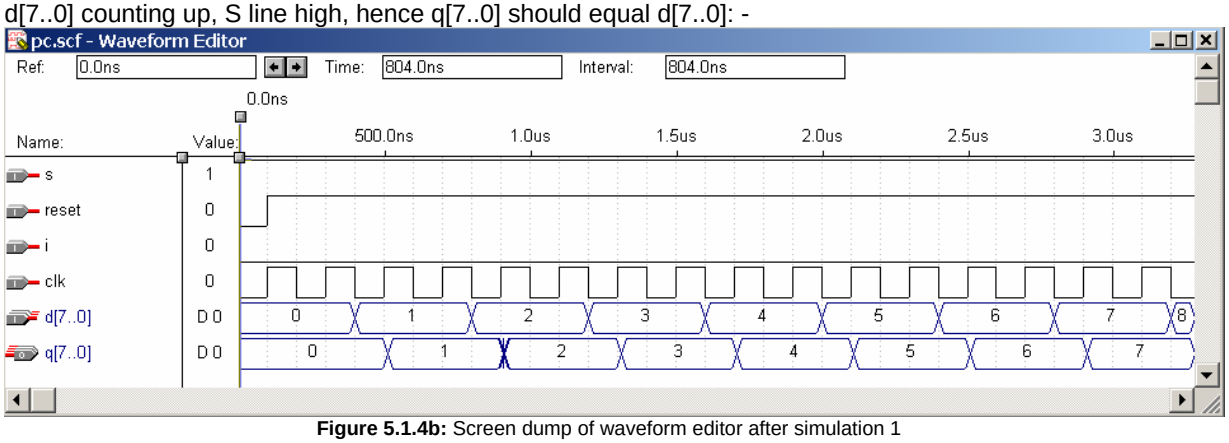


Figure 5.1.4b: Screen dump of waveform editor after simulation 1

S line low, and I line high, hence output q[7..0] should increment on every clock pulse: -

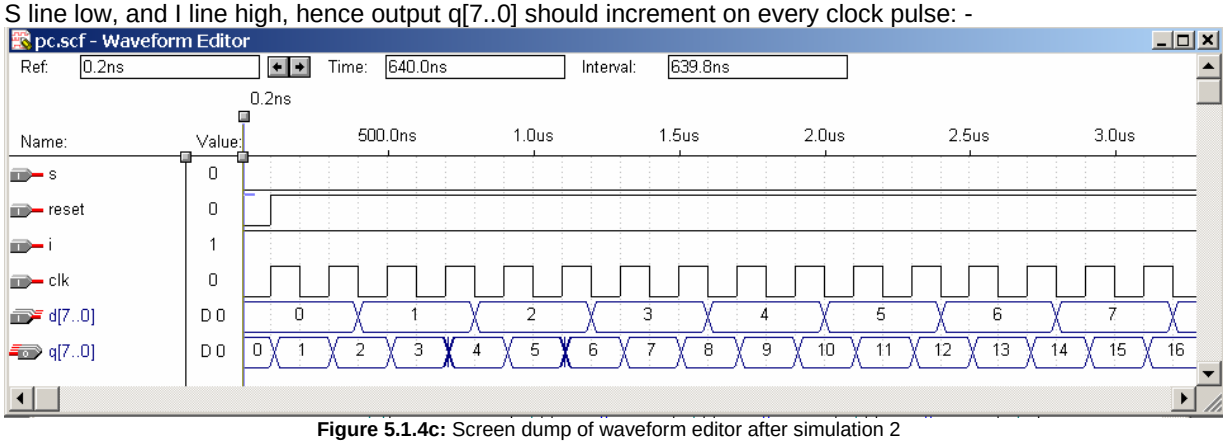


Figure 5.1.4c: Screen dump of waveform editor after simulation 2

5.1.5. t_cell.gdf (T-type Flip-Flop and Latch)

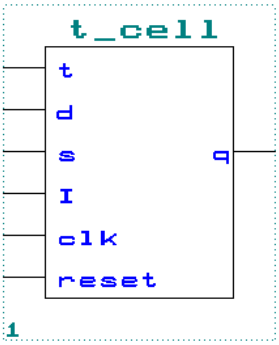


Figure 5.1.5a: t_cell.gdf Symbol

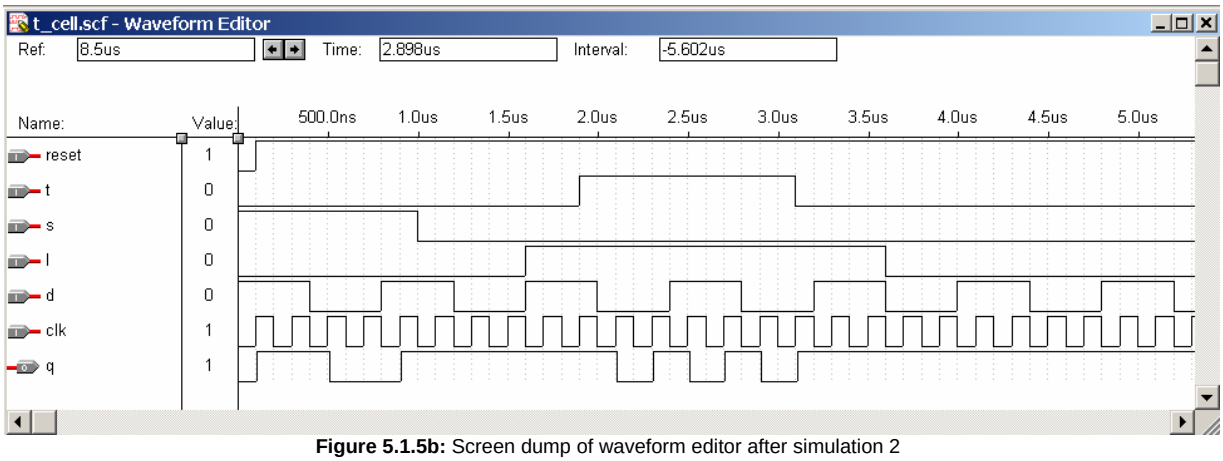


Figure 5.1.5b: Screen dump of waveform editor after simulation 2

The S line is high for the first 1µs of the simulation, hence the cell is in latch mode from figure 5.1.5b it is clear that the output q equalled d during this period of time (slight propagation delay). Between 1.5µs and 3.5µs the I line was set high, hence the cell is now in t-type flip-flop mode, notice when t was also high the output toggled and when t was low the output is held.

5.1.6. alu.vhd (Arithmetic and Logic Unit)

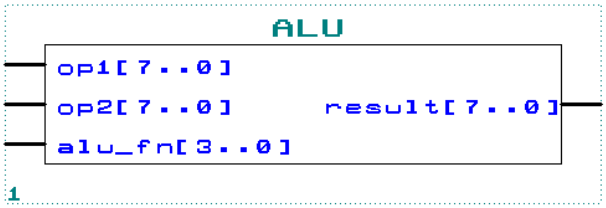


Figure 5.1.6a. Default ALU Symbol

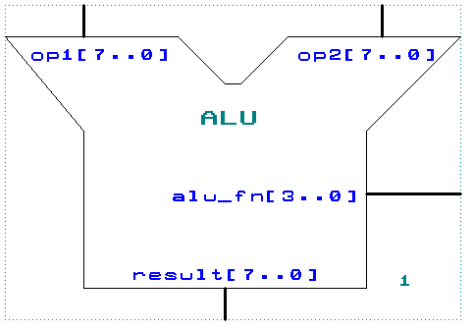


Figure 5.1.6b. Improved ALU Symbol

The Altera package allows the user to manually edit the symbols of any functional block, Figure 5.1.6.b shows a customised symbol of the ALU, this makes the circuit layout more user-friendly and easier to understand. But unfortunately this symbol was lost (accidentally overwritten the improved symbol with the default, shortly after screen dump 5.1.6b was taken), hence the reason why it was not used in the CPU design. It is a good idea to customise all function blocks, but this is time-consuming, since time was limited, it was thought that the time would be better spent improving the CPU rather than the cosmetics.

Let OP1[7..0] = 24, OP2[7..0] = 12, and ALU_FUN[3..0] count through all mathematical operations: -

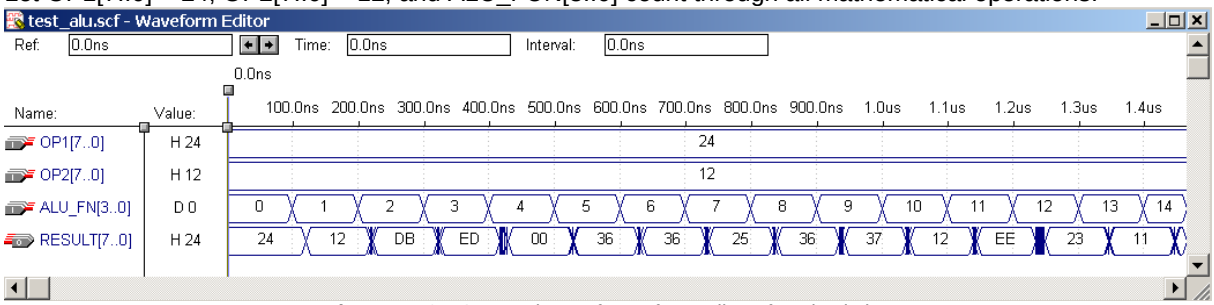


Figure 5.1.6c. Screen dump of waveform editor after simulation

Check the operation of the ALU by manually calculating the result of each function and compare with the simulated result.

ALU_FN	RESULT[7..0]	Simulated Result	Manually calculated result
0	op1	0x24	`result = op1 = 0x24`
1	op2	0x12	`result = op2 = 0x12`
2	not op1	0xDB	`op1 = 0x24 = 0b00100100, result = 0b11011011 = 0xDB`
3	not op2	0xED	`op2 = 0x12 = 0b00010010, result = 0b11101101 = 0xED`
4	op1 and op2	0x00	`result = 0b00100100 and 0b00010010 = 0b00000000 = 0x00`
5	op1 or op2	0x36	`result = 0b00100100 or  0b00010010 = 0b00110110 = 0x36`
6	op1 xor op2	0x36	`result = 0b00100100 or  0b00010010 = 0b00110110 = 0x36`
7	op1 + 1	0x25	`result = op1 + 1 = 0x24 + 1 = 0x25`
8	op1 + op2	0x36	`result = op1 + op2 = 0x24 + 0x12 = 0x36`
9	op1 + op 2 + 1	0x37	`result = op1 + op2 + 1 = 0x24 + 0x12 + 0x01 = 0x37`
10	op1 – op2	0x12	`result = op1 – op2 = 0x24 – 0x12 = 0x12`
11	op2 – op1	0xEE	`result = op2 – op1 = 0x12 – 0x24 = 0xEE`
12	op1 - 1	0x23	`result = op1 – 1 = 0x24 – 0x01 = 0x23`
13	op2 - 1	0x11	`result = op2 – 1 = 0x12 – 0x01 = 0x11`

Clearly the ALU is working…

5.2. Mark 2 – Improved CPU

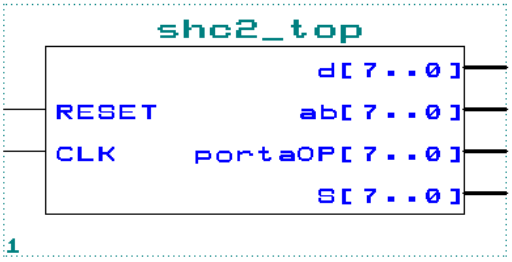


Figure 5.2a. shc2_top symbol

Test 1: Increment output port continuously

The following program was entered into the program memory: -

```
000     :       83 01 ;         % ADDA #1 %
010     :       02    ;         % OUTA    %
011     :       01 00 ;         % JMP 00  %
```

Basically the output port should continuously increment.

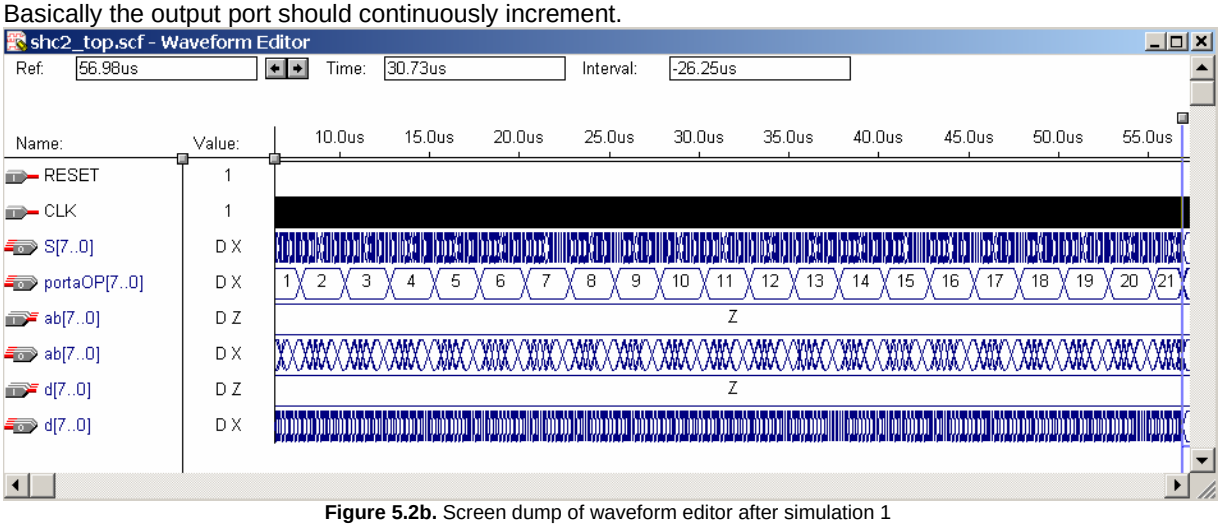


Figure 5.2b. Screen dump of waveform editor after simulation 1

Success, the program works (portaOP counting up)… Note clock period was 100ns that's 10MHz.

Test 2: decrement output port continuously by 16

The following program was entered into the program memory: -

```
000     :       A3 10           ; % SUBA #16 %
010     :       02              ; % OUTA    %
011     :       01 00           ; % JMP 00  %
101     :       0F 0F 0F        ; %UNUSED%
```

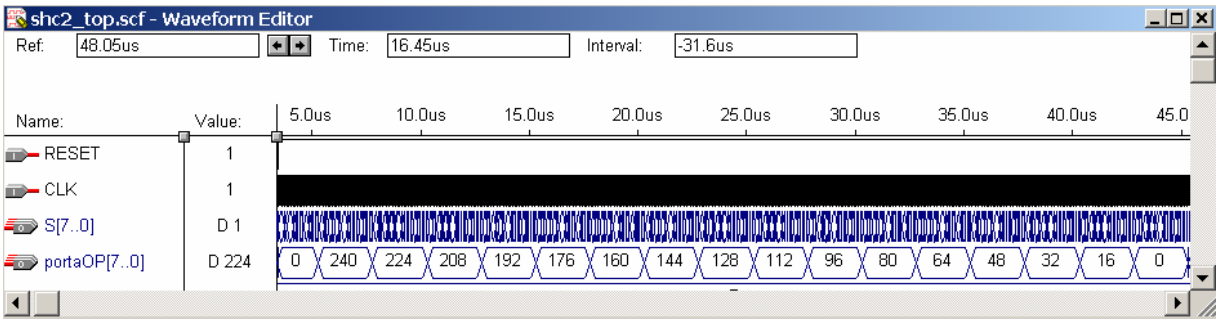


Figure 5.2c. Screen dump of waveform editor after simulation 2

This time the output port should decrement by 16, note the SUBA opcode uses the same microcode as the ADDA, the only difference is the opcode address, where the top four bits specifies the ALU function, which in is 0xA (1010) in this case (subtract op1 from op2).

Success, clearly 240 – 16 = 224 – 16 = 208 – 16 = 192 – 16 = 176 – etc….

Test 3: More Complex Program

The following program was entered into the program memory: -

```
0 :  13 50 ; % LDA #80  %
2 :  02    ; % OUTA     %
3 :  63 05 ; % XORA #05 %
5 :  02    ; % OUTA     %
6 :  83 A  ; % ADDA #10 %
8 :  02    ; % OUTA     %
9 :  A3 05 ; % SUBA #5  %
C :  02    ; % OUTA     %
D :  01 05 ; % JUP 06   %
F :  0F 0F 0F; %UNUSED%
```

80 is loaded into ACC and outputted to the output port, then ACC (80 = 1010000) is XOR with 05 (101) and outputted to the output port (85 = 1010101). 10 is added to ACC (85 + 10 = 95) and outputted to the output port, then 5 is subtracted from ACC (95-5 = 90) and outputted to the output port, then 10 is added, etc...

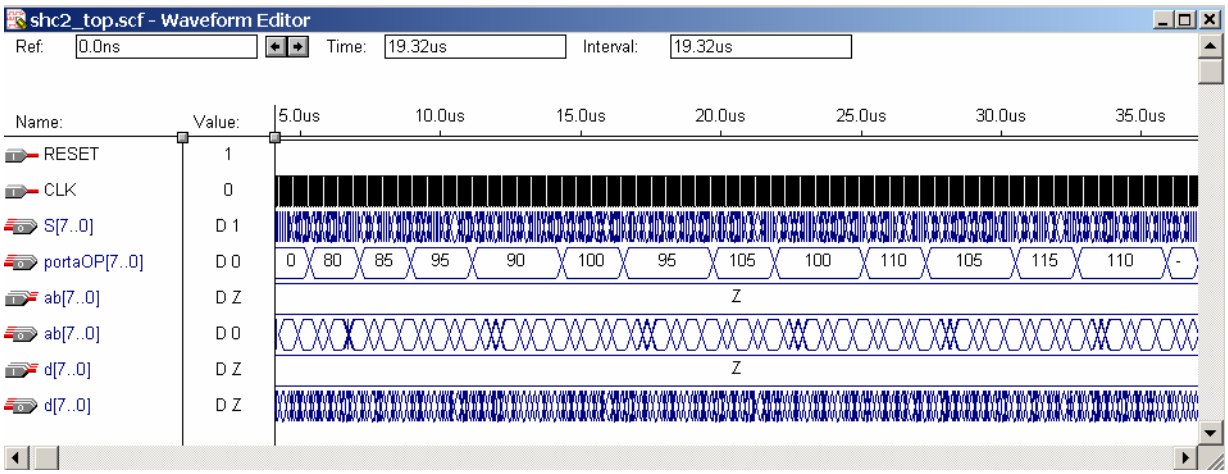


Figure 5.2d. Screen dump of waveform editor after simulation 3

Success, clearly 80 xor 5 = 85 + 10 = 95 – 5 = 90 + 10 = 100 – 5 = 95 + 10 = 105 – 5 = 100 etc…

5.3. Mark 3 – Superior CPU

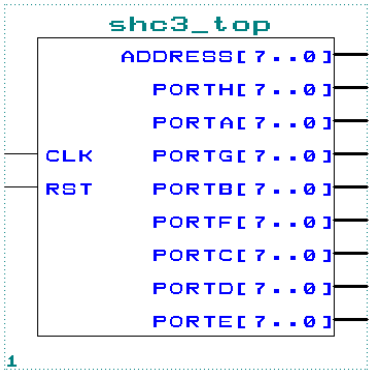


Figure 5.3a. shc3_top symbol

Test 1: Increment portA continuously

The following program was entered into the program memory: -

```
000     :       83 01 ;         % ADDA #1 %
010     :       02    ;         % OUTA A  %
011     :       01 00 ;         % JMP 00  %
```

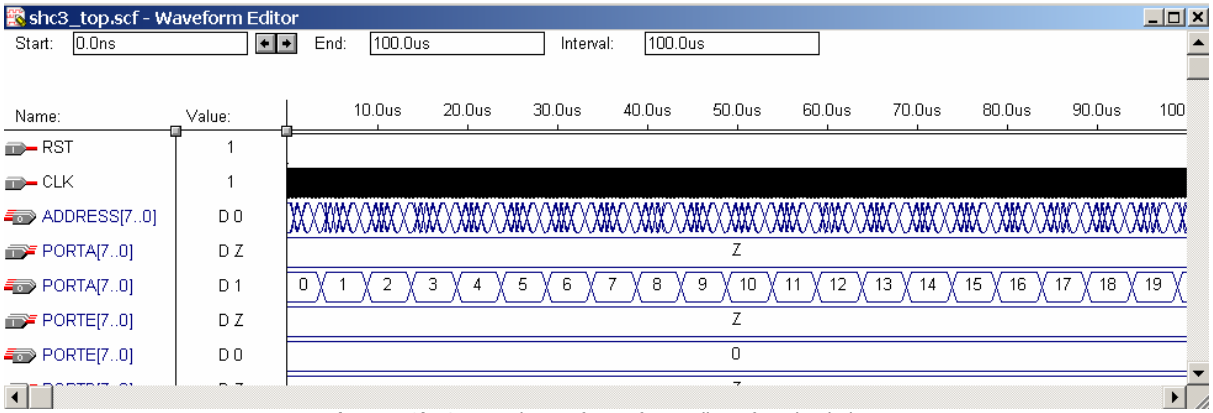


Figure 5.3b. Screen dump of waveform editor after simulation 1

Success, clearly 0 + 1 = 1 + 1 = 2 + 1 = 3 + 1 = 4 + 1 = 5 + 1 = 6 + …

Test 2: Increment portA continuously by 7

The following program was entered into the program memory: -

```
000     :       83 07 ;         % ADDA #7 %
010     :       02    ;         % OUTA A  %
011     :       01 00 ;         % JMP 00  %
```

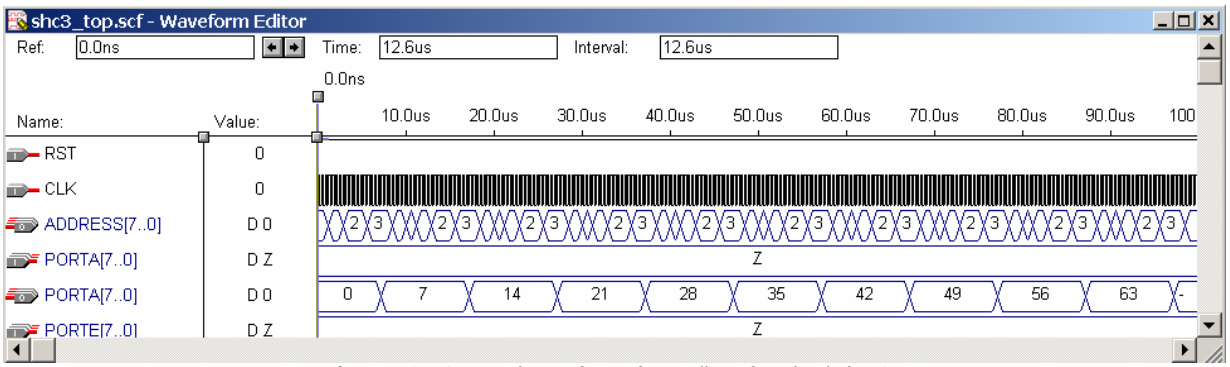


Figure 5.3c. Screen dump of waveform editor after simulation 2

Success, clearly 0 + 7 = 7 + 7 = 14 + 7 = 21 + 7 = 28 + 7 = 35 + …

Test 3: Decrement portA continuously by 3

The following program was entered into the program memory: -

```
000     :       A3 03 ;         % SUBA #3 %
010     :       02    ;         % OUTA A  %
011     :       01 00 ;         % JMP 00  %
```

Originally there was a problem, but after some time it was fixed, see appendix 5 for details on this problem and how it was fixed.

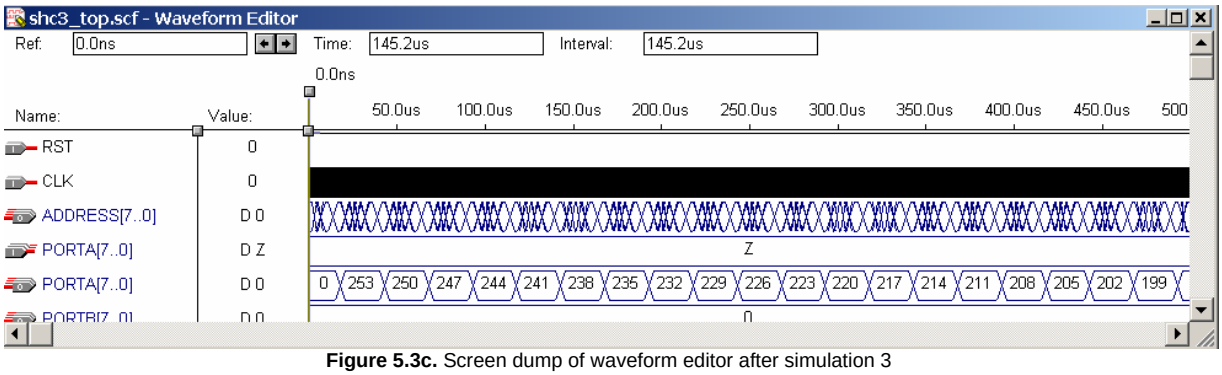


Figure 5.3c. Screen dump of waveform editor after simulation 3

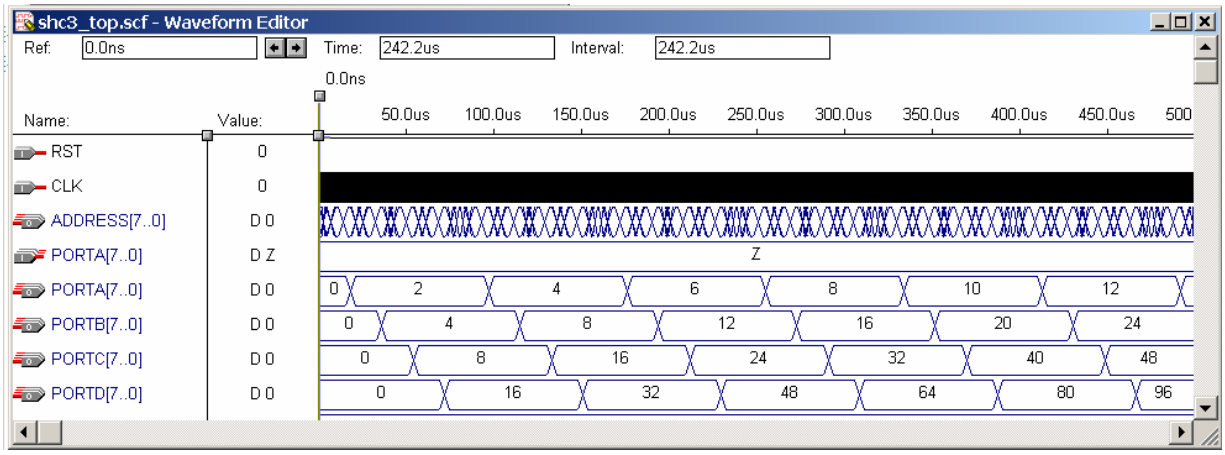
Success, clearly 253 – 3 = 250 – 3 = 247 – 3 = 244 – 3 = 238 – 3 = 235 – 3 = 232 – 3 = 229 - …

Test 4: Using Accumulators A to D, and Ports A to D

The following program was entered into the program memory: -

```
0 :  83 02 ; % ADDA #2  %
2 :  02    ; % OUTA A   %
3 :  84 04 ; % ADDB #4  %
5 :  07    ; % OUTB B   %
6 :  85 08 ; % ADDC #8  %
8 :  08    ; % OUTD C   %
9 :  86 10 ; % ADDD #16 %
B :  09    ; % OUTD D   %
C :  01 00 ; % JUP 00   %
F :  0F 0F 0F; %UNUSED%
```

Basically this program increments accumulator A by 2 which is outputted to PORTA, increments accumulator B by 4 which is outputted to PORTB, increments accumulator C by 8 which is outputted to PORTC and increments accumulator D by 16 which is outputted to PORTD. This process repeats forever.



Success, PORTA = 0 + 2 = 2 + 2 = 4 + 2 = 6 + 2 = 8 + 2 = 10 + …
PORTB = 0 + 4 = 4 + 4 + 8 + 4 = 12 + 4 = 16 + 4 = 20 + …
PORTC = 0 + 8 = 8 + 8 = 16 + 8 = 24 + 8 = 32 + 8 = 40 + …
PORTD = 0 + 16 = 16 + 16 = 32 + 16 = 48 + 16 = 64 + 16 = 80 + …

5.3.1. inputoutput.gdf (Eight 8-bit Bidirectional Ports)

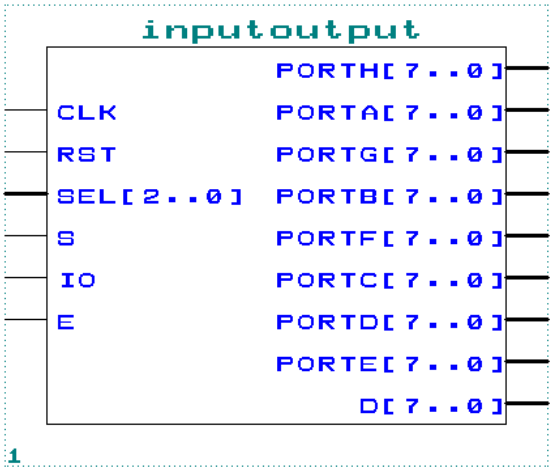


Figure 5.3.1a. inputoutput.gdf symbol

Test 1: Test all ports in output mode

IO = 0, E = 0, send data to all ports: -

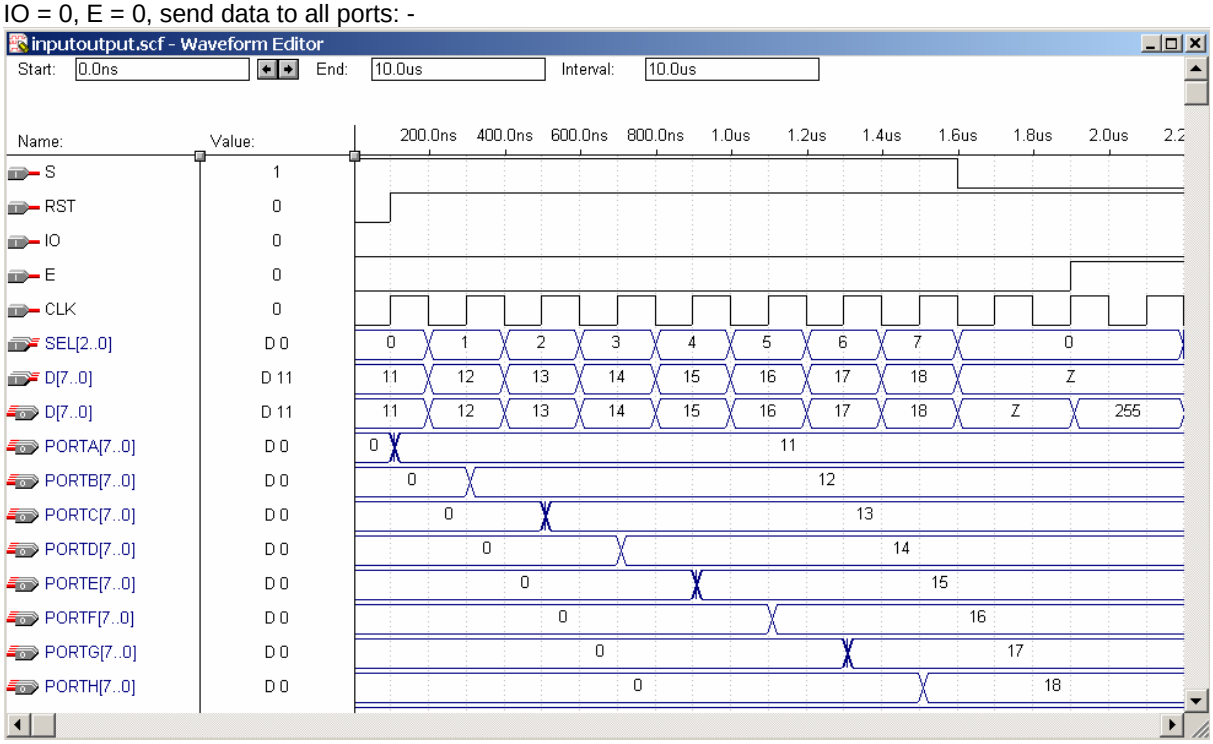


Figure 5.3.1b. Screen dump of waveform editor after simulation 1

Clearly a success, when SEL = 0, 11 was stored into portA, when SEL = 1, 12 was stored into portB, when SEL = 2, 14 was stored into portC, when SEL = 3, 15 was stored into portD, when SEL = 4, 16 was stored into portE, when SEL = 5, 16 was stored into portF, when SEL = 7, 17 was stored into portG, when SEL = 8, 18 was stored into portH.

Notice that when a port was not selected or the strobe line was low the ports held their existing value.

Test 2: Test all ports in input mode

IO = 1, E = 1, each input port given a unique value (101-108): -

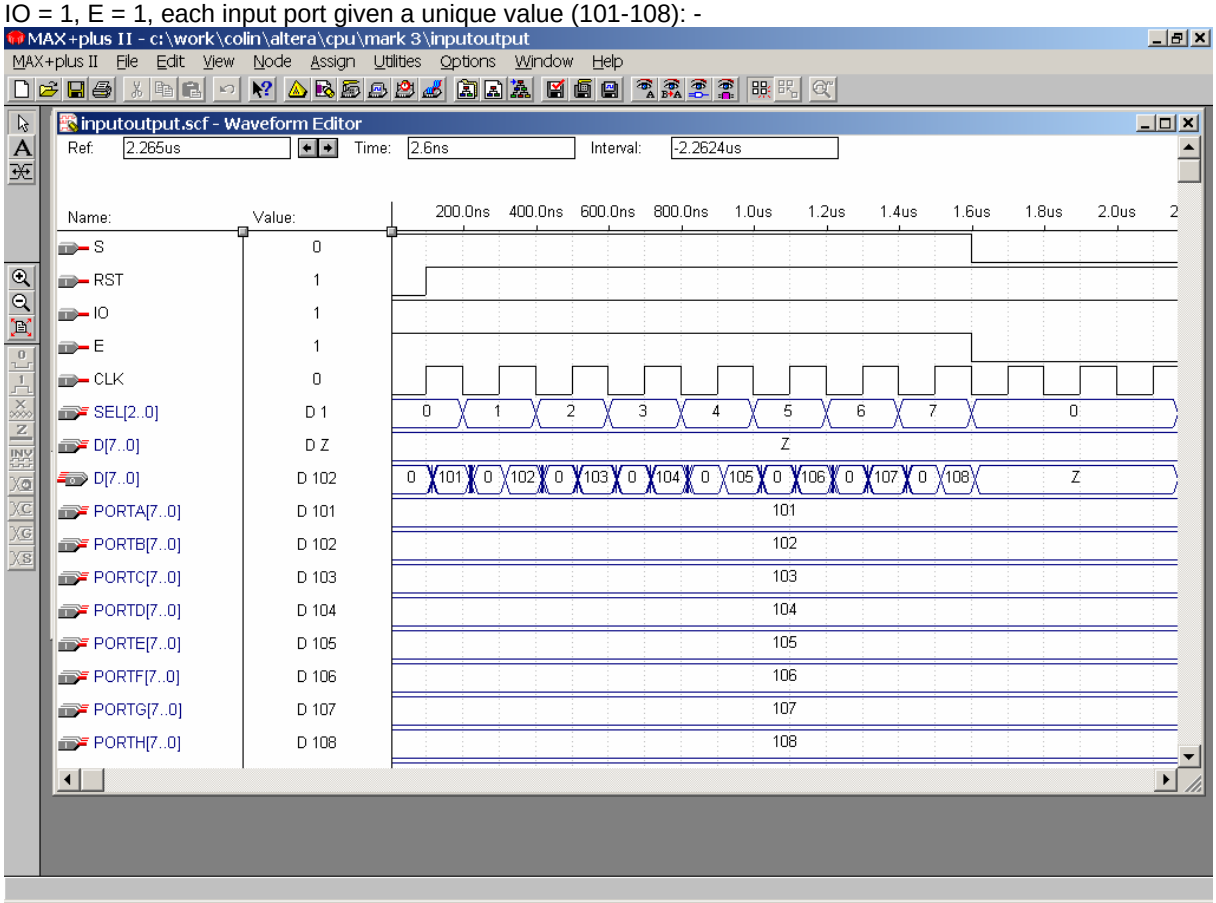


Figure 5.3.1c. Screen dump of waveform editor after simulation 2

It is clear that the preset value of each input port was successfully put onto the data bus.

5.3.2. bidport8.gdf (8-bit Bidirectional Port)

Figure 5.3.2a. bidport8.gdf symbol

Figure 5.3.2b shows the simulation waveforms, the first microsecond tests the port in output mode. Clearly this was a success as data bus data was successfully passed to the port when the strobe line was also high. Between 1.1µs and 2.2µs the port was in input mode, reading preset values on the port and placing them onto the data bus. Clearly this was also a success, e.g. 21,22,23,24 was placed onto the data bus.

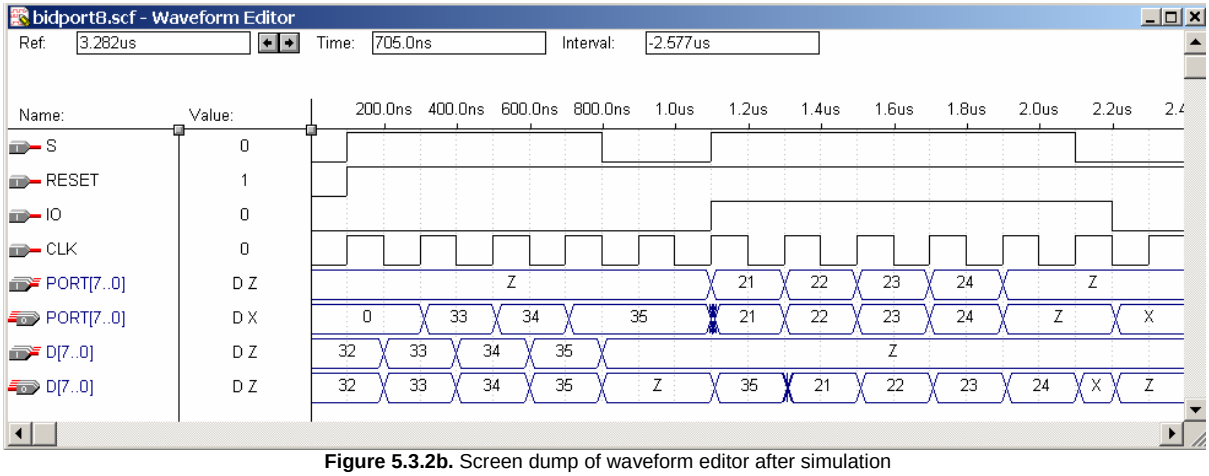


Figure 5.3.2b. Screen dump of waveform editor after simulation

5.3.3. regbank8.gdf (Register Bank of 8 Accumulators)

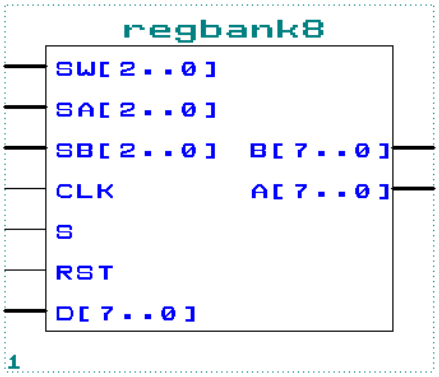


Figure 5.3.3a. regbank8.gdf symbol

First fill each register with a preset value (e.g. ACCA = 10, ACCB = 20, ACCH = 80), then run through all of the selections for output A and output B.

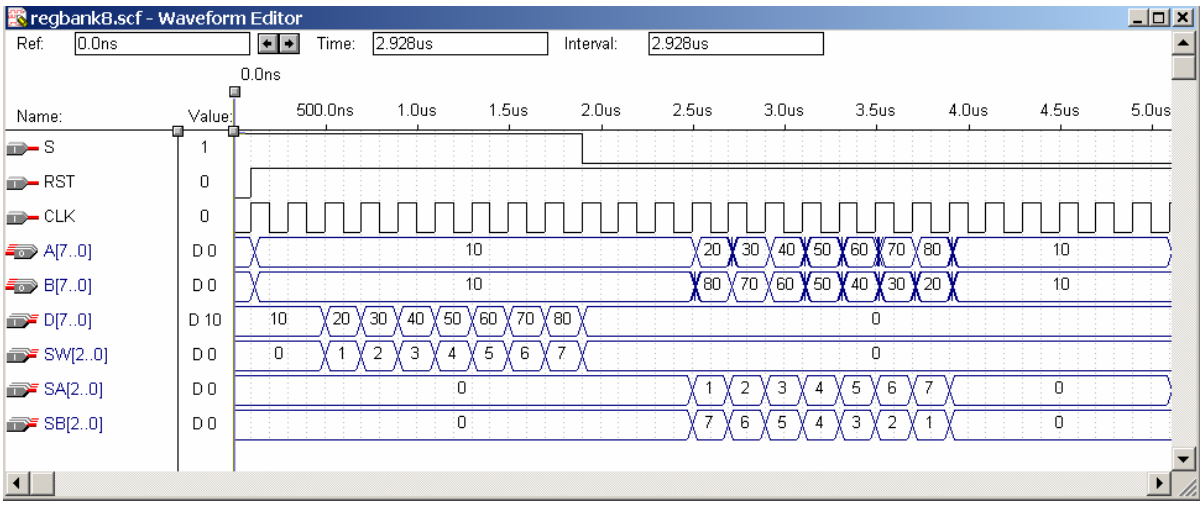


Figure 5.3.3b. Screen dump of waveform editor after simulation

Clearly working, selection line A counted from 0 to 7 and output A contained the correct values of each register (10 to 80). Selection line B counted from 7 to 0 and output B contained the correct values of each register (80 to 10).

5.3.4. acc.gdf (8-bit Accumulator)

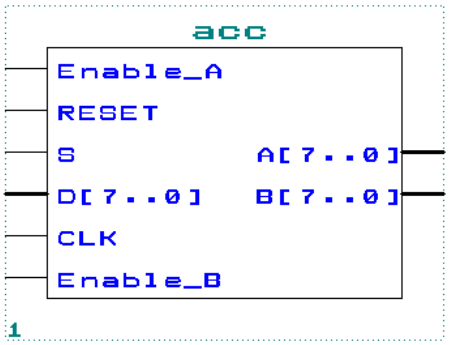


Figure 5.3.4a. acc.gdf symbol

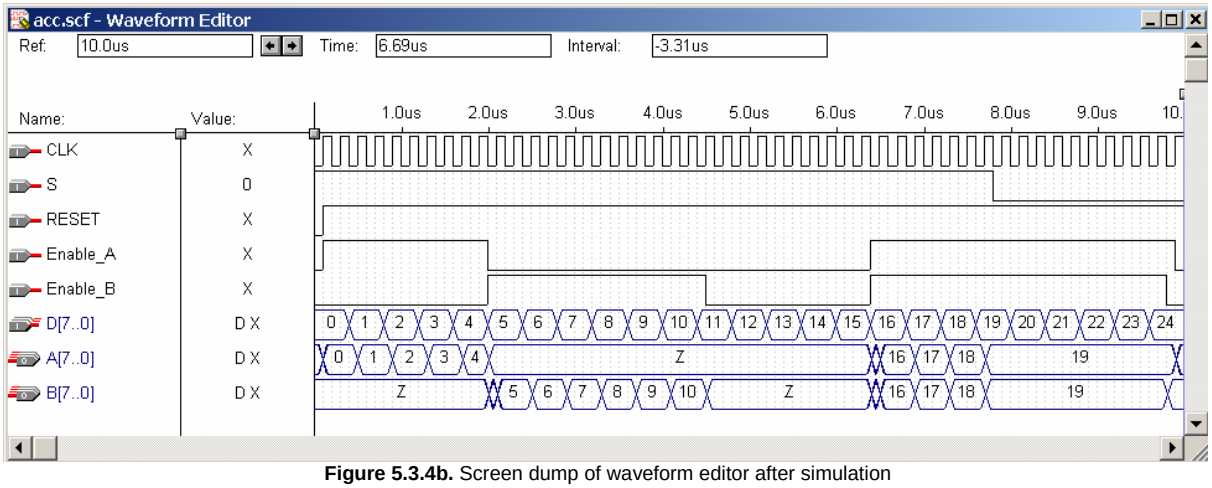


Figure 5.3.4b. Screen dump of waveform editor after simulation

Between 0 and 2µs the Enable_A line was set high, the simulation shows that output A[7..0] was connected to the output of the latch (e.g. 0,1,2,3,4) during this time, when Enable_A went low the output A was Z (high impendence). From 2µs to 4.5µs the Enable_B line was set high, the simulation shows that output B[7..0] was connected to the output of the latch (e.g. 5,6,7,8,9,10), notice when Enable_B was low the output B[7..0] was Z (high impendence).

Between 6.5µs and 10µs both Enable_A and Enable B were set high, the simulation shows that both the A[7..0] and B[7..0] outputs were connected to the output of the latch (e.g. 16,17,18,19). Notice when the strobe line went low, the latch held it old value (e.g. 19).

5.4. Mark 4 – Advanced 16-bit CPU

Note the design of this CPU is not completed due to the lack of time available, hence completed functional blocks are simulated.

5.4.1. alu16.gdf (Powerful 16-bit ALU)

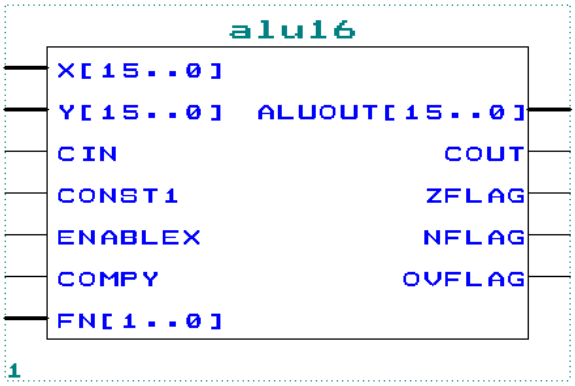


Figure 5.4.1a. ALU16.gdf symbol

X = 35425, Y = 2144, cycle through all four ALU functions: -

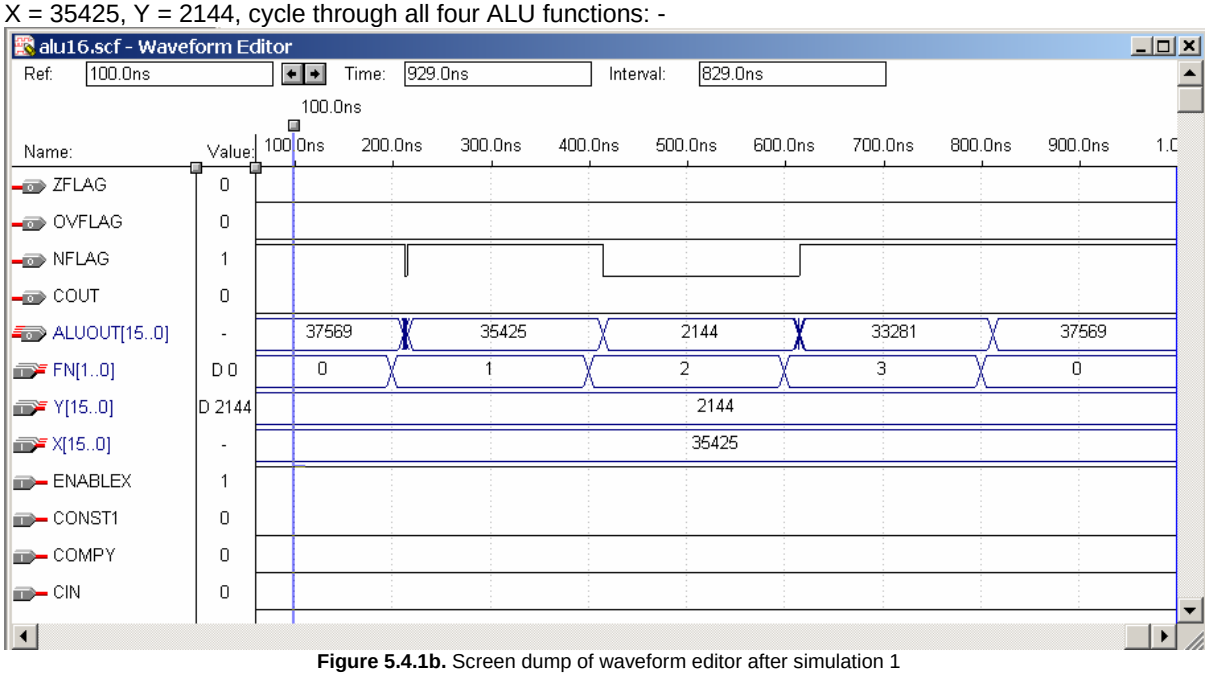


Figure 5.4.1b. Screen dump of waveform editor after simulation 1

When FN = 0, ALUOUT = 37569, that's X + Y (e.g. 2144 + 35425 = 37569)
When FN = 1, ALUOUT = 35425, that's X.
When FN = 2, ALUOUT = 2144, that's Y.
When FN = 3, ALUOUT = 33281, that's X – Y (e.g. 35425 – 2144 = 33281).

X = 35425, Y = 2144, CIN = 1, cycle through all four ALU functions: -

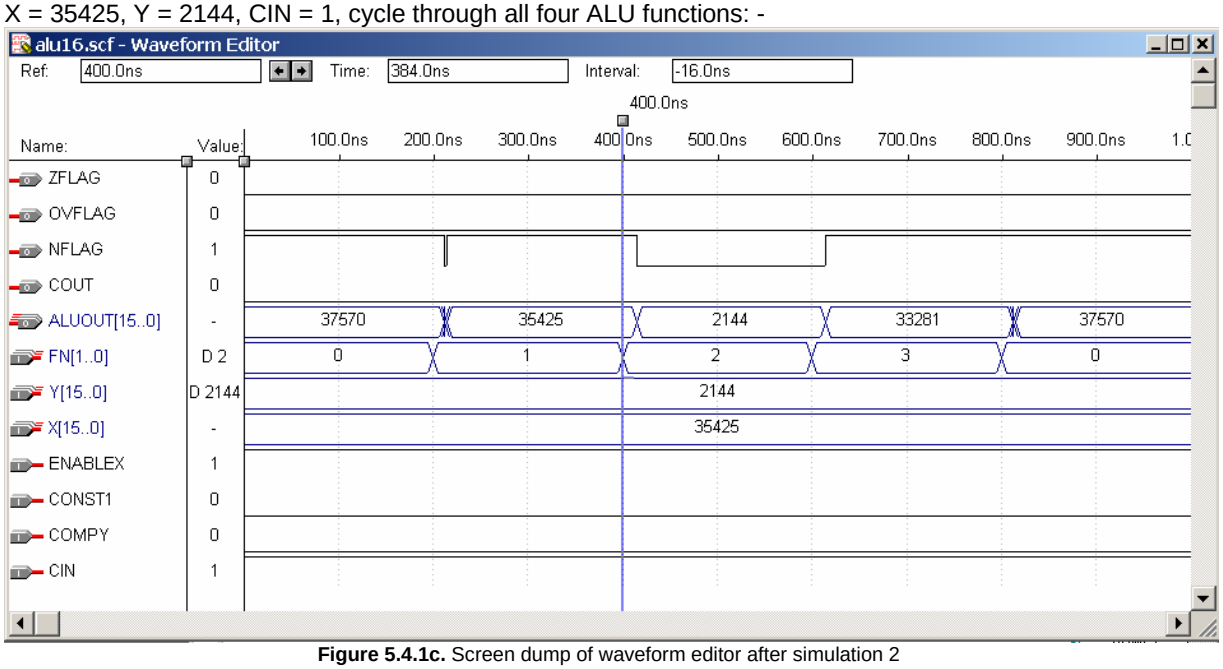


Figure 5.4.1c. Screen dump of waveform editor after simulation 2

When FN = 0, ALUOUT = 37570, that's X + Y + CIN (e.g. 2144 + 35425 + 1 = 37570)
When FN = 1, ALUOUT = 35425, that's X.
When FN = 2, ALUOUT = 2144, that's Y.
When FN = 3, ALUOUT = 33281, that's X – Y (e.g. 35425 – 2144 = 33281).

X = 35425, Y = 2144, CONST1 = 1, cycle through all four ALU functions: -

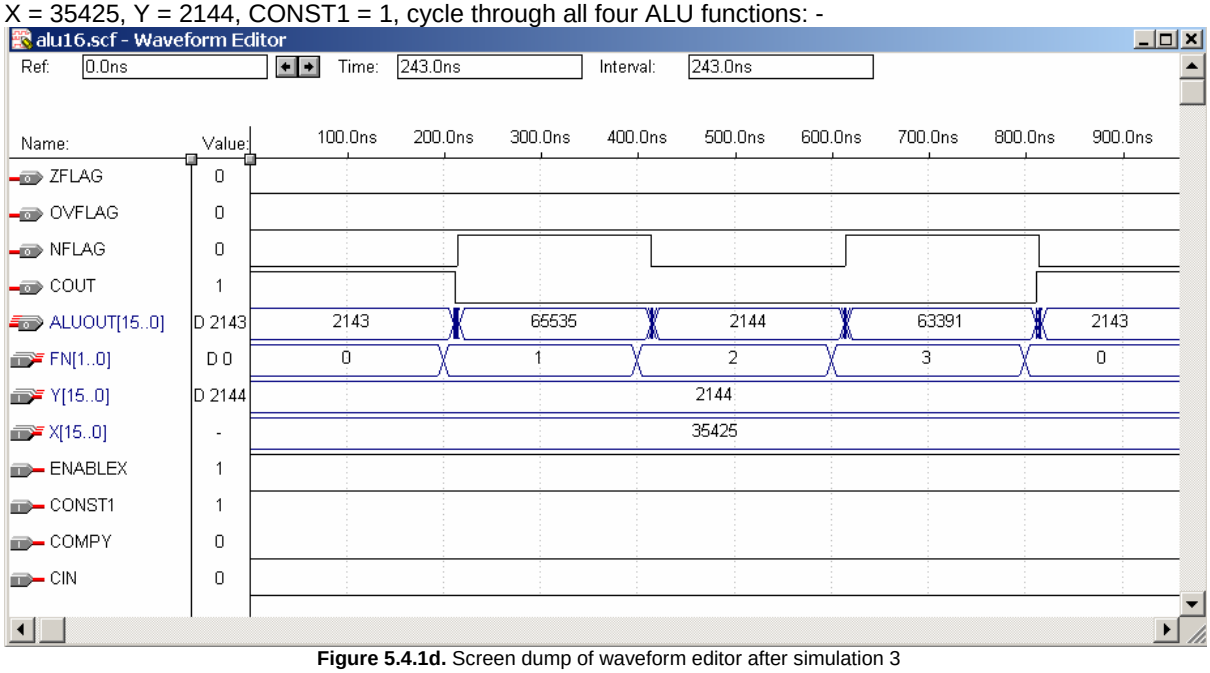


Figure 5.4.1d. Screen dump of waveform editor after simulation 3

When FN = 0, ALUOUT = 2143, that's X + Y (e.g. 2144 + 65535 = 67679, that's greater than the maximum 16-bit value, hence the answer is 2143 carry '1', e.g. 2143 = 0x085F, 2143 carry 1 = 0x1085F = 67679).
When FN = 1, ALUOUT = 65535, that's CONST1 (e.g. 0b1111111111111111).
When FN = 2, ALUOUT = 2144, that's Y.
When FN = 3, ALUOUT = 33281, that's X – Y (e.g. 65535– 2144 = 63391).

X = 35425, Y = 0, cycle through all four ALU functions: -

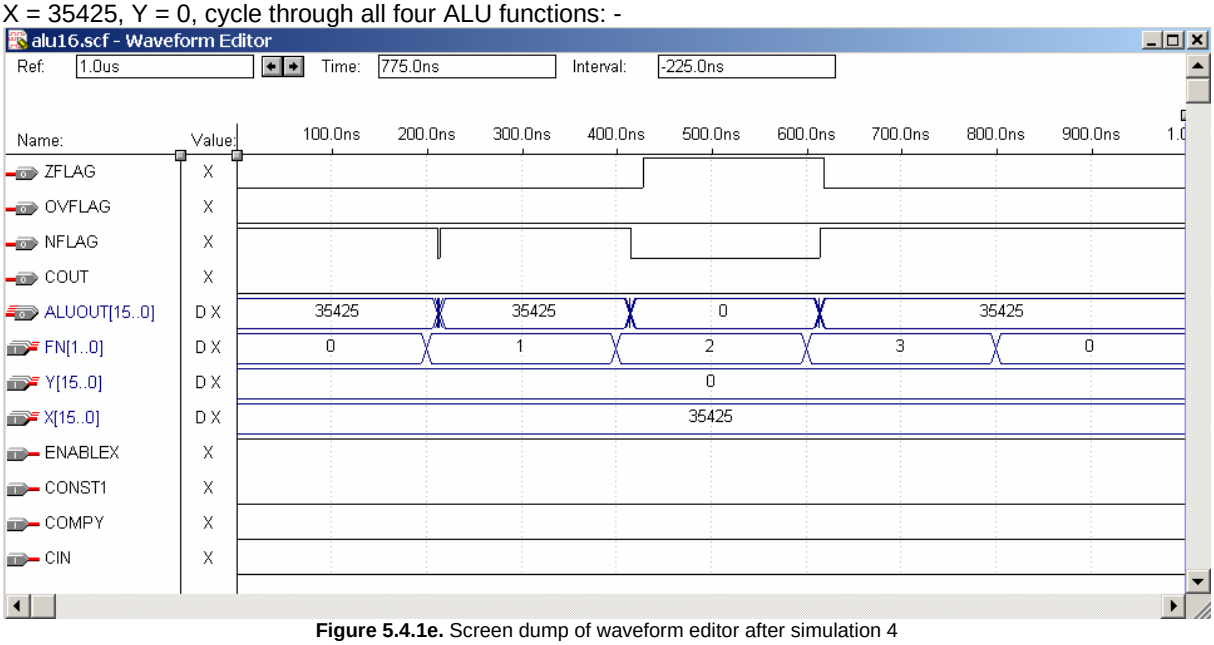


Figure 5.4.1e. Screen dump of waveform editor after simulation 4

When FN = 0, ALUOUT = 35425, that's X + Y (e.g. 0 + 35425 = 35425),
When FN = 1, ALUOUT = 35425, that's X.
When FN = 2, ALUOUT = 0, that's Y (notice the zero flag).
When FN = 3, ALUOUT = 35425, that's X – Y (e.g. 35425 – 0 = 35425).

5.4.2. register16.gdf (16-bit Register)

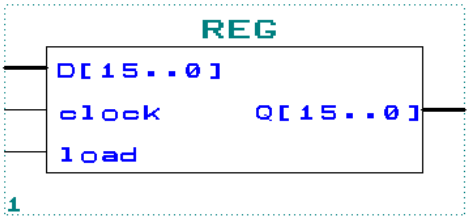


Figure 5.4.2a. register.gdf symbol

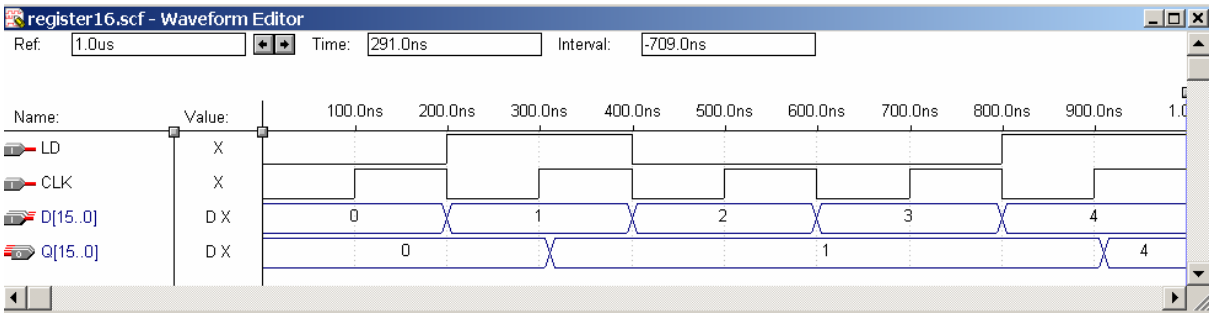


Figure 5.4.2b. Screen dump of waveform editor after simulation

Success, when LD went high the register loaded new value (e.g. 1,4), else it held its old value.

5.4.3. regmux16.gdf (16-bit Register Multiplexer)

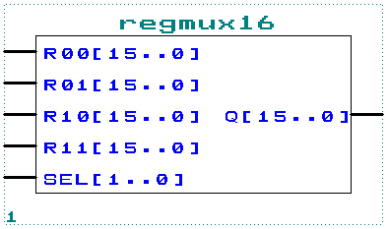


Figure 5.4.3a. regmux16.gdf symbol

R00 = 1000, R01 = 2000, R10 = 3000, R11 = 4000, cycle through all selections: -

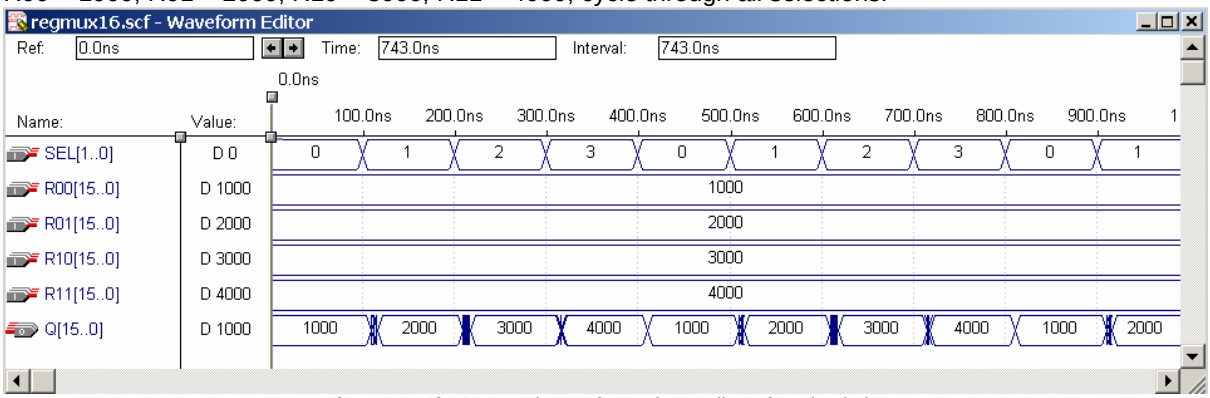


Figure 5.4.3b. Screen dump of waveform editor after simulation

Clearly working, as the output Q correctly displayed the value of the selected bus.

5.4.4. regsel.gdf (Four 16-bit Accumulators)

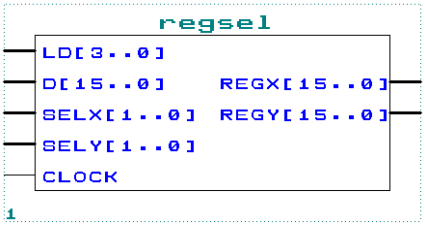


Figure 5.4.4a. regsel.gdf symbol

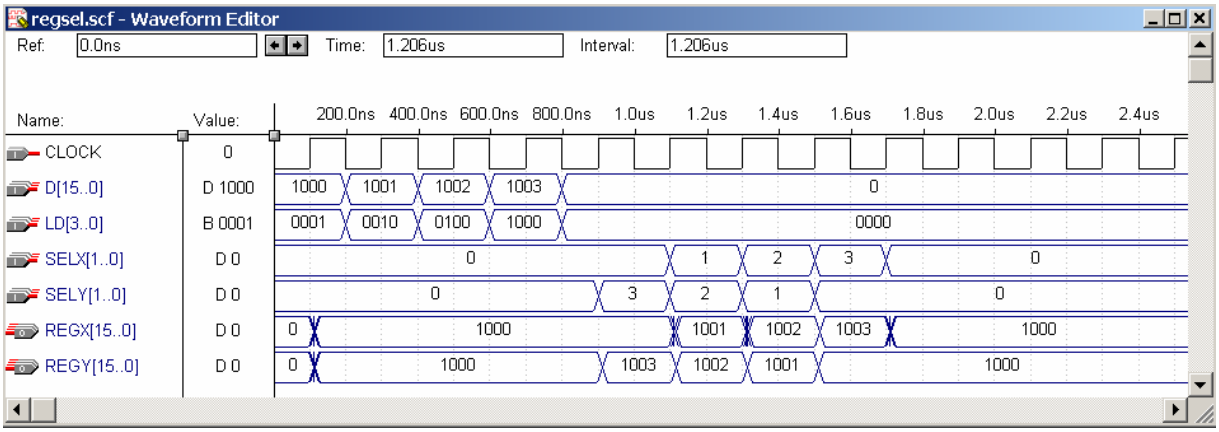


Figure 5.4.4b. Screen dump of waveform editor after simulation

Between 0 and 800ns, preset values are being loaded into each accumulator (e.g. A = 1000, B = 1001, C = 1002, D = 1003), then SELX is cycled from 0 to 3, notice on the REGX output during this time, the correct value of each accumulator was displayed. SELY is cycled from 3 to 0, notice on the REGY output during this time, the correct value of each accumulator was displayed.

6.0. CONCLUSIONS

The purpose of this assignment was to design a simple CPU that can execute a few instructions using the Altera Max2Plus package. Clearly this objective has been achieved successfully, as it has been proven using simulation the design works, although further development is required as there are limitations and flaws in the current design.

Clearly microprocessors have come a long way since they were first developed by Intel in the early 1970s (e.g. 4004, 2,300 transistors, see appendix 4). VLSI technologies are advancing at an incredible pace; transistors are getting smaller and smaller for example in 1988 1.2 million transistors were squeezed into one IC, but now it is possible to produce up-to 110 million transistors in a single IC, that's an increase of nearly 100 times in just 14 years. In fact it is expected for the transistor count to double every 2 years, for example in 1999 it was only possible to squeeze 42 millions transistors into a single IC, 2 years later it was possible to squeeze 110 million. Obviously modern PC microprocessor design is at the forefront of VLSI design; companies like AMD and Intel are the world leaders. Their latest chips may be a view years behind in terms of the transistor count (AMD latest XP processor has 37.5 Million transistors), but in terms of performance their at the forefront of VLSI / ASICS technology running at clock speeds above 2GHz, that just a view years ago was through to be impossible without using liquid nitrogen to cool the chip.

Obviously the design of the simple microprocessors in this report are using first generation (obsolete) technology and cannot compete with the modern microprocessor, as the modern microprocessor is much more powerful (32-bits / 64-bits) running at incredible clock speeds (above 2GHz), executing multipliable instructions simultaneously (e.g. AMDs QuantiSpeed technology) with specialised instructions sets (e.g. AMD 3D now, Intel MMX, etc…) designed to optimise common operations.

The simple microprocessors in this report are based on the classic "Von Neumann" CPU architecture, obviously this makes the design of the CPU simple as each device is connected to a common data and address bus. But in recent years with the development of powerful GUI operating systems (e.g. MS Windows) which requires an extremely fast CPUs to run smoothly, this architecture has become known as the "Von Neumann Bottleneck" because only one device can use the data bus at any one time (slow….). Traditionally the solution to this problem was to increase clock speeds (now beyond 2GHz), but since the success of the PIC microcontroller that uses the Harvard architecture (separate program memory and separate data memory with impendent buses) and a RISC (Reduce Instruction Set Computer), but each instruction executes faster. The next generation of microprocessors will use technology from the world of microcontrollers, and maybe eventually a fully-functional asynchronous microprocessor could be developed (reduced power consumption, increased speed, but difficult and perhaps impossible to design a working reliable CPU).

Each design (e.g. mark 1 - 4) was partitioned into many functional blocks; each block did one task that was easily described. Evidently this partitioning method of design made the design of the system extremely easy. Screen dumps of the design of each functional block were included in this report along with an English description on how each block operated. Notice comments are also included on the design files.

The mark 1 (Simple CPU) has many limitations: the program ROM is limited to 3 opcodes, the microcode ROM is limited to 64 locations, one accumulator, one output port, no input port and no RAM. Clearly this CPU has too many limitations to be a practical CPU, hence major improvements are required to design. Evidently it has been proven using simulation that this simple CPU does work and successfully executed three instructions. The top layout is cluttered making it difficult to understand exactly how the system works; obviously this is bad design practice and needs to be improved.

The arithmetic and logic unit (mark 1 – 3) can carry out one of 14 mathematical operations, but there are two operations that are not included, divide and multiply. Obviously it is possible to carry out these operations in software using successive subtraction for divide and successive addition for multiply, but this is slow as the program must loop several times in order to calculate the result. Clearly a hardware solution is better (e.g. Motorola 68000 has divide and multiply instructions), this can easily be achieved in Altera Max2plus using lpm_divide and lpm_multiply from the Altera lpm library (can easily be integrated into VHDL). Floating point division could be added, but this does not suit an 8-bit micro (e.g. IEEE-754 24-bit) and would be more suitable for powerful 32-bit micros. Allow a separate math coprocessor (e.g. floating point divider) unit could be developed for this task using two 24-bit registers for input (filled in 3 stages, e.g. 8-bit bus) and one 24-bit

register for output. Microcode could be written for the floating point division opcode, were the coprocessor unit latches are filled, and given a signal to start the calculation, which sends a signal back to the CPU when the calculation is finished, the result is then retrieved from the 24-bit output register and placed into 3 8-bit RAM locations.

Clearly the mark 2 (Improved CPU) is more powerful than the mark 1 as it can support up to 15 opcodes (actually much more than this) and has many unused control lines for future expansion. Evidently the Altera hardware is better suited to 8-bit ROM blocks; hence the reason for using two 8-bit ROMs for the microcode rather than one non-standard sized ROM. This CPU also has a RAM block, clearly this makes the CPU extremely powerful as complex calculations can be carried out using the RAM for storage of intermediate results, also a portion of the RAM could be reserved for use as a software stack, storing the program counter, when calling subroutines or interrupt routines.

It is clear that something very elegant has been done with the ALU control lines, instead of the control unit controlling the ALU function (as in mark 1), this is set in the program ROM (top 4-bits of operand), and fid directly into the ALU from the instruction register. Therefore each set of microcode can carry out every ALU function, for example the microcode for ADD immediate and SUB immediate is identity the only difference is the top four bits of the opcode instruction which specifies the ALU function. Therefore this CPU design can actually support many more than 15 opcode instructions, as up to 16 instructions (16 ALU functions) could be achieved per microcode instruction. Evidently this has been proven as 28 opcode exist using only three microcode instructions, some of which have been proven to operate using simulation. Obviously it is not practical to test every possible opcode, hence a view key opcodes were simulated, since most of the opcodes share microcode, it is likely the other opcodes will work as each ALU function was simulated separately and proven to work.

Obviously the mark 3 (Superior CPU) is much more powerful than the mark 2. The top layout has only four functional blocks as suggested by “Von Neumann” (memory, ALU, Control Unit and Input/Output). Clearly this simplifies the design dramatically; as it is clear immediately how the CPU works, unlike the mark 1 and mark 2 were some time had to be spent tracing wires. This demonstrates the power of the partitioning, as this CPU looks simpler than the mark 1 and mark 2, but is actually much more complicated. Clearly even the most complex of designs can be simplified dramatically if partitioned into small enough blocks and this is the key to the design of extremely complex CPUs.

The mark 3 has 8 bidirectional ports and 8 accumulators making this design far more useful than the mark 1 and mark 2. Notice 3 8-bit ROMs are required for the microcode; this is because the port select lines and accumulator select lines are control directly from the control unit. Clearly this is not a good idea and should be changed, for example the advantages of removing the ALU function lines from the control unit in the mark 2 are clear, micro instructions are reusable. Clearly at present many microcode instructions are required for full functionally of the CPU, for example outA A, outA B, outA C, outA D, outA E, outA F, outA G, outA H would require 8 independent microcode instructions for outputting every accumulator to port A, but outB A-H, outC A-H, outC A-H, outD A-H, outE A-H, outF A-H, outG A-H, outH A-H are also required that’s 256 microcode instructions. Clearly this is not practical and something clever has to be done with the port and accumulator control lines to reduce required microcode as was done for the ALU.

Clearly one option to reduce microcode is to using a latch to configure the accumulator bank and ports, this latch is connected to the data bus, hence the program code could configure the ports and accumulators (e.g. OUTA #, where # = 0 for accA, 1 for accB, 2 for accC, etc…) using one common microcode instruction. Another option is to use a 24-bit instruction register, where 8-bits are used for the microcode address (up to 256 micro instructions) and the top 16-bits are used to configure the ALU function (4-bits), select port (3-bits), select accumulator for OP1 (3-bits), select accumulator for OP2 (3-bit), select accumulator to write to (3-bits). Clearly this will reduce the amount of microcode required, as each opcode can carry out many operations, but this makes the CPU is 3-times slower as the data bus is only 8-bits wide, hence IR must be filled in three stages, another drawback is that programs will required more space (24-bit operand instead of 8-bit). It is critical that one of these solutions are implemented as the CPU is useless at present as full functionally is not possible. Allow over 100 opcode were listed, most of which are useless and many of the key opcodes required to write a practical program are missing, e.g. cannot get access to all ports from any accumulators (would required 256 micro instructions, not practical). Clearly this CPU has a complex instruction set, with probably over 300 opcodes, if the design was completed and had a full functional list of opcodes (i.e. branch if equal to zero, call subroutine, lots of addressing modes, bit test, shift left, shift right, etc…).

The design of the Mark 4 (Advanced 16-bit CPU) is not finished, but some key functional block were designed and simulated. Clearly the 16-bit ALU is much more powerful than the 8-bit ALU used in the marks 1-3. Obviously being 16-bits makes it more powerful, but the key aspect of the design is the additional flags which make this ALU extremely powerful, allowing the programmer to carry out complex calculations easily. Evidently the ALU has been proven to work using simulation, with all flags operating correctly. Clearly it makes since to add additional functions like divide / multiply, and perhaps floating point arithmetic. A bank of four 16-bit accumulators have also been design, and proven to work. Undoubtedly a lot of work still needs to be done in order to have a fully functional 16-bit CPU, but a good start has been made.

Clearly a simple CPU was successfully designed, analyses and proved using the Altera Max2Plus package. The system (mark 3) was successfully fitted into Altera's EPF10K20TC144-3; appendix 6 shows the pin layout. The system is actuate and reliable, but is a long way off being a practical useful CPU and much more development work is required. Overall (given the amount of time available) extremely good process was made.

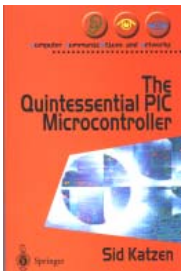
7.0. REFERENCES

Text Books

[B1] 

Computer Engineering Hardware Design

by M. Morris Mano
Published by Prentice Hall
ISBN: 0-13=162710-4
UUJLIB: 004.21MAN
1998

[B2] 

The Quintessential PIC Microcontroller

By Dr Sid Katzen
Published by Springer
ISBN: 1-85233-309-X.
2001

Websites

[W1] http://www.engj.ulst.ac.uk/sidk/ (microprocessor / microcontroller resource site, by Dr S.J. Katzen, University of Ulster)

[W2] http://www.intel.com (Intel's official website)

[W3] http://www.motorola.com (Motorola's official website)

[W4] http://www.ami.bolton.ac.uk/courseware/emsys/ch1/emsys01hist.html (A Brief History of Microprocessors).

[W5] http://www.hkrmicro.com/course/micro.html (A simple course on microprocessors)

[W6] http://www.vautomation.com/v8.htm (V8-µRISC^{tm} Microprocessor Core)

[W7] http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/html_single/CPU-Design-HOWTO.html (CPU Design HOW-TO by Alavoor Vasudevan)

[W8] http://www.fpga.org/ (FPGA CPU News)

[W9] http://www.gmvhdl.com/hc11core.html (GM HC11 CPU Core)

[W10] http://www.engin.umd.umich.edu/~nnarasim/courses/ece475/mano/ (Altera based CPU)

[W11] http://www.vautomation.com/Product.htm (Microprocessor Cores 8086 80186 8051 USB Ethernet CPU cores from ARC International's VAutomation product line)

[W12] http://www02.so-net.ne.jp/~morioka/cqpic.htm (VHDL PIC Core)

A1. 8-BIT COMMERCIAL CPU CORE

DIGITAL CORE DESIGN

DF6811CPU

8-bit FAST Microcontroller

ver 2.04

OVERVIEW

Document contains brief description of DF6811CPU core functionality. The DF6811CPU is a advanced 8-bit MCU IP Core with highly sophisticated, on chip peripheral capabilities. DF6811CPU soft core is binary-compatible with the industry standard 68HC11 8-bit microcontroller and can achieve a performance of up to **25 million instructions** per second in today's integrated circuit technologies. DF6811CPU has FAST architecture that is 3.8 times faster compared to original implementation.

Self-monitoring circuitry is included on-chip to protect against system errors. An illegal opcode detection circuit provides a non-maskable interrupt if illegal opcode is detected.

Two software-controlled power-saving modes, WAIT and STOP, are available to conserve additional power. These modes make the DF6811CPU IP Core especially attractive for automotive and battery-driven applications.

The DF6811CPU have built in the development support features designed into DF6811CPU. The LIR signal is intended as a debugging aid. This signal is driven to active low for the first bus cycle of each new instruction, making it easy to reverse assemble (disassemble) instructions from the display of a logic analyzer.

KEY FEATURES

- FAST architecture, 3,8 times faster than the original implementation
- Software compatible with industry standard 68HC11
- 10 times faster multiplication
- 16 times faster division
- 64 bytes of remapped System Function Registers space (SFRs)
- Up to 16M bytes of Data Memory
- De-multiplexed Address/Data Bus to allow easy connection to memory
- Interrupt Controller
  - *3 interrupt sources*
- Two power saving modes: STOP, WAI
- User programmable External Data Memory Write and Read pulses between 1 to 8 clock periods
- Fully synthesizable, static synchronous design with no internal tri-states
- No internal reset generator or gated clock
- Scan test ready
- 480 MHz virtual clock frequency compared to original implementation

http://www.DigitalCoreDesign.com
http://www.dcd.pl

DESIGN FEATURES

- ONE GLOBAL SYSTEM CLOCK
- SYNCHRONOUS RESET
  The DF6811CPU has 3 reset vectors sources, which easy identify a cause of system reset.
- ALL ASYNCHRONOUS INPUT SIGNALS ARE SYNCHRONIZED BEFORE INTERNAL USE
- DATA MEMORY:
  The DF6811CPU can address up to 16M bytes of Data Memory via the function interconnect signals. The 64 bytes of Data Memory in every 64k page is reserved for the Function Registers. Extra DPP (Data Page Pointer) register is used for segments swapping. Data Memory can be implemented as synchronous or asynchronous.
- SYSTEM FUNCTION REGISTERS:
  Up to 64 System Function Registers (SFRs) may be implemented to the DF6811CPU design. SFRs are memory mapped into Data Memory within any 64k bytes address space.

SPECIAL FEATURES

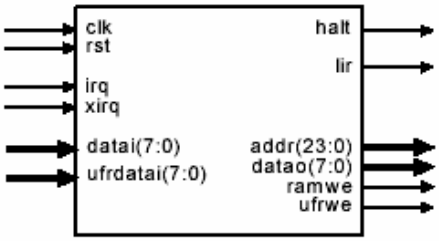
- Real time interrupt system (RTI)
- I/O Ports
- Synchronous serial peripheral interface system (SPI)
- Full-duplex UART system (SCI)
- 16 bits timer system includes 3 input capture and 5 output compare systems
- Watchdog system (COP)
- 8 bits pulse accumulator
- PWM Timer/Counter
- I2C master & slave bus controllers
  - *Master operation*
  - *Multi-master systems supported*
  - *Performs arbitration and clock synchronization*
  - *Interrupt generation*
  - *Supports speed up to 3,4Mb/s (standard, fast & HS modes)*
  - *Allows operation from a wide range of clock frequencies (build-in 8-bit timer)*
  - *User-defined timing*
- Floating-Point arithmetic coprocessor IEEE-754 standard single precision
  - *FADD, FSUB - addition, subtraction*
  - *FMUL, FDIV- multiplication, division*
  - *FSQRT- square root*
  - *FUCOM - compare*
  - *FCHS - change sign*
  - *FABS - absolute value*
- Floating-Point math coprocessor - IEEE-754 standard single precision real, word and short integers
  - *FADD, FSUB- addition, subtraction*
  - *FMUL, FDIV- multiplication, division*
  - *FSQRT- square root*
  - *FUCOM- compare*
  - *FCHS - change sign*
  - *FABS - absolute value*
  - *FSIN, FCOS- sine, cosine*
  - *FPTAN, FPATAN- tangent, arcs tangent*

DELIVERABLES

- Source code:
  - VHDL Source Code or/and
  - VERILOG Source Code or/and
  - ALTERA's Megafunction or/and
  - EDIF netlist
- VHDL & VERILOG test bench environment
  - Active-HDL automatic simulation macros
  - ModelSim automatic simulation macros
  - Tests with reference responses
- Technical documentation
  - Installation notes
  - HDL core specification
  - Datasheet
- Synthesis scripts
- Example application
- Technical support
  - IP Core implementation support
  - 3 months maintenance
    - Delivery the IP Core updates, minor and major versions changes
    - Delivery the documentation updates
    - Phone & email support

http://www.DigitalCoreDesign.com
http://www.dcd.pl

SYMBOL

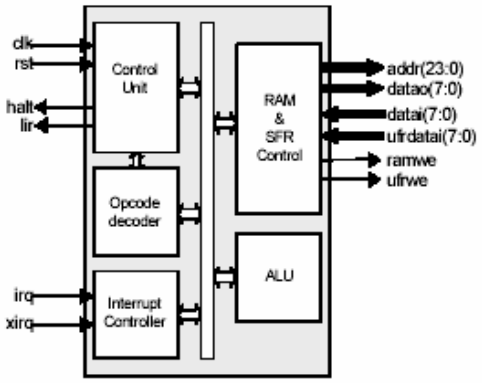


PINS DESCRIPTION

PIN	ACTIVE	TYPE	DESCRIPTION
clk	-	input	Global system clock
rst	Low	input	Global system reset
datai[7:0]	-	input	External memory bus input
ufrdatai[7:0]	-	input	UFRs data bus input
irq	*	input	Interrupt input
xirq	Low	input	Non-maskable interrupt input
addr[23:0]	-	output	Data memory & FR address bus
datao[7:0]	-	output	Data memory & UFR bus output
lir	Low	output	Load instruction register
ramwe	Low	output	External memory write enable
ufrwe	Low	output	UFRs write enable
halt	High	output	Halt clock system (STOP inst.)

* Kind of activity is configurable

BLOCK DIAGRAM



ALU - Arithmetic Logic Unit performs the arithmetic and logic operations during execution of an instruction. It contains accumulator (A, B), Condition Code Register (CCREG), and related logic like arithmetic unit, logic unit, multiplier and divider.

Control Unit - Performs the core synchronization and data flow control. This module manages execution of all instructions.

Opcode Decoder - Performs an instruction opcode decoding and the control functions for all other blocks.

Memory and SFR's Controller - Data Memory & SFR's (Special Function Register) interface controls access into the internal and external program and data memories and special registers. It contains Stack Pointer (SP) register, INIT register (INIT), Data Page Pointer (DPP), Stretch register (ST) and related logic.

Interrupt Controller - Interrupt Control module is responsible for the interrupt manage system for the external and internal interrupt sources.

PERFORMANCE

The following table gives a survey about the DF6811CPU performance in the ALTERA® and XILINX devices after Place & Route (all key features have been included):

Device	Speed grade	Logic Cells	Performance
APEX20KE	-1	2009	45 MHz
ACEX1K	-1	1967	44 MHz
FLEX10KE	-1	1999	44 MHz

Device	Speed grade	CLB Slices	Performance
VIRTEX-II	-5	1045	53 MHz
VIRTEX-E	-8	1044	47 MHz
VIRTEX	-6	1045	38 MHz

For user the most important is application speed improvement. The most commonly used arithmetic functions and theirs improvement are shown in table below. Improvement was computed as {M68HC11 clock periods} divided by {DF6811CPU clock periods} required to execute an identical function. More details are available in core documentation

Function	Improvement
8-bit addition (*immediate data*)	4
8-bit addition (*direct addressing*)	4
8-bit subtraction (*immediate data*)	4
8-bit subtraction (*direct addressing*)	4
16-bit addition (*immediate data*)	5,3
16-bit addition (*direct addressing*)	5
16-bit addition (*indirect addressing*	4,8
16-bit subtraction (*immediate data*)	5,3
16-bit subtraction (*direct addressing*)	5
16-bit subtraction (*indirect addressing*	4,8
Multiplication	10
Fractional division	14,9
Integer division	16.4

http://www.DigitalCoreDesign.com
http://www.dcd.pl

A2. 16-BIT/32-BIT COMMERCIAL CPU CORE

DIGITAL CORE DESIGN

D68000

16/32-bit Microprocessor

ver 1.10

OVERVIEW

D68000 soft core is binary-compatible with the industry standard 68000 32-bit microcontroller. D68000 has a 16-bit data bus and 24-bit address data bus. It is code-compatible with the MC68008 and is upward code compatible with the MC68010 virtual extensions and the MC68020 32-bit implementation of the architecture. D68000 has improved instructions set allows execution of a program with higher performance than standard 68000 core.

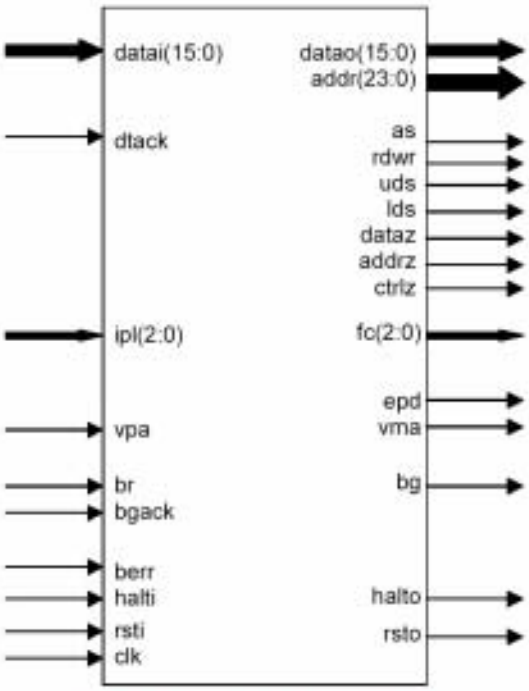
KEY FEATURES

- Software compatible with industry standard 68000
- **MULS, MULU** take **28 clock periods**
- **DIVS, DIVU** take **28 clock periods**
- Optimized shifts and rotations
- Idle cycles removed to improve performance
- Shorter effective address calculation time
- Bus cycle timings **identical** to 68000
- 32 bit data and address registers
- 14 addressing modes:
  - *Direct:*
    - *Data register direct*
    - *Address register direct*
  - *Indirect:*
    - *Register indirect*
    - *Postincrement register indirect*
    - *Predecrement register indirect*
    - *Register indirect with offset*
    - *Indexed register indirect with offset*
  - *PC relative:*
    - *Relative with offset*
    - *Relative with index and offset*
  - *Absolute data:*
    - *Absolute short*
    - *Absolute long*
  - *Immediate data:*
    - *Immediate*
    - *Quick immediate*
  - *Implied*
- 5 data types supported:
  - *bits*
  - *BCD*
  - *bytes, words and long words*
- Arithmetic Logic Unit includes:
  - *8,16,32-bit arithmetic & logical operations*
  - *16x16 bit signed and unsigned multiplication*
  - *32/16 bit signed and unsigned division*
  - *Boolean operations*
- Interrupt controller:
  - *7 priority levels interrupt controller*
  - *Unlimited number of virtual interrupt sources*
  - *Vectored and auto-vectored modes*

http://www.DigitalCoreDesign.com
http://www.dcd.pl

- Memory interface includes:
  - *Up to 4 GB of address space*
  - *16-bit data bus*
  - *Asynchronous bus control*
- M6800 family synchronous interface
- 3- and 2- wire bus arbitration
- Supervisor and user modes
- Fully synthesizable, static synchronous design with no internal tri-states

SYMBOL



DELIVERABLES

- Source code:
  - VHDL Source Code or/and
  - VERILOG Source Code or/and
  - ALTERA's Megafunction or/and
  - EDIF netlist
- VHDL & VERILOG test bench environment
  - Active-HDL automatic simulation macros
  - ModelSim automatic simulation macros
  - Tests with reference responses
- Technical documentation
  - Installation notes
  - HDL core specification
  - Datasheet
- Synthesis scripts

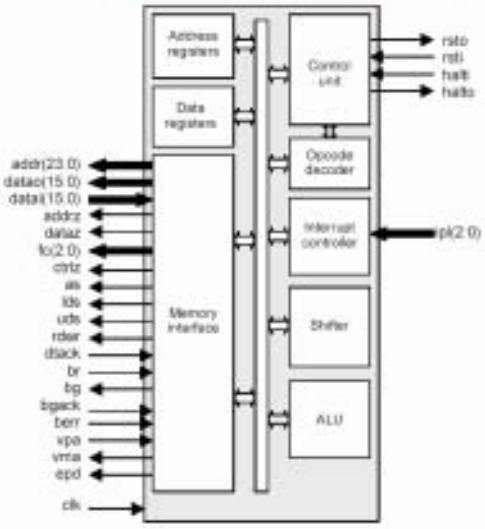
- Example application
- Technical support
  - IP Core implementation support
  - 3 months maintenance
    - Delivery the IP Core updates, minor and major versions changes
    - Delivery the documentation updates
    - Phone & email support

PINS DESCRIPTION

PIN	TYPE	ACTIVE	DESCRIPTION
clk	input	High	Global clock
rsti	input	Low	Global reset input
halti	input	Low	Halt input
berr	input	Low	Bus error
vpa	input	Low	Valid peripheral address
ipl(2:0)	input	Low	Interrupt control
dtack	input	Low	Data transfer acknowledge
br	input	Low	Bus request
bgack	input	Low	Bus grant acknowledge
datai[15:0]	input	-	Data bus input
datao[15:0]	output	-	Data bus output
addr[23:0]	output	-	Address data bus
bg	output	Low	Bus grant
as	output	Low	Address strobe
rdwr	output	High/Low	Read write signal
uds	output	Low	Upper data byte strobe
lds	output	Low	Lower data byte strobe
addrz	output	High	Turns Address bus into 'Z' state
dataz	output	High	Turns Data bus into 'Z' state
ctrlz	output	High	Turns as, rdwr, uds, lds, vm, fc(2:0) signals into 'Z' state
fc(2:0)	output	High	Processor function code
epd	output	High	Enable peripheral device
vma	output	Low	Valid memory address
halto	output	Low	Halt output
rsto	output	Low	Reset output

http://www.DigitalCoreDesign.com
http://www.dcd.pl

BLOCK DIAGRAM



ALU – Arithmetic Logic Unit performs the arithmetic and logic operations during execution of an instruction. It contains accumulator and related logic such as arithmetic unit, logic unit, multiplier and divider. BCD operation are executed in this unit and condition code flags (N-negative, Z-zero, C-carry V-overflow) for most instructions.

Shifter – Performs shifting operations for the appropriate instructions, mainly for rotation, shift and bit operations.

Control Unit – Performs the core synchronization and data flow control. This module manages execution of all instructions. Contains SR (status register is consisted of two portions supervisor byte and user byte) and its related logic.

Opcode Decoder – Performs an instruction opcode decoding and the control functions for all other blocks.

Memory Interface – Contains memory access related registers It performs the memory addressing instructions code fetching and data transfers. It is responsible for all external bus cycle actions such as: read & write, repeated read & write, halt and resume of bus cycles, bus arbitration provided by 3- and 2- wire system, correct bus and address errors handling, wait states cycle insertion and M6800 synchronous cycle generation.

Interrupt Controller – Interrupt Control module is responsible for the interrupt manage system for the external & internal interrupts and exceptions processing. It manages auto-vectored interrupt cycles, priority resolving and correct vector numbers creation.

Address registers – Contains 32-bit A0 to A6 address registers, two stack pointers USP (user SP) and SSP (Supervisor SP), 32-bit Program counter and related logic to perform word and long address operations. An effective address operation are executed in this unit.

Data registers – Contains 32-bit data registers D0 to D7 and related logic to perform byte, word and long data operations.

PERFORMANCE

The following tables give a survey about the D68000 area and performance in the ALTERA® and XILINX® devices after Place & Route (all key features have been included):

Device	Speed grade	Logic Cells	Performance
APEX20KE	-1	6560	35 MHz
	-2	6560	29 MHz
	-3	6560	23 MHz
APEX20K	-1	6560	32 MHz
	-2	6560	28 MHz
	-3	6560	21 MHz

D68000 performance in ALTERA® devices

Device	Speed grade	CLB Slices	Performance
VIRTEX-II	-5	3070	56 MHz
	-4	3070	44 MHz
VIRTEX-E	-8	3426	41 MHz
	-7	3426	35 MHz
VIRTEX	-6	3426	32 MHz
	-5	3426	26 MHz

D68000 performance in XILINX® devices

http://www.DigitalCoreDesign.com
http://www.dcd.pl

A2. 8-BIT COMMERCIAL PIC MICROCONTROLLER CORE

DIGITAL CORE DESIGN

DFPIC165X

8-bit Fast RISC Microcontroller

ver 1.22

OVERVIEW

The DFPIC165X is a low-cost, high performance, 8-bit, fully static IP Core, binary-compatible with the industry standard PIC16C54, PIC16C55, PIC16C56, PIC16C57 and PIC16C58. It employs a modified RISC architecture (2 times faster than original implementation). The DFPIC165X fits perfectly in applications ranging from high-speed automotive and appliance motor control to low-power remote transmitters/receivers, pointing devices and telecom processors. Built-in power save mode make this IP perfect for applications where power consumption is critical. The small used area in programmable devices make this IP ideal applications with space limitations. Low-cost, low-power, high performance and ease of use make the DFPIC165X very versatile even in areas where no microcontroller use has been considered before.

DFPIC165X can address Data Memory of up to 128 bytes, and program space up to 2k words. The Data Memory is implemented as Single-Port RAM.

KEY FEATURES

- Software compatible with industry standard PIC16C5X
- 33 instructions
- Modified ~2 times faster architecture
- 12 bit wide instruction word
- Up to 128 bytes of internal Data Memory
- Up to 2K words of Program Memory
- 2 level deep hardware stack
- 8-bit timer/counter with programmable prescaler
- Three I/O ports
- Programmable Watchdog Timer with its own CLK input for reliable operations
- Power saving SLEEP mode
- Direct, indirect and relative addressing modes
- Fully synthesizable, static synchronous design with no internal tri-states
- Over 400 MHz in a 0.35u technological process

SPECIAL FEATURES

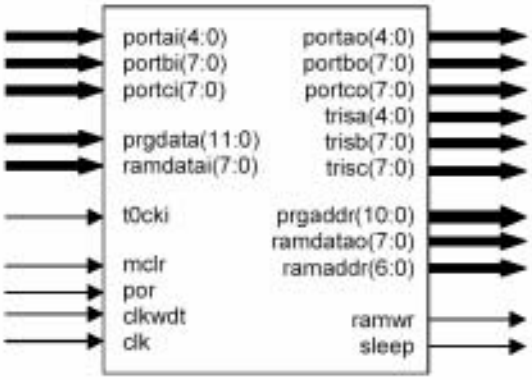
- I2C bus controller
- SPI bus controller
- PWM Pulse/Width Modulation Timer
- UART
- External or internal interrupts
- 256 Bytes of Data Memory
- 4 k words of Program memory

http://www.DigitalCoreDesign.com
http://www.dcd.pl

DELIVERABLES

- Source code:
  - VHDL Source Code or/and
  - VERILOG Source Code or/and
  - ALTERA's Megafunction or/and
  - EDIF netlist
- VHDL & VERILOG test bench environment
  - Active-HDL automatic simulation macros
  - ModelSim automatic simulation macros
  - Tests with reference responses
- Technical documentation
  - Installation notes
  - HDL core specification
  - Datasheet
- Synthesis scripts
- Example application
- Technical support
  - IP Core implementation support
  - 3 months maintenance
    - Delivery the IP Core updates, minor and major versions changes
    - Delivery the documentation updates
    - Phone & email support

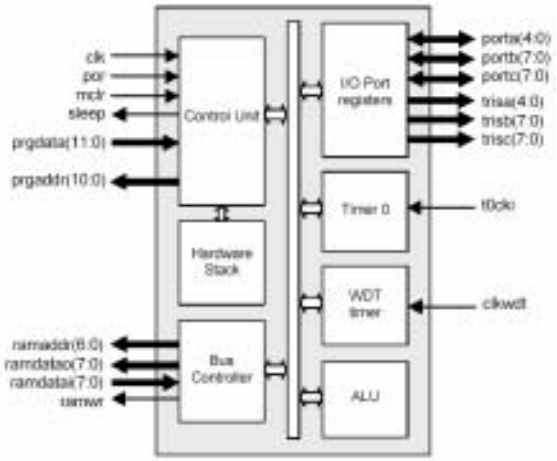
SYMBOL



BLOCK DIAGRAM

Figure below shows the DFPIC165X IP Core block diagram.

ALU – Arithmetic Logic Unit performs arithmetic and logic operations during execution of an instruction. This module contains work register (W) and Status register.



Control Unit – It performs the core synchronization and data flow control. This module manages execution of all instructions. Performs decode and control functions for all other blocks. It contains program counter (PC) and hardware stack.

Bus Controller – It performs interface functions between Data memory and DFPIC165X internal logic. It assures correct Data memory addressing and data transfers.

Hardware Stack - Configurable deep of stack according to programmer needs. By default 2-level hardware stack is automatically used by microcontroller.

Timer 0 – Main system's timer and prescaler. It is 8-bit timer/counter register configurable by OPTION register.

WDT - The watchdog timer is a free running timer. WDT has own clock input separate from system clock. It means that WDT will run even if the system clock is stopped by SLEEP instruction.

I/O – Block contains DFPIC165X's general purpose I/O ports and data direction register (TRIS). Each port's bit can be individually accessed by bit addressable instructions. Each port's pin has an corresponding bit of TRIS register. When a bit of TRIS register is set this means that the corresponding pin of port is configured as an input.

http://www.DigitalCoreDesign.com
http://www.dcd.pl

PINS DESCRIPTION

PIN	TYPE	DESCRIPTION
clk	input	Global clock
clkwdt	input	Watchdog clock
por	input	Global reset Power On Reset
mclr	input	User reset
portai[4:0]	input	Port A input
portbi[7:0]	input	Port B input
portci[7:0]	input	Port C input
prgdata[11:0]	input	Data bus from program memory
ramdata[7:0]	input	Data bus from int. data memory
t0cki	input	Timer 0 input
portao[4:0]	output	Port A output
portbo[7:0]	output	Port B output
portco[7:0]	output	Port C output
trisa[4:0]	output	Data direction pins for Port A
trisb[7:0]	output	Data direction pins for Port B
trisc[7:0]	output	Data direction pins for Port C
prgaddr[10:0]	output	Program memory address bus
sleep	output	Sleep signal
ramaddr[6:0]	output	RAM address bus
ramdatao[7:0]	output	Data bus for internal data memory
ramwr	output	Data memory write

PERFORMANCE

The following tables give a survey about the DFPIC165X area and performance in the ALTERA® and XILINX® devices after Place & Route (all key features have been included):

Device	Speed grade	Logic Cells[1]	Performance
APEX20KE	-1	540+3ESB	92 MHz
	-2	540+3ESB	76 MHz
	-3	540+3ESB	60 MHz
ACEX1K	-1	534+3EAB	77 MHz
	-2	534+3EAB	50 MHz
FLEX10KE	-1	546+3EAB	78 MHz
	-2	546+3EAB	50 MHz

DFPIC165X performance in ALTERA® devices

[1]- 128 Bytes data memory and 512 words program memory

Device	Speed grade	CLB Slices	Performance
VIRTEX-II	-5	274	139 MHz
	-4	274	126 MHz
VIRTEX-E	-8	276	103 MHz
	-7	276	92 MHz
VIRTEX	-6	276	80 MHz
	-5	276	69 MHz
SPARTAN-II	-6	276	90 MHz
	-5	276	79 MHz

DFPIC165X performance in XILINX® devices

The main features of each DFPIC family member have been summarized in table below. It gives a briefly member characterization helping user to select the most suitable IP Core for its application. User can specify its own peripheral set (including listed below and the others) and requests the core modifications.

Design	Program Memory space	Data Memory space	Program word length	Number of instructions	I/O Ports	Timer 0	Watchdog Timer	Sleep Mode	External interrupts	Internal Interrupts	Levels of hardware stack	Wake up on port pin change	Speed rate
DFPIC125X	1k	128	12	33	6	✓	✓	✓	-	-	2	✓	2
DFPIC165X	2k	128	12	33	21	✓	✓	✓	-	-	2	-	2
DFPIC1655X	8k	256	14	35	13	✓	✓	✓	4	1	8	✓	2

DFPIC family of High Performance Microcontroller Cores

http://www.DigitalCoreDesign.com
http://www.dcd.pl

A4. MICROPROCESSOR GENERATIONS

A4.1. In the beginning (8-bit) Intel 4004

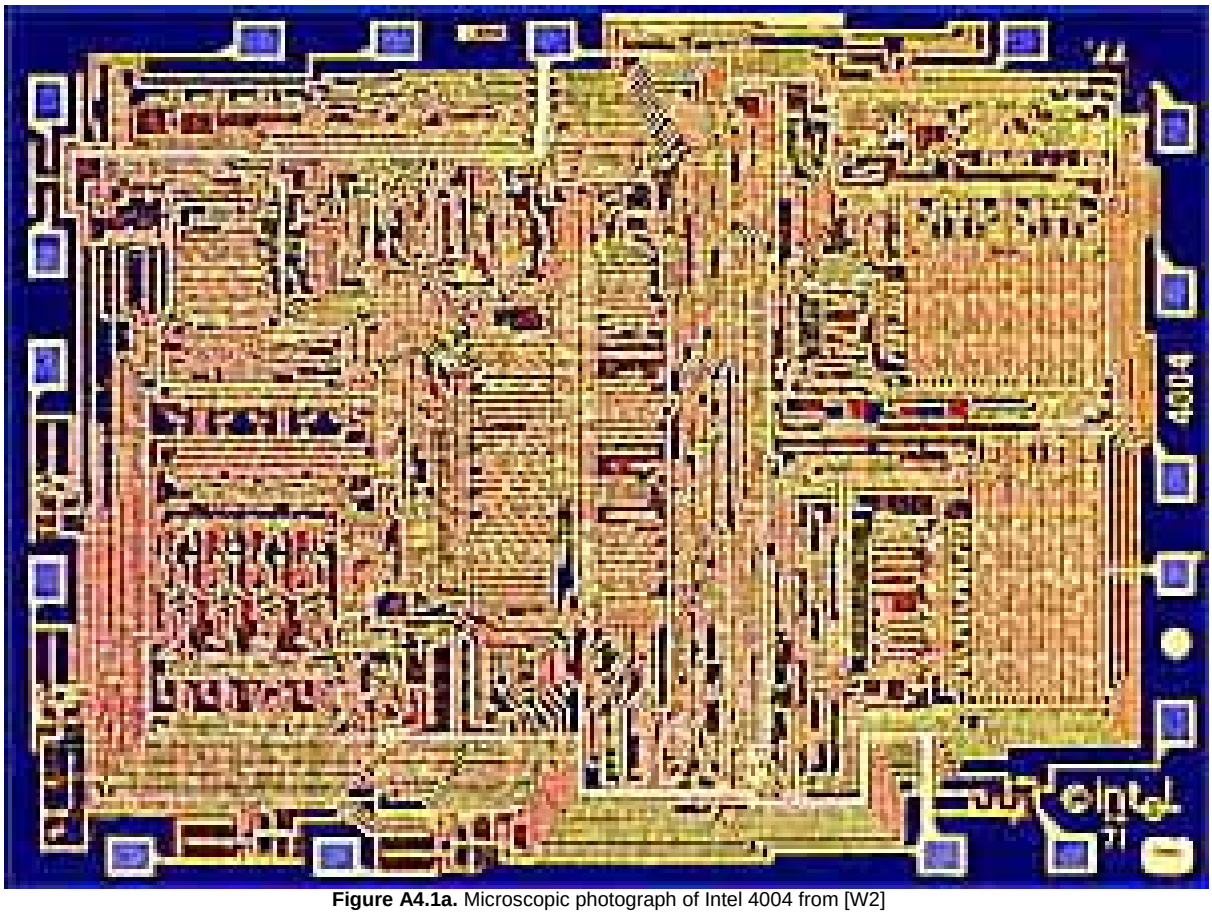


Figure A4.1a. Microscopic photograph of Intel 4004 from [W2]

- First general-purpose, single-chip microprocessor
- Shipped in 1971
- 8-bit architecture, 4-bit implementation
- 2,300 transistors
- Performance < 0.1 MIPS
  (Million Instructions Per Sec)
- 8008: 8-bit implementation in 1972
  - 3,500 transistors
  - First microprocessor-based computer (Micral)
    - Targeted at laboratory instrumentation
    - Mostly sold in Europe

A4.2. 1st Generation (16-bit) Intel 8086

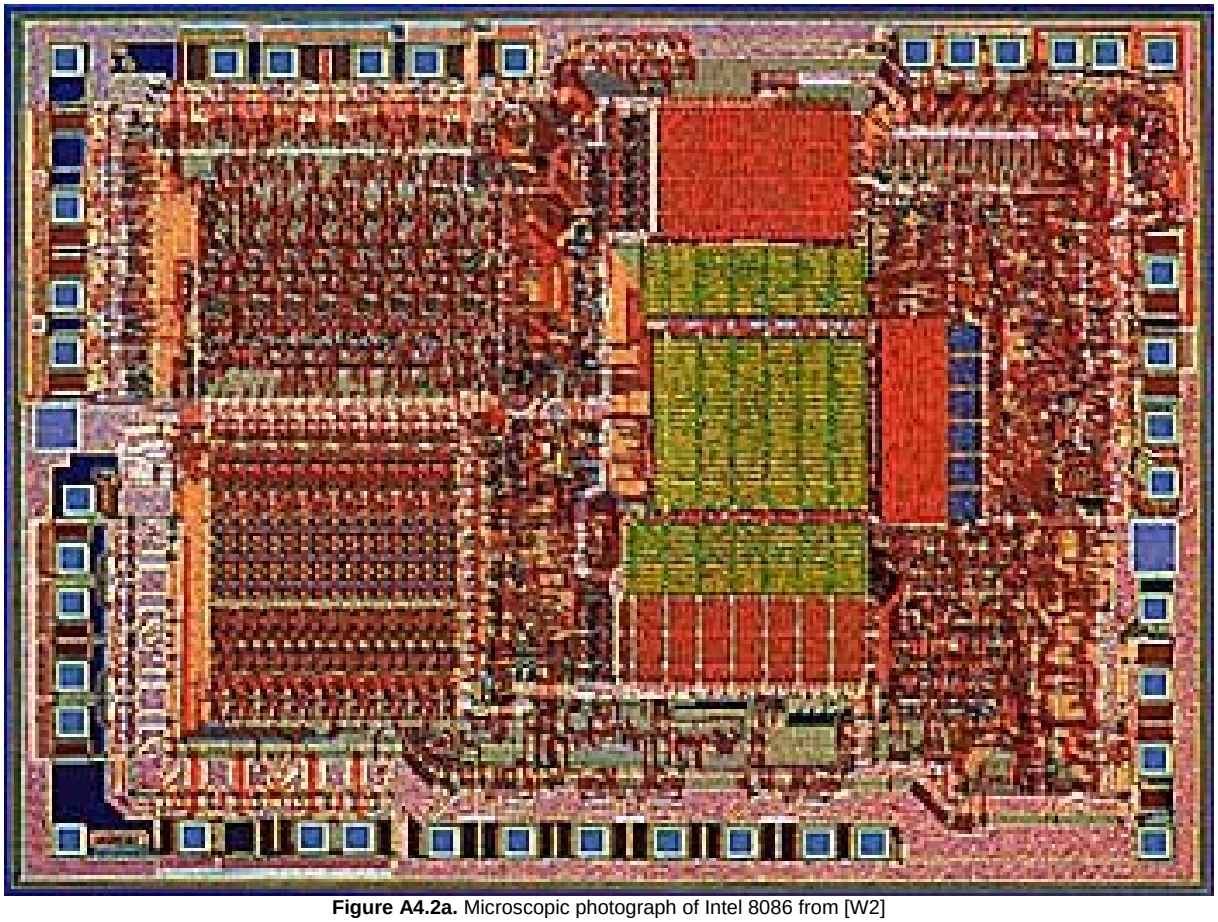


Figure A4.2a. Microscopic photograph of Intel 8086 from [W2]

- Introduced in 1978
  - Performance < 0.5 MIPS
- New 16-bit architecture
  - “Assembly language” compatible with 8080
  - 29,000 transistors
  - Includes memory protection, support for Floating Point coprocessor
- In 1981, IBM introduces PC
  - Based on 8088--8-bit bus version of 8086

A4.3. 2nd Generation (32-bit) Motorola 68000

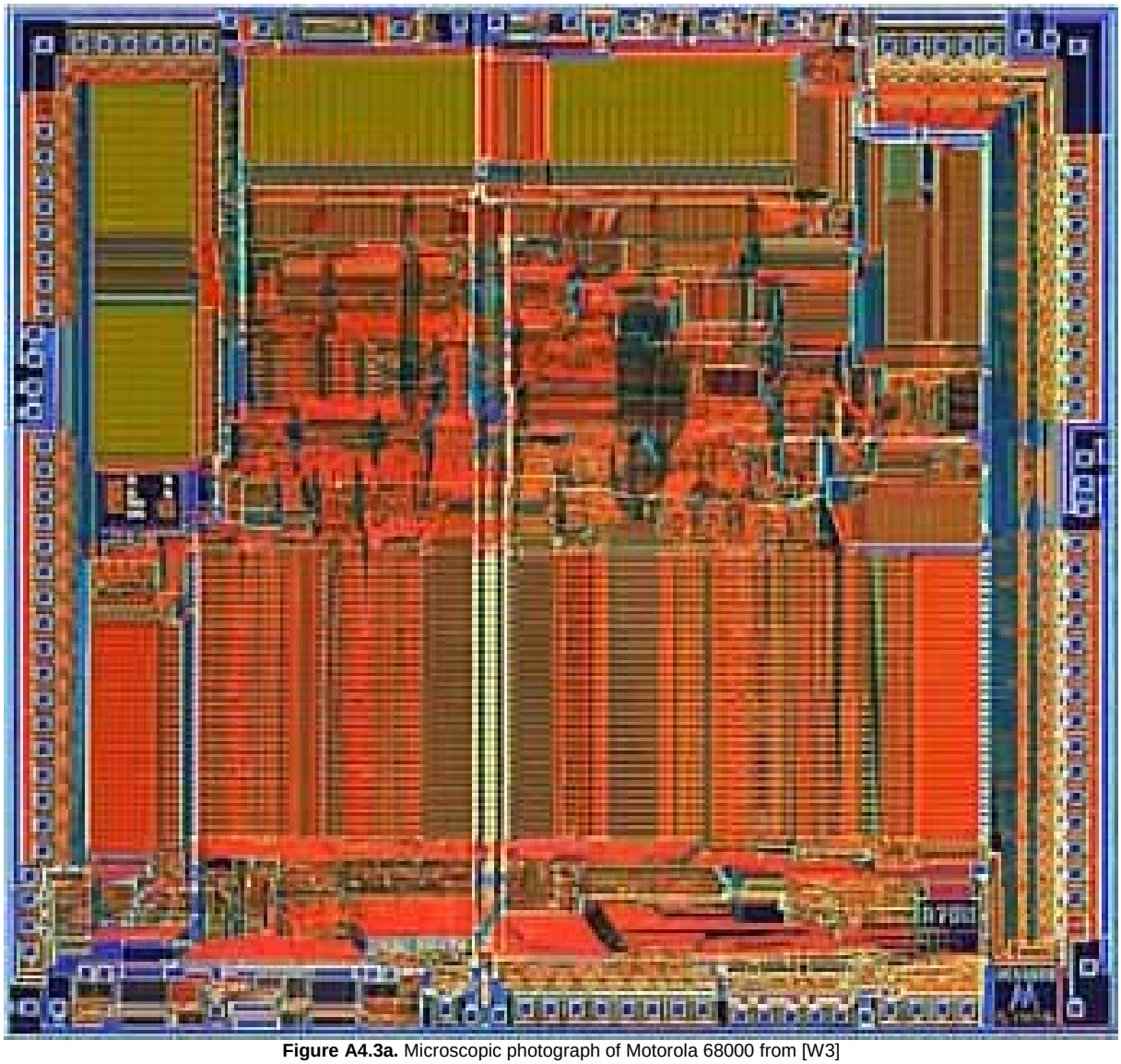


Figure A4.3a. Microscopic photograph of Motorola 68000 from [W3]

- Major architectural step in microprocessors:
  - First 32-bit architecture
    - initial 16-bit implementation
  - First flat 32-bit address
    - Support for paging
  - General-purpose register architecture
    - Loosely based on PDP-11 minicomputer
- First implementation in 1979
  - 68,000 transistors
  - < 1 MIPS (Million Instructions Per Second)
- Used in
  - Apple Mac
  - Sun , Silicon Graphics, & Apollo workstations
  - Sega Mega Drive

A4.4. 3rd Generation: MIPS R2000

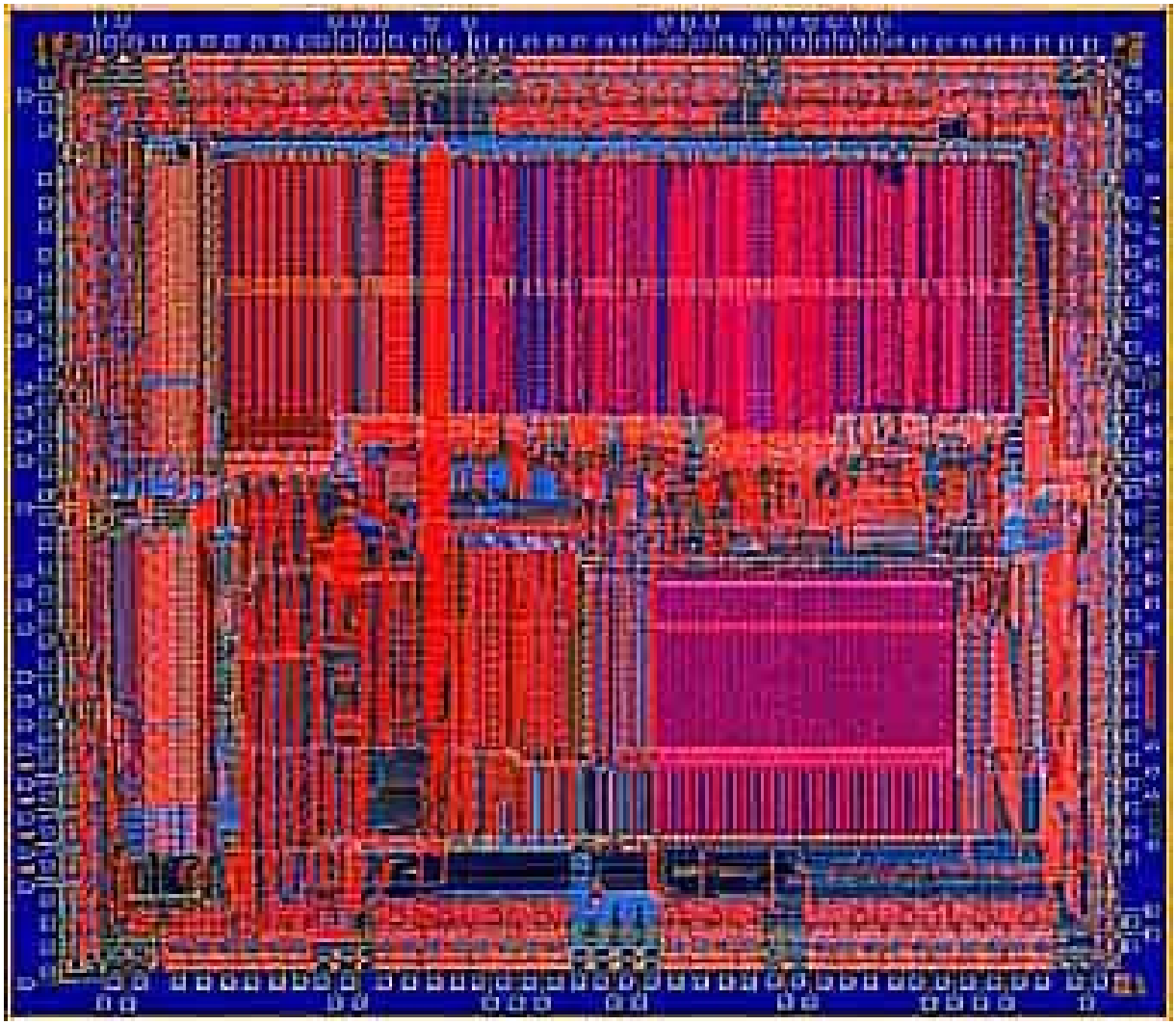


Figure A4.4a. Microscopic photograph of MIPS R2000 from [W2]

- Several firsts:
  - First (commercial) RISC microprocessor
  - First microprocessor to provide integrated support for instruction & data cache
  - First pipelined microprocessor (sustains 1 instruction/clock)
- Implemented in 1985
  - 125,000 transistors
  - 5-8 MIPS (Million Instructions per Second)

A4.5. 4th Generation (64 bit) MIPS R4000

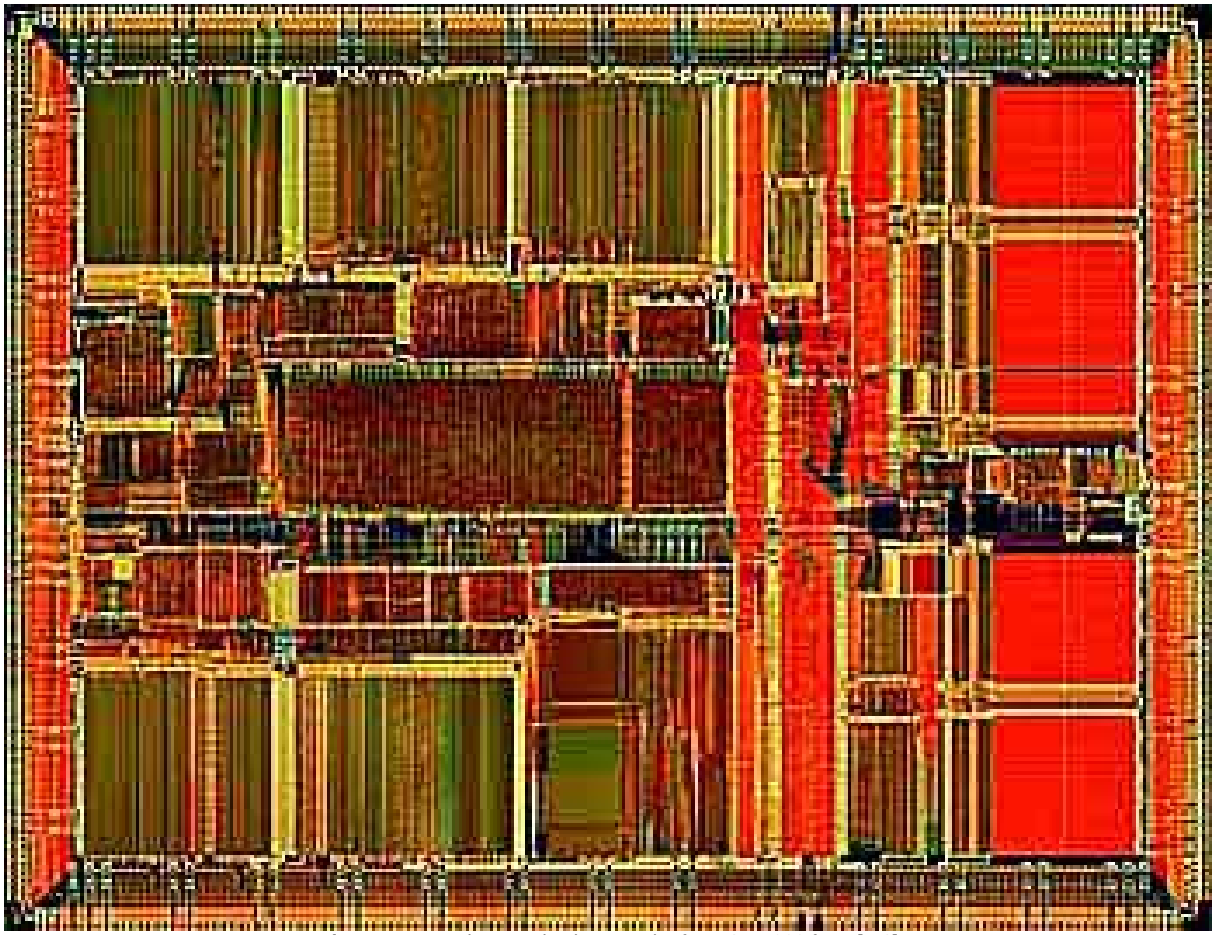


Figure A4.4a. Microscopic photograph of MIPS R4000 from [W2]

- First 64-bit architecture
- Integrated caches
  - On-chip
  - Support for off-chip, secondary cache
- Integrated floating point
- Implemented in 1991:
  - Deep pipeline
  - 1.4M transistors
  - Initially 100MHz
  - > 50 MIPS
- Intel translates 80x86/ Pentium X instructions into RISC internally

A5. MARK 3 CPU PROBLEM

Originally there was a problem with the mark 3 CPU, the add opcode, worked, but the other immediate opcodes (e.g. SUBA) did not. At first it was not understood why unpredictable results were being simulated for the SUBA opcode, since the ADDA opcode worked flawlessly (how is this possible ?) when both instructions use the same microcode, except that the ALU function is different (directly from IR).

A large amount of time (about 6 hours) was spent checking microcode, and circuit design. Eventually the problem was found, it was so simple it was unbelievable. Look closely at figure A5a, the problem is obvious (it was checked about 10 times, and not noticed, maybe not so obvious, or it never was thought possible that such a simple mistake could be made), notice B[7..0] is connected to OP1[7..0] and A[7..0] is connected OP2[7..0], clearly this is wrong as the microcode assumes that A → OP1 and B → OP2 (duuu). Blame it on Altera for mixing up the positions of A and B in the creation of the default symbol (clearly A should be above B).

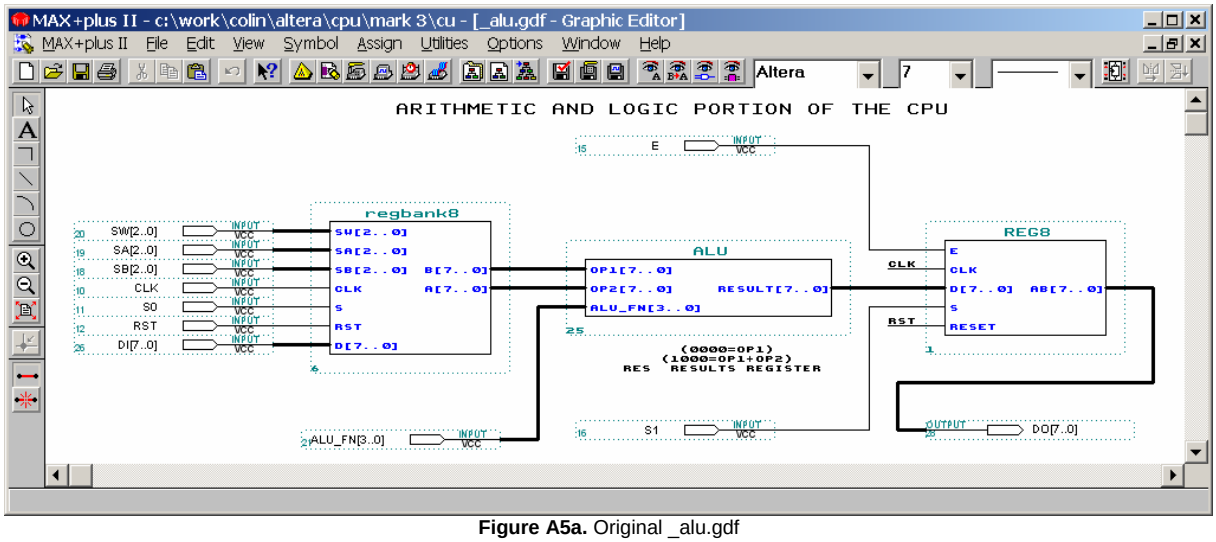


Figure A5a. Original _alu.gdf

The solution was simple, edit the default symbol and connect A to OP1 and B to OP2, see figure A5.b for the corrected ALU design.

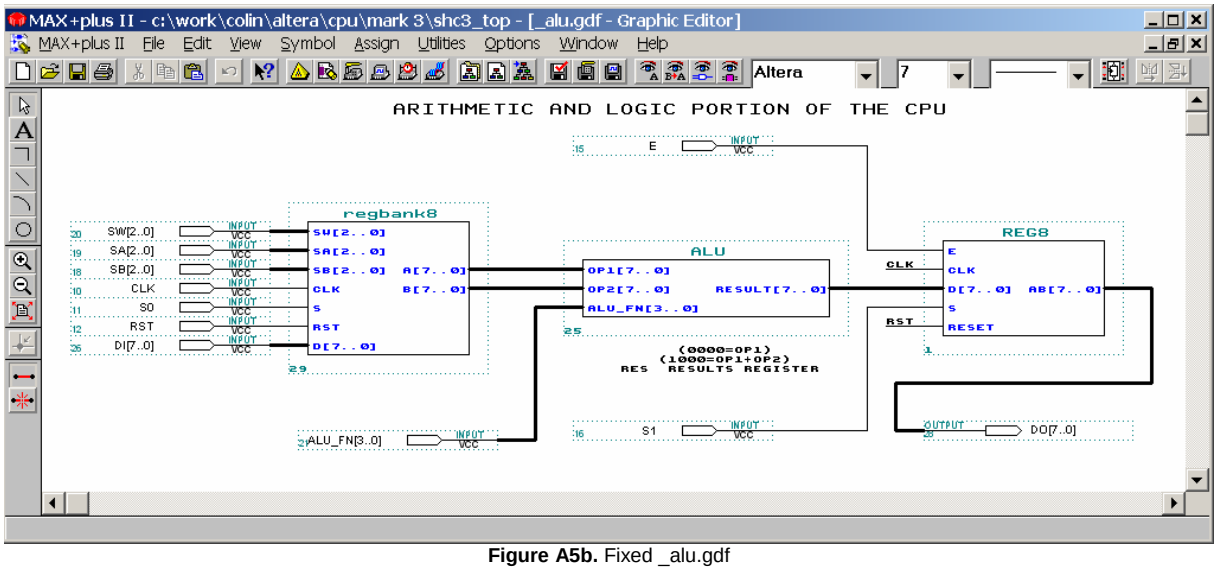


Figure A5b. Fixed _alu.gdf

The time wasted fixing this problem meant that there was not enough time to finish the mark 4.

A6. MARK 3 – HIERARCHY CHART AND PIN LAYOUT

A6.1. Hierarchy Chart

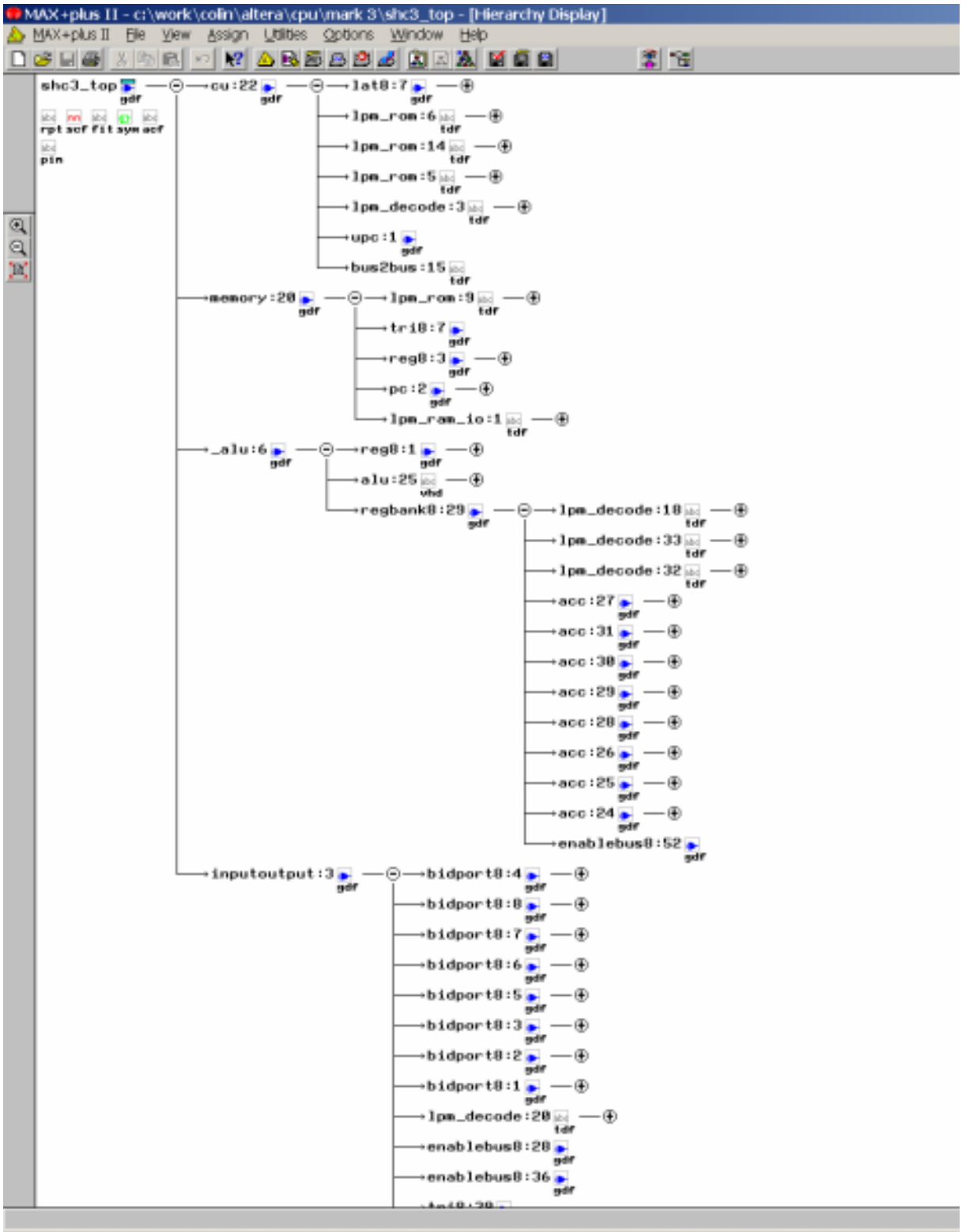


Figure A6.1a. Hierarchy Chart

A6.2. shc3_top.pin

```
-- Copyright (C) 1988-2001 Altera Corporation
-- Any megafunction design, and related net list (encrypted or decrypted),
-- support information, device programming or simulation file, and any other
-- associated documentation or information provided by Altera or a partner
-- under Altera's Megafunction Partnership Program may be used only to
-- program PLD devices (but not masked PLD devices) from Altera.  Any other
-- use of such megafunction design, net list, support information, device
-- programming or simulation file, or any other related documentation or
-- information is prohibited for any other purpose, including, but not
-- limited to modification, reverse engineering, de-compiling, or use with
-- any other silicon devices, unless such use is explicitly licensed under
-- a separate agreement with Altera or a megafunction partner.  Title to
-- the intellectual property, including patents, copyrights, trademarks,
-- trade secrets, or maskworks, embodied in any such megafunction design,
-- net list, support information, device programming or simulation file, or
-- any other related documentation or information provided by Altera or a
-- megafunction partner, remains with Altera, the megafunction partner, or
-- their respective licensors.  No other licenses, including any licenses
-- needed under any third party's intellectual property, are provided herein.



N.C. = No Connect. This pin has no internal connection to the device.
VCCINT = Dedicated power pin, which MUST be connected to VCC (5.0 volts).
VCCIO = Dedicated power pin, which MUST be connected to VCC (5.0 volts).
GNDINT = Dedicated ground pin or unused dedicated input, which MUST be connected to GND.
GNDIO = Dedicated ground pin, which MUST be connected to GND.
RESERVED = Unused I/O pin, which MUST be left unconnected.

-----------------------------------------------------------------------------

CHIP "shc3_top" ASSIGNED TO AN EPF10K20TC144-3
TCK                             : 1
CONF_DONE                       : 2
nCEO                            : 3
TDO                             : 4
VCCIO                           : 5
VCCINT                          : 6
PORTH7                          : 7
PORTG6                          : 8
PORTB5                          : 9
PORTA5                          : 10
PORTE3                          : 11
PORTF3                          : 12
PORTA7                          : 13
PORTE7                          : 14
GNDIO                           : 15
GNDINT                          : 16
PORTB3                          : 17
PORTA3                          : 18
PORTH2                          : 19
RESERVED                        : 20
PORTA0                          : 21
PORTB2                          : 22
PORTB0                          : 23
VCCIO                           : 24
VCCINT                          : 25
PORTD1                          : 26
PORTH4                          : 27
PORTC1                          : 28
PORTH3                          : 29
PORTF6                          : 30
PORTF1                          : 31
PORTC6                          : 32
PORTE0                          : 33
TMS                             : 34
nSTATUS                         : 35
RESERVED                        : 36
PORTC3                          : 37
PORTE4                          : 38
PORTF4                          : 39
GNDIO                           : 40
RESERVED                        : 41
RESERVED                        : 42
```

```
PORTF0                          : 43
PORTB1                          : 44
VCCIO                           : 45
RESERVED                        : 46
PORTE2                          : 47
RESERVED                        : 48
PORTG4                          : 49
GNDIO                           : 50
RESERVED                        : 51
VCCINT                          : 52
VCCINT                          : 53
RST                             : 54
CLK                             : 55
GNDINT                          : 56
GNDINT                          : 57
GNDINT                          : 58
PORTG0                          : 59
PORTG2                          : 60
VCCIO                           : 61
ADDRESS1                        : 62
ADDRESS0                        : 63
RESERVED                        : 64
RESERVED                        : 65
GNDIO                           : 66
ADDRESS3                        : 67
ADDRESS7                        : 68
RESERVED                        : 69
RESERVED                        : 70
VCCIO                           : 71
RESERVED                        : 72
RESERVED                        : 73
nCONFIG                         : 74
VCCINT                          : 75
MSEL1                           : 76
MSEL0                           : 77
PORTB6                          : 78
PORTA4                          : 79
PORTE6                          : 80
PORTB4                          : 81
PORTG1                          : 82
PORTG3                          : 83
GNDINT                          : 84
GNDIO                           : 85
PORTH1                          : 86
PORTD0                          : 87
PORTA1                          : 88
PORTH0                          : 89
PORTC7                          : 90
PORTF7                          : 91
PORTC0                          : 92
VCCINT                          : 93
VCCIO                           : 94
PORTD5                          : 95
PORTE5                          : 96
PORTC4                          : 97
PORTD4                          : 98
RESERVED                        : 99
ADDRESS2                        : 100
RESERVED                        : 101
PORTD3                          : 102
GNDINT                          : 103
GNDIO                           : 104
TDI                             : 105
nCE                             : 106
DCLK                            : 107
DATA0                           : 108
PORTH5                          : 109
RESERVED                        : 110
RESERVED                        : 111
RESERVED                        : 112
RESERVED                        : 113
ADDRESS5                        : 114
VCCIO                           : 115
PORTF2                          : 116
RESERVED                        : 117
RESERVED                        : 118
ADDRESS4                        : 119
```

```
ADDRESS6                        : 120
RESERVED                        : 121
PORTG7                          : 122
VCCINT                          : 123
GNDINT                          : 124
GNDINT                          : 125
GNDINT                          : 126
GNDINT                          : 127
PORTG5                          : 128
GNDIO                           : 129
PORTE1                          : 130
PORTC5                          : 131
PORTA6                          : 132
PORTB7                          : 133
VCCIO                           : 134
PORTD7                          : 135
PORTD6                          : 136
PORTD2                          : 137
PORTF5                          : 138
GNDIO                           : 139
PORTA2                          : 140
RESERVED                        : 141
PORTC2                          : 142
RESERVED                        : 143
PORTH6                          : 144
```

A7. CD ROM: CONTAINING MAX2PLUS DESIGN FILES

This CD ROM contains all of the design files used in this report.

CD-ROM