Chapter 6

The Hardware Organisation

This chapter presented the hardware organisation mainly consisting of two parts: a custom microprocessor and a fitness evaluator. The remaining sections are organised as follows. Section 6.1 presents the top-level design. Section 6.2 presents the microprocessor in detail. Section 6.3 presents the fitness evaluator.

6.1 Top-level Design

The department of computer engineering provided us plenty of prototyping boards for Xilinx XC4010 FPGAs (Figure 6.1). A prototyping board consisted of a small-size FPGA permanently connected to an EEPROM and a static RAM. There were four input/output ports for connecting the FPGA to other devices. Because the overall design was too large to synthesise in one FPGA, it was necessary to split the design into two parts: the microprocessor and the fitness evaluator. The top-level design is shown in Figure 6.2.

Microprocessor: The 16-bit, load/store architecture microprocessor was used to execute the GA. The processor was connected to a program memory (EEPROM) and a data memory (RAM). The processor was able to write data to the BUFFER and start/stop the fitness evaluator (EV).

EEPROM(1): The (32k $\times$ 8-bit) EEPROM was used as program memory storing the machine code of GA. The object code was about 2k bytes. The first 512 bytes were used to store the constant values used by the program. The memory organisation was shown in Figure 6.3.

RAM: The (32k $\times$ 8-bit) RAM was used as data memory storing a population, program variables, and CPU stack. The population was stored at address `0x0000` upto `0x0FFF`. The program variables began at address `0x1000` upto `0x3FFF`. The

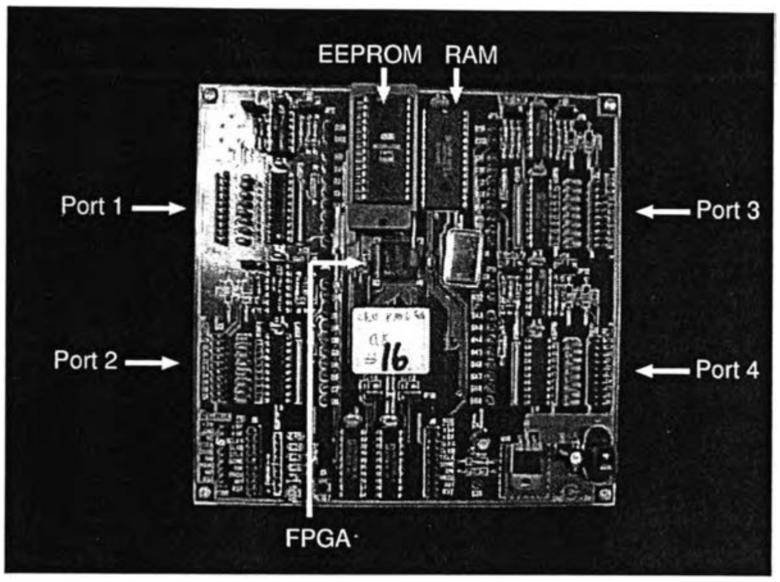

Figure 6.1: The prototyping board.

CPU stack began at address `0x3FFF` downto `0x0000`. Note that the CPU stack possibly collided with the program variables. The memory organisation was shown in Figure 6.4.

BUFFER: The BUFFER was a dual-port memory used to stored an individual while it was being evaluated. The processor wrote an individual to the BUFFER, then started the EV to evaluate the individual.

EEPROM(2): The (32k $\times$ 8-bit) EEPROM was used to stored the input/output sequences collected from the target circuit. The maximum size of input and output were set at 4 bits, and thus the EEPROM was able to store 32k input/output pairs.

EV: The evaluator was used to speedup the fitness evaluation. The fitness evaluator used the input/output sequences stored in the EEPROM(2) to calculate the fitness of an individual stored in the BUFFER.

6.2 The Microprocessor

The 16-bit, load/store architecture microprocessor was customised to the execution of GA. At the design stage, the number of registers was set at 8. It was later reduced to 4 because the processor was too large to be implemented in single FPGA. The instruction set consisted of load/store instructions and arithmetic instructions. The load/store instructions were used to performed loading and storing data between data memory and registers. The arithmetic instructions were used to performed arithmetic operations between the registers, then the result was stored in a register.

All instructions were 16 bits in length. The operand can be an address or a register number. The instruction will be further described in this format:

```
LDC(4) r1(3) addr(9)            REG[r1] = Pmemo[addr];
```

In the example, `LDC` is an opcode. The `r1` and `addr` are operands. The number in the braces is the number of used bits, for example, the opcode is 4 bits. The result of an execution is shown in the right-hand side.

There were 4 conditional jump instructions: `JEQ`, `JNE`, `JGR`, and `JLE`. The jump action was determined by 3 flags: `eq`, `gr`, and `le`.

```
JEQ(4) addr(12)            if eq flag was set to "1" then PC = PC + addr;
JNE(4) addr(12)            if eq flag was set to "0" then PC = PC + addr;
JGR(4) addr(12)            if gr flag was set to "1" then PC = PC + addr;
JLE(4) addr(12)            if le flag was set to "1" then PC = PC + addr;
```

There was one unconditional jump instruction.

```
JMP(4) addr(12)            PC = PC + addr;
```

The jump-to-subroutine instruction saved the program counter (PC) to CPU stack before jumping into subroutine. The `Dmemo[SP]` denoted the data memory pointed by the stack pointer.

```
JSR(4) addr(12)            SP = SP - 1;
                           Dmemo[SP] = PC;
                           PC = PC + addr;
```

The `CIJ` instruction was a combination of 3 instructions: `CMP`, `INC`, and `JNE`.

```
CIJ(4) r1(2) r2(2) addr(8)   eq = gr = le = 0;
                             if (REG[r1] == REG[r2]) eq = 1;
```

```
                                  if (REG[r1] > REG[r2]) gr = 1;
                                  if (REG[r1] < REG[r2]) le = 1;
                                  REG[r1] = REG[r1] + 1;
                                  if eq flag was set to "0" then PC = PC + addr;
```

The return-from-subroutine instruction had no operand. The PC was restored from the CPU stack to continue the execution.

```
RES(4)                            PC = Dmemo[SP];
                                  SP = SP + 1;
```

The LDC instruction was used to load a constant value to a register. The `addr` in the operand pointed to the constant value stored in the program memory (EEPROM).

```
LDC(4)  r1(3)  addr(9)            REG[r1] = Pmemo[addr];
```

The `LDD` and `STD` instructions were used to load/store directly to a specified address. These instructions always performed with the register number 0.

```
LDD(4)  addr(12)                  REG[0] = Dmemo[addr];
STD(4)  addr(12)                  Dmemo[addr] = REG[0];
```

The `LDR` and `STR` instructions were used to load/store to an address pointed by a register.

```
LDR(4)  r1(3)  r2(3)              REG[r2] = Dmemo[REG[r1]];
STR(4)  r1(3)  r2(3)              Dmemo[REG[r2]] = REG[r1];
```

The `LDX` and `STX` instructions were used to load/store to an address pointed by the sum of two registers.

```
LDX(4)  r1(3)  r2(3)  r3(3)       REG[r3] = Dmemo[REG[r1] + REG[r2]];
STX(4)  r1(3)  r2(3)  r3(3)       Dmemo[REG[r2] + REG[r3]] = REG[r1];
```

The `SEV`, `REV`, `LFH`, and `LFN` instructions were used to control the fitness evaluator. The EV was operated by the following steps. First, the EV was reset using `REV` instruction. Second, the EV was started using `SEV` instruction. Next, the EV evaluated the fitness of an individual stored in the BUFFER. While the individual was being evaluated, the `LFN` instruction gave a value `0x0000`. Once the evaluation was finished, the

`LFN` instruction gave a value `0x0001`. Finally, the `LFH` instruction was used to read the fitness value from the EV.

`SEV(10)`	start the evaluator;
`REV(10)`	reset the evaluator;
`LFH(10) r1(3)`	REG[r1] = fitness value (from the evaluator);
`LFN(10) r1(3)`	REG[r1] = finish signal (from the evaluator);

The `HLT` instruction was used to halt the microprocessor.

`HLT(10)`	halt processor;

The `SED` instruction was used to seed the random number generator.

`SED(10) r1(3)`	seed = REG[r1];

The `MOV` instruction was used to duplicate a register value.

`MOV(10) r1(3) r2(3)`	REG[r2] = REG[r1];

The `CMP` instruction was used to compare the values between two registers, then the flags were set according to the comparison.

`CMP(10) r1(3) r2(3)`	eq = gr = le = 0;
	if (REG[r1] == REG[r2]) eq = 1;
	if (REG[r1] > REG[r2]) gr = 1;
	if (REG[r1] < REG[r2]) le = 1;

The following instructions were used to performed the arithmetic operations: 1's complement, shifting left, shifting right, increment, decrement, and clearing.

`COM(10) r1(3)`	REG[r1] = ~REG[r1];
`SFL(10) r1(3)`	REG[r1] = REG[r1] ≪ 1;
`SFR(10) r1(3)`	REG[r1] = REG[r1] ≫ 1;
`INC(10) r1(3)`	REG[r1] = REG[r1] + 1;
`DEC(10) r1(3)`	REG[r1] = REG[r1] - 1;
`CLR(10) r1(3)`	REG[r1] = 0;

The `POT` instruction was used to display a register to the output port.

`POT(10) r1(3)`	output port = REG[r1];

The `PSH` and `POP` instructions were used to push and pop CPU stack.

`PSH(10) r1(3)`	SP = SP - 1;
	Dmemo[SP] = REG[r1];

`POP(10) r1(3)`	REG[r1] = Dmemo[SP];
	SP = SP + 1;

The following instructions were used to perform the arithmetic and logical operations: add, and, or, exclusive or. The result was stored in REG[r1].

`ADD(10) r1(3) r2(3)`	REG[r1] = REG[r1] + REG[r2];
`AND(10) r1(3) r2(3)`	REG[r1] = REG[r1] & REG[r2];
`ORR(10) r1(3) r2(3)`	REG[r1] = REG[r1] | REG[r2];
`XOR(10) r1(3) r2(3)`	REG[r1] = REG[r1] $\oplus$ REG[r2];

The `STI` instruction was used to store a value to the BUFFER of which the address was pointed by a register.

`STI(10) r1(3) r2(3)`	BUFFER[REG[r2]] = REG[r1];

The `RND` instruction was used to produce a random number.

`RND(10) r1(3)`	REG[r1] = random number;

The `AD3` instruction was used to store the sum of two registers to another register.

`AD3(7) r1(3) r2(3) r3(3)`	REG[r3] = REG[r1] + REG[r2];

The instruction set was summarised in Table 6.1. We designed the microprocessor that was capable of executing those instructions. The design of the microprocessor is shown in Figure 6.6 and Figure 6.7. The microprocessor consisted of the following components: LATCH, REG, ALU, MUX, IR, PC, SP, MIM, PORT OUT, CONTROL UNIT.

LATCH

The latch was used to block a signal sent to the bus. If the signal was blocked, the output will be high impedance. The signal pins of latch(1) and latch(2) were presented as follows.

LATCH(1)

Pin name	Type	Width	Description
reg_out1	in	16	signal sent to data bus
i_data	out	16	i_data = (l1_ena == 1 ? reg_out1 : Z)
l1_ena	in	1	

LATCH(2)

Pin name	Type	Width	Description
alu_out	in	16	signal sent to address bus
i_addr	out	16	i_addr = (l2_ena == 1 ? alu_out : Z)
l2_ena	in	1	

REG

The register bank consisted of 4 16-bit registers. The selected registers were sent to the arithmetic unit (ALU). The value sent from MUX was stored in a specified register when the `reg_load` was positive edge (posedge denoted positive edge).

Pin name	Type	Width	Description
reg_sel1	in	2	register number (0-3)
reg_sel2	in	2	register number (0-3)
reg_out1	out	16	reg_out1 = REG[reg_sel1]
reg_out2	out	16	reg_out2 = REG[reg_sel2]
mux_out	in	16	a value to be stored in a register
reg_load	in	1	if (posedge reg_load) REG[reg_sel1] = mux_out;

ALU

The ALU was used to perform the arithmetic and logical operations of the selected registers. When the `alu_load` was positive edge, the ALU performed the arithmetic operation according to the opcode sent from `IR`.

Pin name	Type	Width	Description
reg_out1	in	16	register value

reg_out2	in	16	register value
alu_load	in	1	if (posedge alu_load) do arithmetic operation;
alu_out	out	16	result of the arithmetic operation
ir_instr	in	10	opcode sent from IR

The 16-bit random number generator (Hortensius, 1989), embedded in the ALU, was an one-dimensional, 2-state cellular automata (CA) in which the rule was 150-150-90-150-90-150-......-90-150 (see Figure 6.5). The CA consisted of 16 cells updated synchronously according to the rule. The rule 90 and rule 150 were defined as:

$$\texttt{Rule 90:} \qquad S_i = S_{i-1} \oplus S_{i+1}$$
$$\texttt{Rule 150:} \qquad S_i = S_{i-1} \oplus S_i \oplus S_{i+1}$$

where

S_i	is a state of cell$_i$
S_{i-1}	is a state of the left neighbor of cell$_i$
S_{i+1}	is a state of the right neighbor of cell$_i$

The CA with the rule 150-150-90-150-...-90-150 can produce a random number between `0x0001` and `0xFFFF`. We choosed the CA instead of linear feedback shift registers (LFSR) because CA used a small number of gates and produced a random number in one clock.

MUX

The multiplexor was used to selected the data to be stored in a register. For load instructions, the data was selected from `i_data`. For arithmetic instructions, the data was selected from `alu_out`.

Pin name	Type	Width	Description
alu_out	in	16	data to be selected
i_data	in	16	data to be selected
mux_sel	in	1	select pin
mux_out	out	16	mux_out = (mux_sel == 0 ? alu_out : i_data);

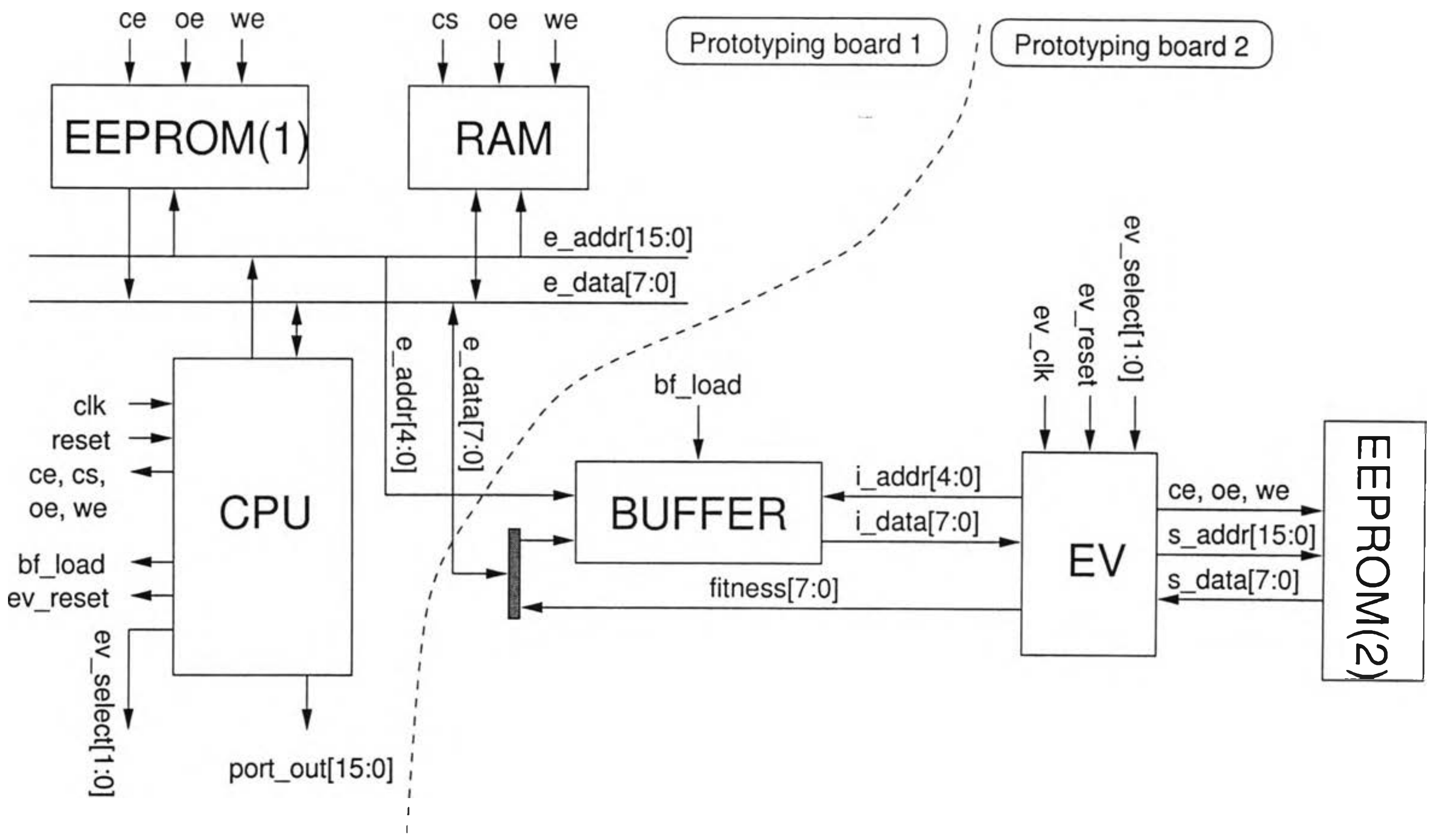

Figure 6.2: Top-level design.

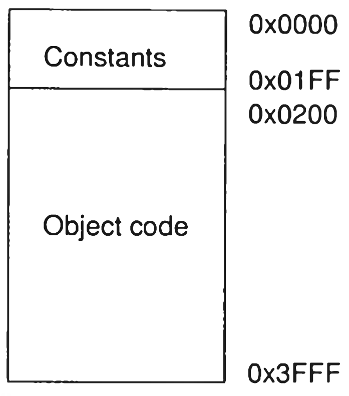

Figure 6.3: EEPROM(1).

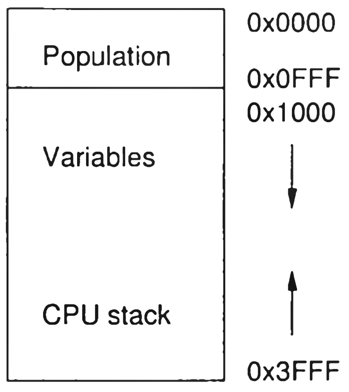

Figure 6.4: RAM.

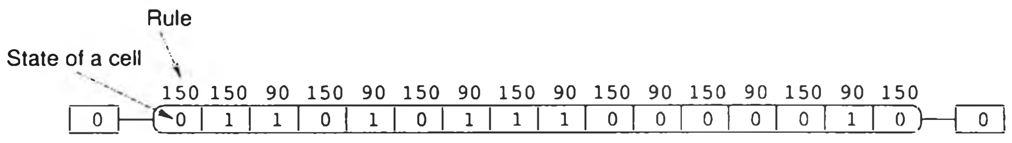

Figure 6.5: Random number generator.

IR

The instruction register (IR) was used to stored an instruction while it was being executed. The opcode and the operand were extracted from the instruction, then sent to the ALU and the control unit.

Pin name	Type	Width	Description
ir_ena	in	1	enable pin
ir_load	in	1	load pin
i_data	in	16	instruction if (posedge ir_load) IR = i_data;
ir_instr	out	10	opcode extracted from IR
ir_rng	out	9	register numbers extracted from IR

i_addr	out	16	direct address extracted from IR
			i_addr = (ir_ena == 1 ?
			direct_address : Z);

ir_rel	out	12	relative address extracted from IR

PC

The program counter (PC) was initialised at address `0x0200`. Every executing an instruction, the PC was increased by one. For jump instruction, the PC was added to the relative address sent from IR. The PC involved the CPU stack when jumping into subroutine.

Pin name	Type	Width	Description
ir_rel	in	12	relative address
pc_ena1	in	1	enable pin 1
pc_ena2	in	1	enable pin 2
pc_mode	in	2	operational mode

pc_load	in	1	if (posedge pc_load)
			if (mode == 00) PC = 0x0200;
			if (mode == 01) PC = PC + 1;
			if (mode == 10) PC = PC + ir_rel;
			if (mode == 11) PC = i_data;
			endif

i_addr	out	16	i_addr = (pc_ena1 == 1 ? PC : Z)
i_data	inout	16	i_data = (pc_ena2 == 1 ? PC : Z)

SP

The stack pointer (SP) was used to stored the top-of-stack address. The SP was initialised at 0x4000. For push operation, the SP was decreased by one. For pop operation, the SP was increased by one.

Pin name	Type	Width	Description
sp_ena	in	1	enable pin
sp_mode	in	2	operational mode

sp_load	in	1	if (posedge sp_load)
			if (mode == 00) SP = 0x4000;
			if (mode == 01) SP = SP + 1;
			if (mode == 10) SP = SP - 1;
			endif

i_addr	out	16	i_addr = (sp_ena == 1 ? SP : Z);

MIM

The memory interface module (MIM) was used to connect to the memory devices (EEPROM, RAM, and BUFFER) of which the data width was 8 bits. In read mode, the MIM read a value from an external device, then the value was placed on i_data. In write mode, the MIM wrote a value from i_data to an external device using the address on i_addr. The bf_load signal was used to write the BUFFER. Once the bf_load was positive edge, the data on e_data and the address on e_addr were used to write the BUFFER. The ev_select signal was used to select 3 signals from the EV to e_data. The three signals were fitness[15:8], fitness[7:0], and finish value.

Pin name	Type	Width	Description
i_addr	in	16	internal address bus
i_data	inout	16	internal data bus
clk	in	1	clock

mim_mode	in	3	if (mim_mode == 000) reset;
			if (mim_mode == 001) read EEPROM;
			if (mim_mode == 010) read RAM;
			if (mim_mode == 011) write RAM;
			if (mim_mode == 100) write BUFFER;
			if (mim_mode == 101) read fitness from EV;

			if (mim_mode == 110) read finish from EV;
ce, cs, oe, we	out	4	EEPROM, RAM control signals
bf_load	out	1	if (posedge bf_load) write BUFFER;
ev_select	out	2	if (ev_select == 01) EV gave high impedance;
			if (ev_select == 10) EV gave fitness[15:8];
			if (ev_select == 01) EV gave fitness[7:0];
			if (ev_select == 11) EV gave finish value;
e_addr	out	16	external address bus
e_data	inout	1	external data bus

Port_Out

The port_out was used to display a register value.

Pin name	Type	Width	Description
po_load	in	1	if (posedge po_load) po_out = i_data;
po_out	out	16	output port (connected to LED)

Control Unit

The control unit was a 16-state FSM used to control the processor. The control unit simply fetched and executed the instructions in sequence.

Pin name	Type	Width	Description
clk	in	1	clock
reset	in	1	reset signal
eq, gr, le	in	3	flags
ir_instr	in	10	opcode
ir_rgn	in	9	operand (register numbers)
ir_ena	out	1	IR enable
ir_load	out	1	IR load
pc_ena1	out	1	PC enable1
pc_ena2	out	1	PC enable2
pc_mode	out	2	PC mode
pc_load	out	1	PC load
sp_ena	out	1	SP enable

sp_mode	out	2	SP mode
sp_load	out	1	SP load
po_load	out	1	Port_Out load
l1_ena	out	1	latch1 enable
l2_ena	out	1	latch2 enable
mux_sel	out	1	MUX select
reg_load	out	1	REG load
reg_sel1	out	2	REG select1
reg_sel2	out	2	REG select2
alu_load	out	1	ALU load
mim_mode	out	3	MIM mode
ev_reset	out	1	EV reset

The microprocessor consumed about 90% of the total chip area. Due to the place-and-route method of the synthesis tools, the FPGA cannot be fully utilised. The processor was able to operate at 6 MHz (the bottleneck of the memory was 8 MHz).

6.3 The Fitness Evaluator

The fitness evaluator consumed a half area of the chip. The evaluator was able to operate at 20 MHz (higher than the bottleneck of the memory). It can be seen that the design yielded a satisfactory result. Therefore it was not necessary to put more effort for optimising area and clock speed. Since the fitness evaluator was designed as single behavioral module, we did not know the inside structure of the actual circuit synthesised by the Xilinx tool.

The behavioral description of the fitness evaluator is shown in Algorithm 6.1. The fitness evaluator performed a 4-stage pipeline. At first stage, the evaluator fetched an input/output from the EEPROM(2). At second stage, the current state and the input read from the EEPROM(2) were used to clock the individual stored in the BUFFER, then the BUFFER gave a next state and an output. At third stage, the output read from the EEPROM(2) was compared to the output read from the BUFFER. The number of similar output bits was recorded. At fourth stage, the fitness was increased by the number of similar output bits. If end of the sequence, the current state was reset to the start state.

Algorithm 6.1 Fitness evaluation.

```
m denoted number of sequences.
n denoted sequence length.

inp[1..m][1..n] denoted the input sequences stored in EEPROM(2).
out[1..m][1..n] denoted the output sequences stored in EEPROM(2).

next_sta denoted next state read from the BUFFER.
next_out denoted next output read from the BUFFER.

BUFFER[state][input] returned the next state and the output of an FSM
                     stored in the BUFFER.

i = 0;
j = 0;
fitness = 0;
next_sta = start_state;
do in pipeline
  stage 1: read inp[i][j], out[i][j] from EEPROM(2);
  stage 2: (next_sta, next_out) = BUFFER[next_sta][inp[i][j]];
  stage 3: sim = number of similar bits between
                  out[i][j] and next_out;
  stage 4: fitness = fitness + sim;
            j = j + 1;
            if (j eq n)
                j = 0;
                i = i + 1;
                if (i eq m) stop pipeline;
            endif
enddo
```

Table 6.1: The instruction set.

Instruction	Opcode	Description	Clocks
JEQ	0000	jump if equal	8
JNE	0001	jump if not equal	8
JGR	0010	jump if greater than	8
JLE	0011	jump if less than	8
JMP	0100	jump	8
JSR	0101	jump subroutine	10
CIJ	0110	compare, increment and jump	9
RES	0111	return subroutine	9
LDC	1000	load constant	9
LDD	1001	load direct	9
STD	1010	store direct	8
LDR	1011	load register	10
STR	1100	store register	9
LDX	1101	load x	10
STX	1110	store x	10
SEV	1111 0000 00	start evaluator	7
REV	1111 0000 01	stop evaluator	7
LFH	1111 0000 10	load fitness	8
LFN	1111 0001 00	load finish	8
HLT	1111 0001 01	halt	7
SED	1111 0001 10	seed	8
MOV	1111 0001 11	move	10
CMP	1111 0010 00	compare	8
COM	1111 0010 01	complement	9
SFL	1111 0010 10	shift left	9
SFR	1111 0010 11	shift right	9
PSH	1111 0011 00	push	9
POP	1111 0011 01	pop	9
POT	1111 0011 11	port out	8
INC	1111 0100 00	increment	9
DEC	1111 0100 01	decrement	9
CLR	1111 0100 10	clear	9
ADD	1111 0100 11	add	9
AND	1111 0101 00	and	9
ORR	1111 0101 01	or	9
XOR	1111 0101 10	exclusive or	9
STI	1111 0110 00	store individual	9
RND	1111 0110 01	randomise	9
AD3	1111 111	add three	10

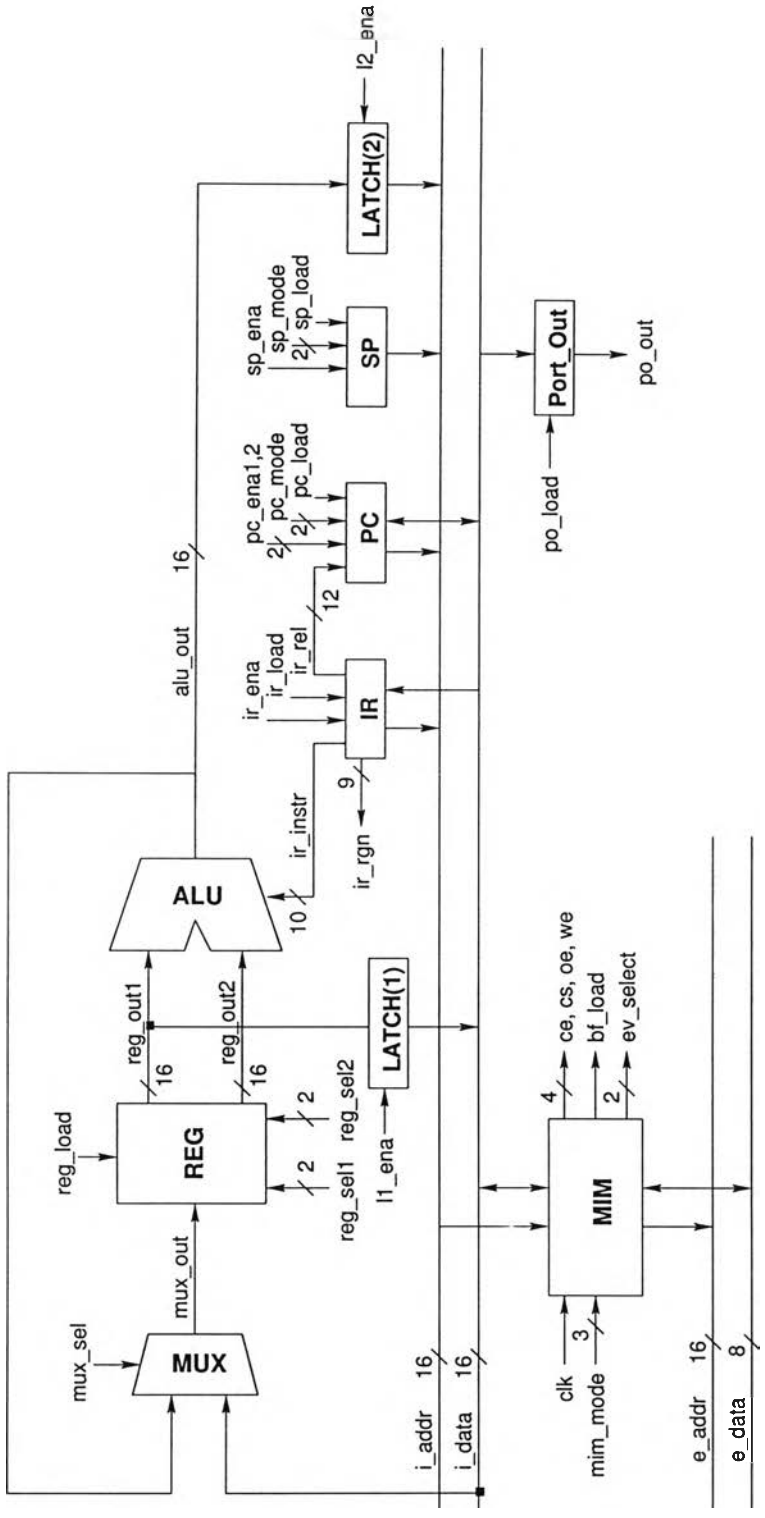

Figure 6.6: 16-bit microprocessor.

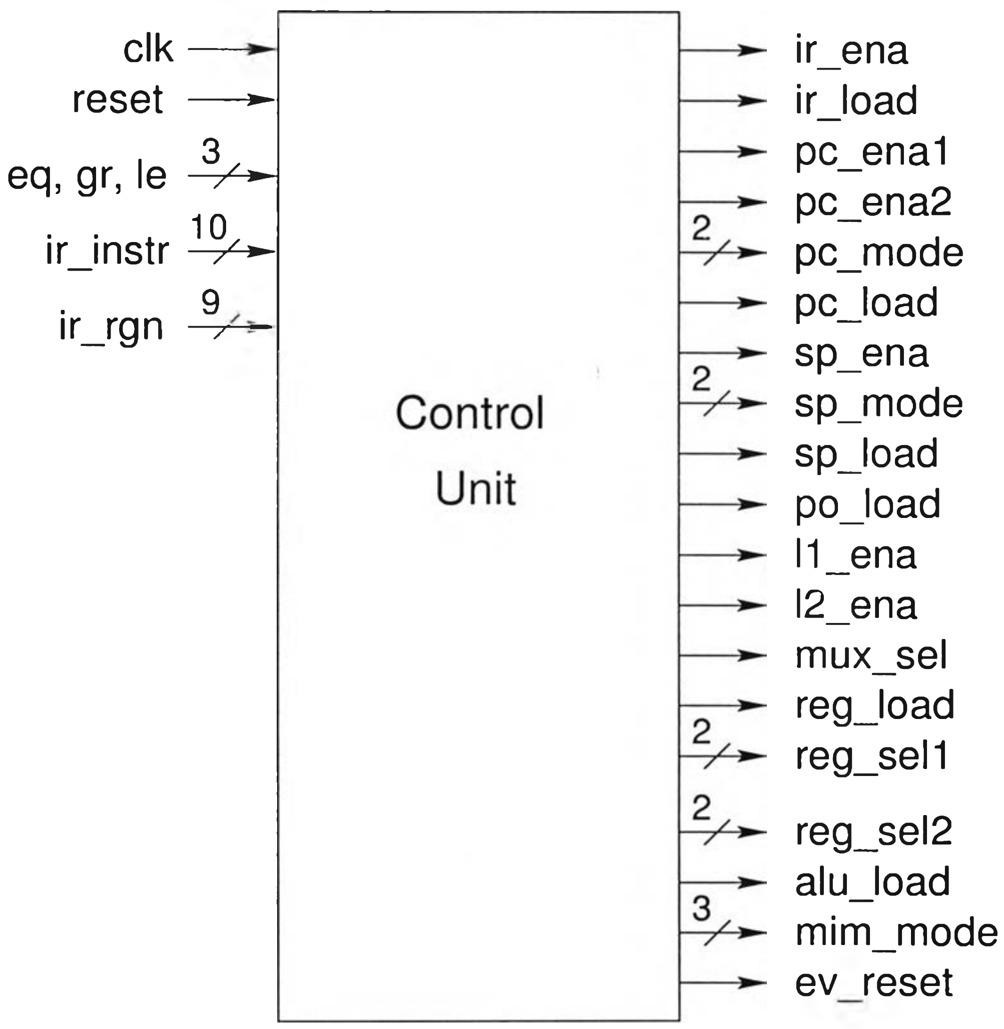

Figure 6.7: Control unit.