M.Tech (Computer Science) Dissertation Series

Redundant Radix - 4 Coprocessor Architecture and Implementation

a dissertation submitted in partial fulfillment of the requirements for the M.Tech (Computer Science) degree of the Indian Statistical Institute

by
Swamy Dhoss P.

under the supervision of

Dr. B.P.Sinha,
Professor & Head,
Advance Computing and Microelectronics Unit.

Indian Statistical Institute
203, Barrackpore Trunk Road,
Calcutta - 700 035.
1999

INDIAN STATISTICAL INSTITUTE
203, Barrackpore Trunk Road,
Calcutta - 700 035.

Certificate of Approval

This is to certify that the dissertation work entitled 'Redundant Radix - 4 Coprocessor - Architecture and Implementation', submitted by ***Swamy Dhoss P.*** (Roll No. *MTC 9722*), in partial fulfillment of the requirements for the M.Tech degree in Computer Science is an acceptable work.

Prof. B.P Sinha
Head,
Advanced Computing & Microelectronics Unit.
Indian Statistical Institute,
Calcutta - 700 035.

Date :

Acknowledgements

I would like to thank my guide, Dr. B.P. Sinha, for his constant encouragement and involvement in this project. During the course of this project, he has helped by giving useful suggestions which made this project a successful one.

I would also like to thank Wellspring corporation for their free evaluation Verilog compiler which helped me in simulating my design.

Finally, I would thank all my friends who have shared my feelings and helped me in various ways throughout the course in ISI.

Swamy Dhoss P.

Contents

Introduction

Development of coprocessors for performing specific set of the host processor's operations has been a good choice for improving the host processor's performance. Arithmetic operations like addition and multiplication of redundant radix - 4 numbers can be done in O(1) time and O(log n) time respectively [1].This gives the motivation to develop a high speed redundant radix - 4 coprocessor (RR - 4 coprocessor) which can perform all arithmetic operations of the host processor. A detailed account of RR - 4 arithmetic, binary to RR -4 conversion and the re-conversion is given in [1] and will not be repeated. Logic for addition, multiplication also has been developed [1]. We consider a closely coupled configuration of the host processor and the coprocessor wherein both the host processor and the coprocessor share the same bus control logic and clock. This report deals with the various aspects of design of coprocessor and the implementation details. Some code segments are also included for reference purpose. Implementation has been done in Verilog Hardware Description Language[2]. The model coprocessor does only integer operations and hence division operation has not been implemented due to floating point considerations. Veriwell Corp.'s Evaluation compiler for verilog was used for simulation purpose.

The host processor and coprocessor operate in parallel by sharing the same address and data bus. The external bus control logic controls the sharing of the bus between the host and the coprocessor. Only the coprocessor design is dealt with and the external bus control logic is not dealt with here . The host processor and coprocessor also exchange control signal for proper synchronization of operations. Fig 1. shows the details of the communication between the host and coprocessor and also with the bus control logic. It can be seen that this design assumes that both the host and coprocessor are driven by the same clock.

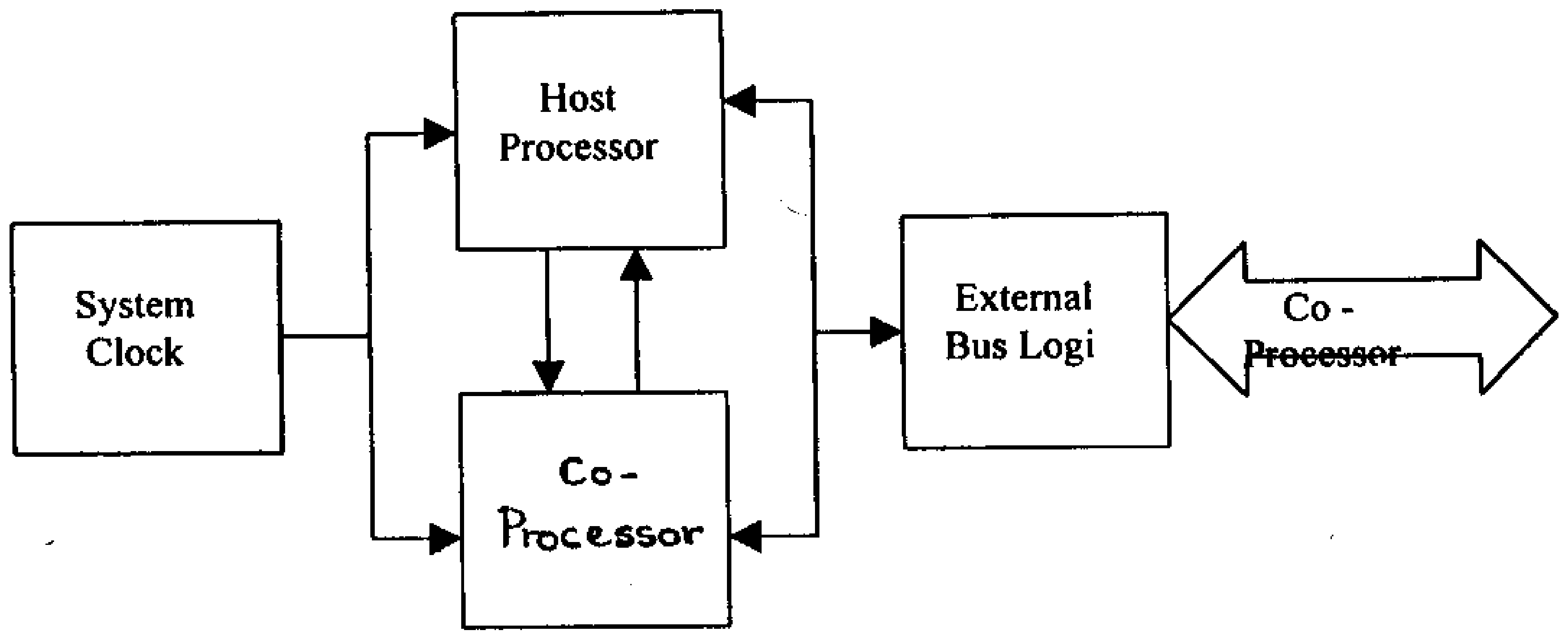


Fig. 1 Closely coupled configuration

The coprocessor runs as an independent processor which means that it has its own instruction set and just needs to be invoked by the main processor. The remaining like decoding instructions and other operations are taken care of by the coprocessor itself. Fig 2. shows the pin-out details of the proposed coprocessor's design.

The data lines and address lines are both 16 - bit wide. The following gives the description of signals that are found in the pin-out diagram in the next page.

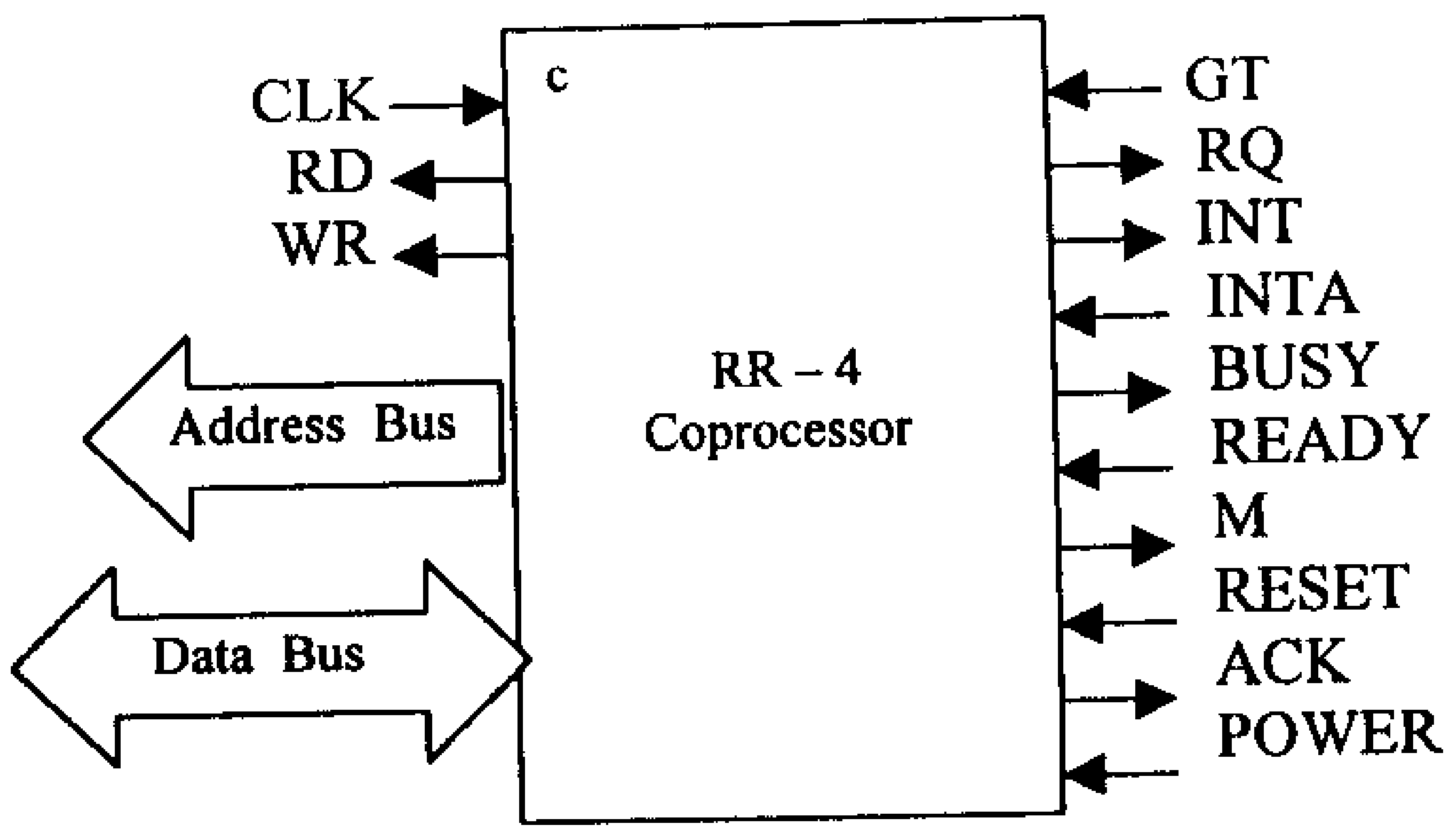


Fig. 2 Pin-Out diagram of RR - 4 Coprocessor

Clk - System clock drives both the host and coprocessor.

Ready - Signal from the host to invoke the coprocessor when it needed to be brought into service. This signal is asserted when the host comes across a special instruction.

Power - Common signal to both the host and coprocessor and initializes the state of the coprocessor by de-asserting all it's control signals.

Reset - Signal from the host to reset the coprocessor, Asserted signal.

Inta - Interrupt acknowledge signal from the host in response to an interrupt request from the coprocessor.

Gt - Signal from the external bus control logic indicating that the bus is available for the coprocessor during the current clock cycle.

Rq - Asserted signal to the external bus control logic requesting control of bus for memory operations.

M - Asserted signal to indicate memory operation. The host can use this signal to place its request for the bus to the external bus logic.

Rd - Asserted signal to the external bus logic which indicates a read operation to the memory.

Wr - Asserted signal to indicate a memory write operation.

Busy - Asserted signal used by the processor to wait upon.

Ack - Handshake signal in response to the ready signal from the host.

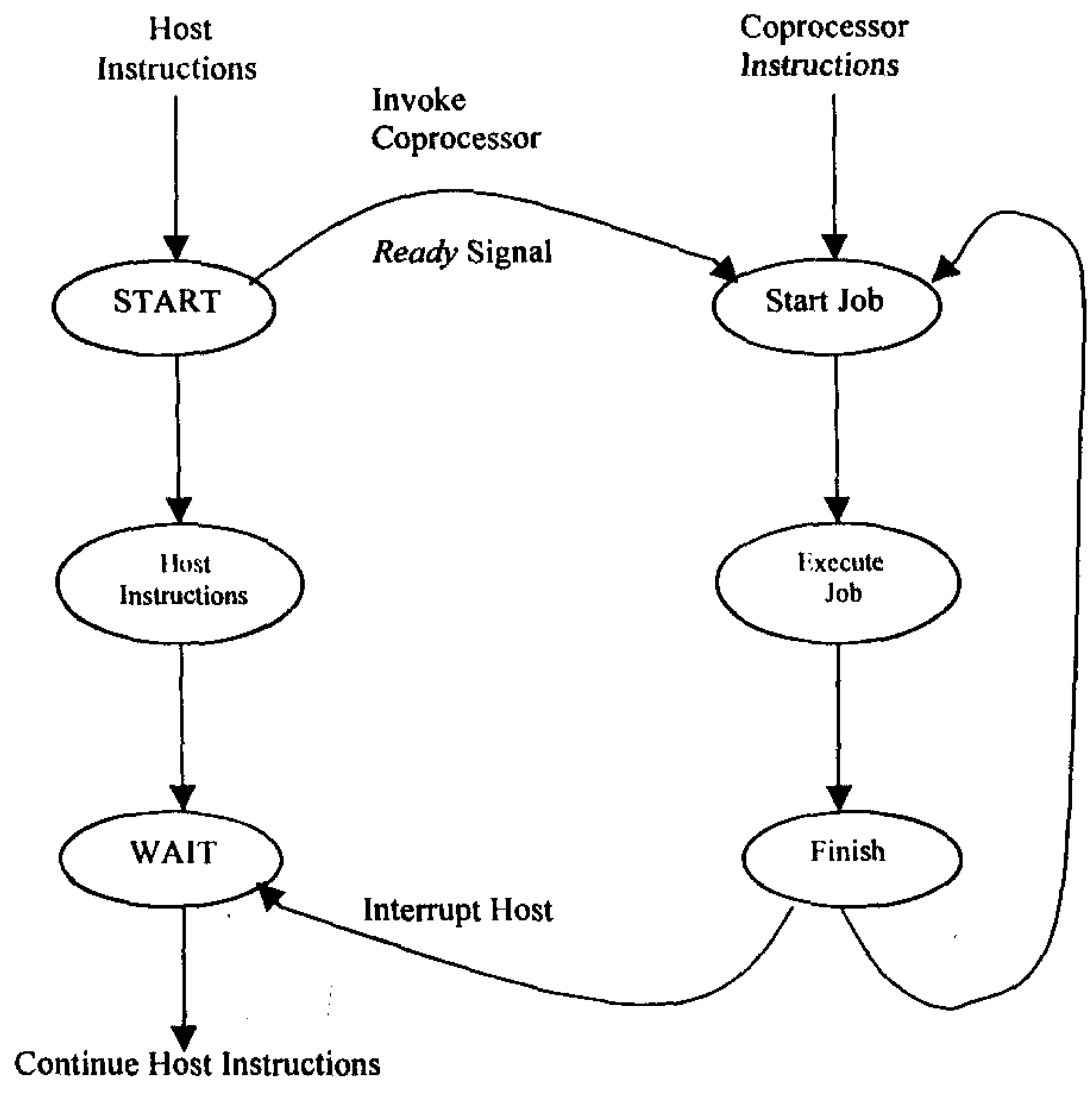
Int - Asserted signal to indicate the processor that the current assigned job is complete.

The coprocessor has its own set of instructions which it can decode. The host processor invokes the coprocessor to do some specified operation when it encounters the special instruction - *START.* This instruction is 5 byte long with one byte for the opcode and the remaining two words specify the starting and ending address of the code segment for the coprocessor. When the host processor encounter's an *START* instruction it issues a *READY* signal to the coprocessor. The subsequent clock cycles are used by the host processor to transfer the two words to the coprocessor. The coprocessor is now ready for independent operation. The first word which is the starting address of the coprocessor's code is placed in the Program counter. The second word is placed in a temporary register. After each instruction is executed the program counter is incremented. The contents of the program counter are begin compared with that of the temporary register at the beginning of each clock cycle. If they are equal then *BUSY* is pulled low and an *INT* is issued to the host processor indicating that the job is complete. The coprocessor then goes back to the initial state expecting the next *READY* signal.

Since the bus is not used by the host processor at all times, the coprocessor also steals some bus cycles to perform its memory operation, a process known as cycle stealing. The coprocessor has to explicitly place a request for the control of the bus to the external bus control logic by asserting it's *RQ* pin. It monitors *GT* pin before any memory operation is made, so as to check if it has the permission to use the common bus. The external bus logic decides when to grant the control of the bus to the coprocessor. The *RQ* pin of the coprocessor is maintained high when it needs to do any memory operation. It is pulled low when the *GT* pin is asserted by the external bus logic. Otherwise, the *RQ* pin is low. Once the coprocessor is ready for executing its required job, the *BUSY* pin is made high and remains high till the current execution is complete. On receiving the *INTA* signal the *INT* pin of the coprocessor is made low.

Whenever the host processor uses the result of the operation done by the coprocessor, then it has to *WAIT* for the coprocessor to finish its job. Hence, a special instruction *WAIT* has to be included in the host processor's instruction set. When the host encounter's a *WAIT* it tests the *BUSY* pin of the coprocessor to see if the current job is finished. If yes then the host proceeds with its execution or else waits till the completion of the task.

The above synchronization between the host and coprocessor can be inferred from the following figure also.



This gives an overview about the design of the coprocessor.

The architecture of the RR - 4 coprocessor is very simple with minimum number of registers. It differs from an 8085 processor in the fact that all the operations in ALU are performed using RR - 4 logic. To facilitate this, the architecture includes a binary to RR -4 conversion unit and an output unit which converts the RR - 4 number to binary. Fig. 3 shows the block diagram of the proposed architecture.

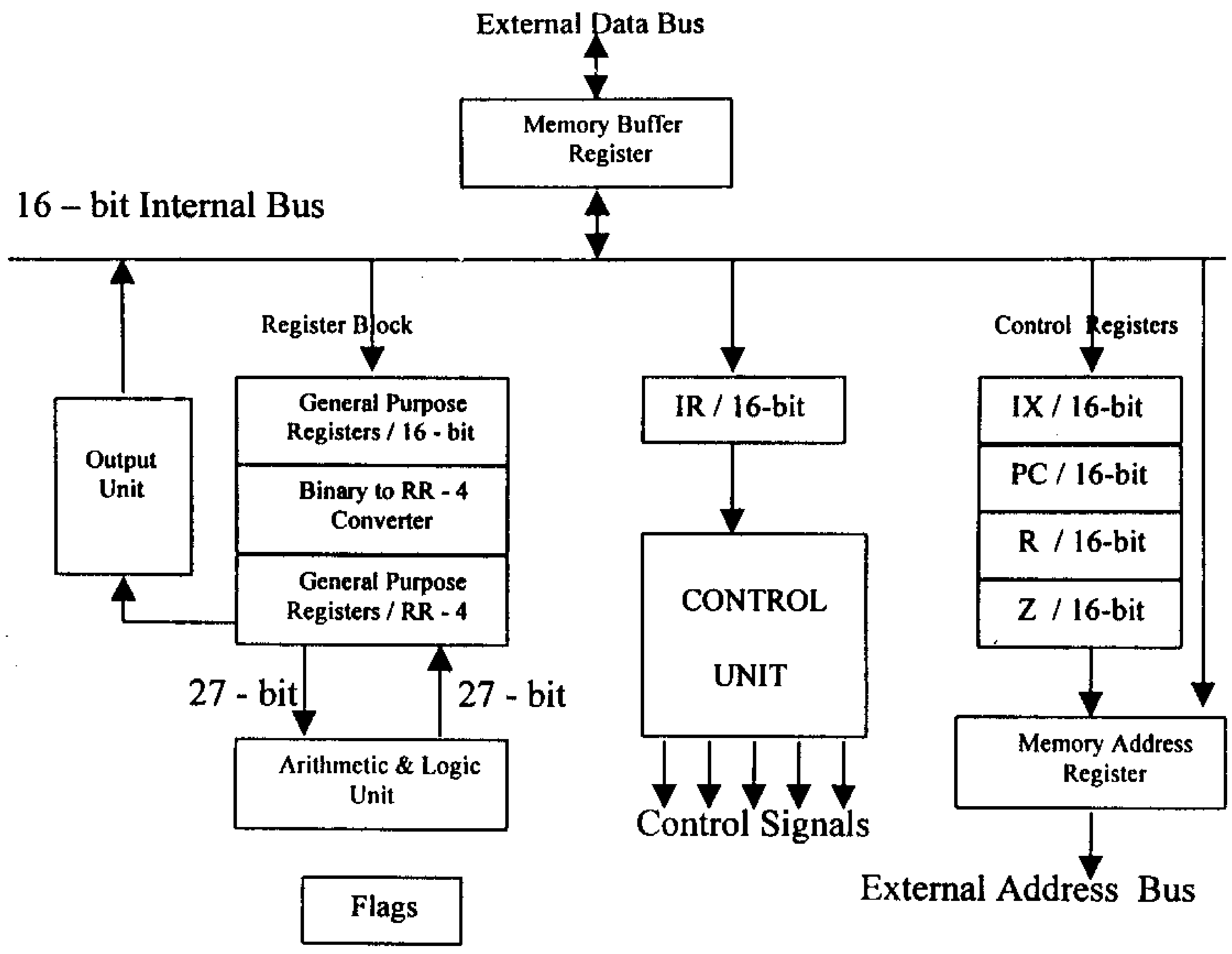


Fig. 3 Architecture of RR – 4 Coprocessor

The instruction register (IR), program counter (PC), index register (IX), temporary registers (R & Z), memory buffer register (MBR), memory data register (MDR), all are 16 - bit wide. The register Q is used to hold the most significant 27 - bits of an RR - 4 multiplication. All the general purpose RR - 4 registers are 27 - bit wide. The Z register holds all the 16 - bit immediate operands of the instructions. The flag register is a 2 - bit register used to hold the overflow and zero flag.

The *register block* houses all the four general purpose registers and their RR - 4 counterparts. When a LOAD instruction is decoded the corresponding 16 - bit register is being enabled and the data is transferred to that register. It is then converted to RR - 4 number by the binary to RR - 4 converter module and then stored in the equivalent RR - 4 register. Fig 4 shows the details for a single register.

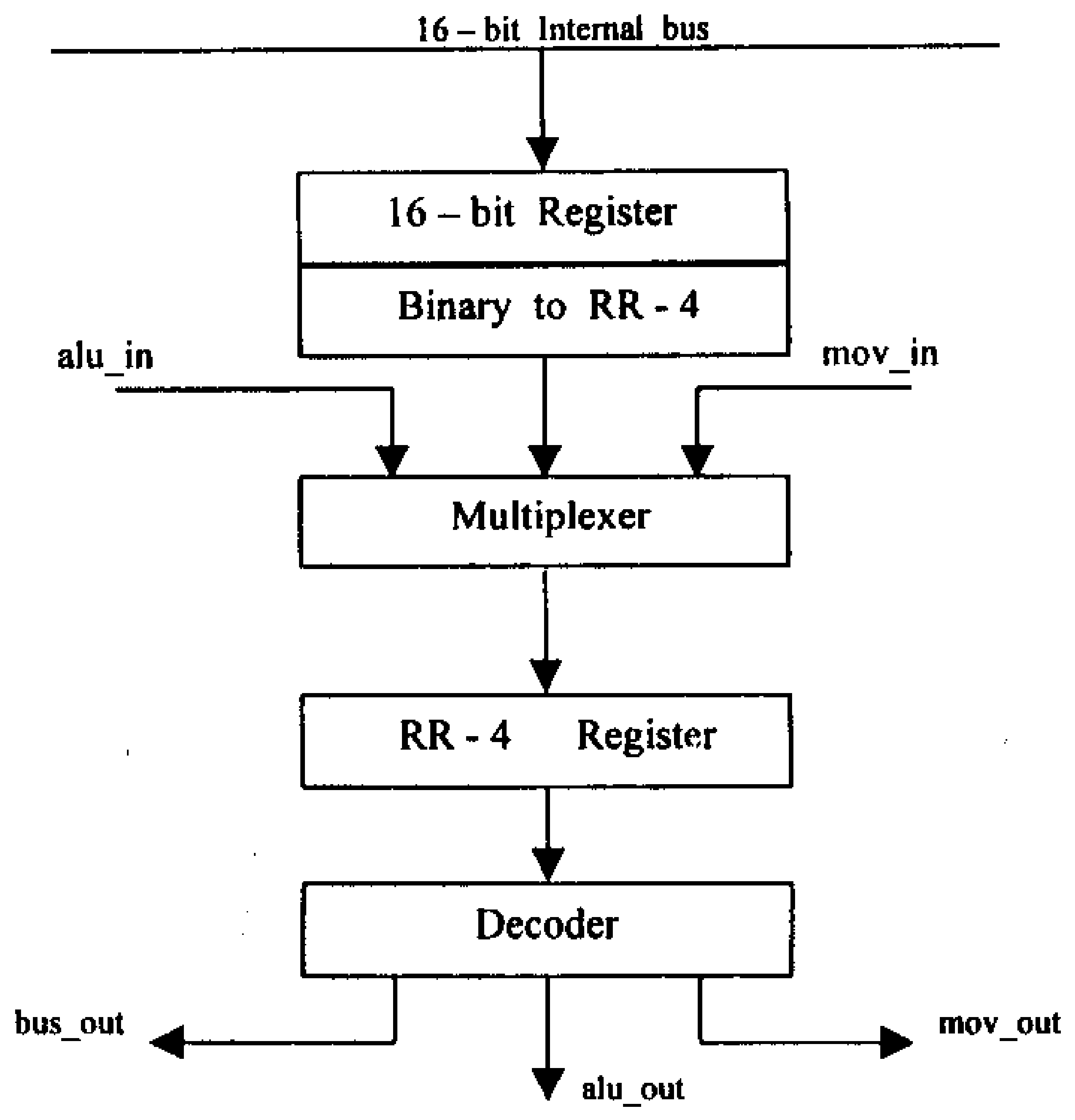


Fig. 4

The conversion of the 16 - bit number to RR - 4 number and storing it in the equivalent RR - 4 register is done in a single clock cycle. A multiplexer selects among the three inputs. The alu_in input to the RR -4 register from the ALU after the completion of any ALU operation, where this register is the destination for storing.The mov_in input is the input during a

MOV operation in which an operand is moved into this register from another register. The third input is for the load instruction, which loads a data into the RR -4 register.

A decoder at the output also serves a similar purpose as above. The bus_out line transfers the contents of the RR - 4 register to the output unit during a STORE instruction. Alu_out line makes the contents of the RR - 4 register available to the ALU for arithmetic operations. Mov_out line is used to move contents of the RR - 4 register to the another register when this register is the source during a MOV operation.

Four sets of the above blocks are combined to make the entire Register block unit. The outputs and inputs to these four blocks are also controlled by externals multiplexers and decoders.

The output unit has the RR - 4 to binary re-conversion module and a temporary register to hold the reconverted 16 - bit binary number. Any input RR - 4 number is reconverted to binary during a STORE operation.

The ALU operates using RR - 4 logic for all arithmetic operations and this logic can be found in [1]. Each of the modules which perform an arithmetic operation are being selected using decoders and multiplexers which can be enabled or disabled. The ALU is implemented more like an conventional ALU. More details are found in the implementation section. The ALU also has the ROM modules which are required for addition operation.

The control unit which synchronizes all the register transfer operations and other control signals will be explained along with the implementation sections.

The RR - 4 coprocessor operates only on signed two's complement integer inputs. The design considered here hence does not include division operation which requires floating point representation. The instruction set does not include division operation. The instruction set is a minimum subset of 8085's arithmetic instructions. Each instruction is 2 byte long or 4 byte long. All immediate operands are two byte long.

There are four general purpose registers available to the user for programming and one index register which can be used as a pointer. Each of the 16 - bit registers is converted into an equivalent RR – 4 register internally, i.e, loading a 16 – bit register with a number is same as loading the RR - 4 register with the equivalent RR – 4 number. The subsequent operations of addition, multiplication etc. are performed using RR – 4 logic [1]. The four general purpose registers are AB, BB, CB, DB and their RR – 4 counterparts are AR, BR, CR, DR. IX register is the index register holding the pointer.

Data transfer instructions

The data transfer instructions move data from one register to another.

LOAD reg, addr - *reg* can be any of the four general purpose registers and *addr* is the 16 – bit address where the operand can be found.

MOV reg 2 , reg 1 - *reg 2* destination register and *reg 1* is the source register for data transfer.

MVI reg, immediate operand - 16 – bit immediate operand is moved to the *reg*.

LDX addr - Loads a pointer into the index register.

STORE reg, addr - Contents of the *reg* are stored in the location *addr*.

MVX reg - Contents of the memory location pointed by the index register is moved into *reg*.

No status flag is affected in the above operations.

Arithmetic instructions

The set of arithmetic instruction include addition, multiplication and increment instructions.

ADD reg 2, reg 1 - Contents of *reg 1* is added with that of *reg 2* and the result of the operation is in *reg 2*.

ADI reg, immediate operand - 16 – bit immediate operand is added with the contents of *reg* and the result is available in *reg*.

SUB reg 2, reg 1 - Contents of *reg 1* are subtracted from the contents *reg2* and the result is in *reg 2*.

MUL reg 2, reg 1 - Contents of *reg 1* are multiplied with that of *reg 2* and the result can be found in *reg 2* and another internal register.

INC reg - Increments the value of *reg* by one.

DNC reg - Decrements the value of *reg* by one.

INX - Increments the index register by one to point to the next location.

NEG reg - Negates the contents of *reg*.

SQR reg - Squares the contents of *reg*.

CLR reg - Sets the contents of *reg* to zero.

All the instructions affect the zero flag, overflow flag and sign flag. The subtraction and multiplication operations also have immediate operand instructions.

Control instructions

In the case of RR – 4 logic due to redundant representation, there is frequent chances for overflow eventhough the binary equivalent of the RR – 4 number may be within limits. Hence, instruction which transfer control on detecting overflow is included. It is also a good practice to test for overflow at the end of every arithmetic operation. Since the carry out of any arithmetic operation is not used, no instruction for checking carry is included.

JZ addr - Jump to the location *addr* if the zero flag is set, $z = 1$.

JNZ addr - Jump to the location *addr* if the zero flag is not set, $z = 0$.

JO addr - Jump to the location *addr* if the overflow flag is set , $Ov = 1$.

JNO addr - Jump to the location *addr* if the overflow flag is not set,$Ov = 0$.

Timing and Control

The coprocessor is run by the system clock which also runs the host processor. Hence it is easy to synchronize the host with the coprocessor. Any signal either from the host to the coprocessor gets activated only when the clock is high. The basic instruction cycle is divided into Fetch and Execute phases. The decoding of instruction is integrated with the execution phase. When the host processor invokes the coprocessor, the initial few clock cycles are used for the transfer of the start address and end address of the code segment of the coprocessor. Here we will discuss the micro-operation sequence for the various instructions in the instruction set.

When the host encounters the START instruction, it issues the READY signal to the coprocessor. The following sequence of micro-operations is executed.

```
$T_1$ :  $Ack \leftarrow 1$  ( in response to $Ready$ signal )
$T_2$ :  $MBR \leftarrow Start\ Address\ (16\text{ - bit}),\ Ack \leftarrow 0$
$T_3$ :  $Busy \leftarrow 1, PC \leftarrow MBR$
$T_4$ :  $Busy \leftarrow 0, MBR \leftarrow End\,Address$ (16 - bit)
$T_5$ :  $R \leftarrow MBR$
```

Each T_i is a clock cycle. BUSY signal goes high so that the host can delay the input of the end address while the start address is begin processed.

Once the coprocessor has been invoked, it is ready for independent operation by stealing bus cycles and decoding its own instructions. Every instruction is preceded by the FETCH phase, hence the later micro-operation sequences will exclude it. The following is the sequence of micro-operations for fetch phase.

```
$T_1$ : $MAR \leftarrow PC$
$T_2$ : if $(GT)$
            $RD \leftarrow 1, M \leftarrow 1$ go to $T_3$
        else
            $RQ \leftarrow 1$ go to $T_5$
```

```
              T_3 : $RD \leftarrow 0, M \leftarrow 0, MBR \leftarrow [MAR]$
              T_4 : $IR \leftarrow MBR, PC \leftarrow PC + 1$
WAIT STATE    T_5 : if(GT)
                        $RQ \leftarrow 0$, go to $T_2$
                    else repeat $T_5$.
```

During fetch operation, the coprocessor should have the control of the bus else it enters a wait state.

LOAD reg, addr

```
$T_1$ : $MAR \leftarrow PC$
$T_2$ : $RD \leftarrow 1, M \leftarrow 1, MBR \leftarrow [MAR]$
$T_3$ : $MAR \leftarrow MBR, RD \leftarrow 0, M \leftarrow 0$
$T_4$ : $RD \leftarrow 1, M \leftarrow 1, MBR \leftarrow [MAR]$
$T_5$ : $reg \leftarrow MBR, PC \leftarrow PC + 1$
```

STORE reg, addr

```
$T_1$ : $MAR \leftarrow PC$
$T_2$ : $RD \leftarrow 1, M \leftarrow 1, MBR \leftarrow [MAR]$
$T_3$ : $MAR \leftarrow MBR, RD \leftarrow 0, M \leftarrow 0$
$T_4$ : $MBR \leftarrow reg$
$T_5$ : $WR \leftarrow 1, M \leftarrow 1, PC \leftarrow PC + 1$
```

In both of the above instructions the memory operation will be successful only if the *GT* pin is high. The other instructions like ADD, SUB,MUL require only a single clock cycle in which the appropriate control signals are generated to perform the required operation.

LDX addr

```
$T_1$ : $MAR \leftarrow PC$
$T_2$ : $RD \leftarrow 1, M \leftarrow 1, MBR \leftarrow [MAR]$
$T_3$ : $IX \leftarrow MBR, RD \leftarrow 0, M \leftarrow 0$
```

MVX reg

```
$T_1$ : $MAR \leftarrow IX$
$T_2$ : $RD \leftarrow 1, M \leftarrow 1, MBR \leftarrow [MAR]$
```

T_3 : $reg \leftarrow MBR, RD \leftarrow 0, M \leftarrow 0$

ADI reg, immediate operand

T_1 : $MAR \leftarrow PC$
T_2 : $RD \leftarrow 1, M \leftarrow 1, MBR \leftarrow [MAR]$
T_3 : $reg \leftarrow MBR, RD \leftarrow 0, M \leftarrow 0$

The other immediate instructions have similar timing sequences and can be easily written down.

Here we will discuss about the implementation details of the project. Since the processor is a simple one to look at, the coding for it is very large. We shall start with the basic blocks first and then move on to construct complex circuits out of it. This implementation has been done in Verilog language [2]. Verilog is very much C - like. The basic building block of the language is a *Module*. A module represents a black box with some inputs and outputs, if any. Every circuit is described as a module in terms of its basic functionality. Most of the logic used for the ALU can be found in [1], where the RR - 4 arithmetic has been discussed in detail. The input to the coprocessor are 16 - bit binary numbers and the output is also 16 - bit number. All intermediate operations are done using RR - 4 logic.

Binary to RR - 4 Conversion

The algorithm for converting binary to RR - 4 is discussed well in [1]. For any 16 - bit input number we assume that the number is represented in signed two's complement form. Hence the *msb* is the sign bit and the remaining bits constitute the data bits. The following logic converts a 2- bit binary number with its sign bit and sign bit from the previous digit position into an RR - 4 equivalent number. The first step is to convert each two bit binary number along with its sign bit into an equivalent RR - 4 carry and sum digit. Let $s_a\, a_0\, a_1$ be the input two bit binary number and s_a the *sign* bit. Also, let s_p be the sign bit of the *previous* digit position. The output RR – 4 *carry* is $s_c\, c_0\, c_1$ and *sum* is $s_d\, d_0\, d_1$.

$$
\begin{aligned}
L_1:\; & s_c = s_a s_p \\
& c_0 = 0 \\
& c_1 = s_a s_p + s_a' s_p' (a_0 + a_1) \\
& s_d = s_p' (a_0 + a_1) \\
& d_0 = a_0 a_1' + a_0 (s_p \oplus s_a) + a_0' a_1 (s_p \otimes s_a) \\
& d_1 = a_1
\end{aligned}
$$

The above logic has been implemented as an module. The inputs will be s_p,s_a a_0,a_1 and outputs are s_c, c_0,c_1 and s_d, d_0,d_1. $\otimes$ represents the ex-nor operation.

We use the basic block for building a bigger module which will convert an input 16 - bit signed bit binary number into RR - 4 carry vector and sum vector.

The generated RR - 4 *sum* and *carry* vectors have to be added using a carry propagation free adder to get the final equivalent 9 digit RR- 4 number.

RR - 4 to Binary conversion

The method to convert RR - 4 to binary has been described in [3]. The algorithm described in [4] has been used to add two binary numbers in O(log N) time. The logic for this has also been given in [4] and will not be dealt here. The output unit consists of a temporary register and RR - 4 to binary converter. Here a problem to be noted is Overflow. Since the *msb* of the reconverted number has to be the sign bit and the remaining 15-bits are the data bits. When the re-conversion is made, check for overflow has to be made. Hence, it is advisable to do a overflow check after any arithmetic operation.

Register block

This has been explained earlier in the architecture section and the implementation is similar to the diagram given there. Here the basic registers are implemented using the *always* construct of Verilog. Each register has an input enable , output enable. This sample code shows how a 16 - bit register is implemented.

```
// 16 - bit register
// Inputs are 16 - bit data, in_en - input enable , out_en - output enable
// Outputs are 16 - bit data
module reg_16(in,out,in_en,out_en);
input [15 :0] in;
input in_en,out_en;
output [15:0] out;
reg [15:0] temp;
always @(in or in_en) // Whenever the input or input enable changes
 begin
   if(in_en)
       temp = in;
 end
assign out = (in_en == 1'b1) ? temp : 16'h0000;
endmodule
```

The multiplexers and decoders also can be modeled in a similar fashion.

```
// Multiplexers  4 by 1
// Input  16 - bit  I1,I2,I3,I4, Output  Out, Selection - Sel
module Mux_4_1(I1,I2,I3,I4,Out,Sel);
input [15:0] I1, I2, I3, I4;
input [1:0] Sel;
output Out;
reg [15:0] temp;
always @(I1 or  I2 or I3 or I4 or Sel)
begin
   case(Sel)
      2' b00 :  temp =  I1;
      2' b01 :  temp =  I2;
      2' b10 :  temp =  I3;
      2' b11 :  temp =  I4;
   endcase
end
assign Out = temp;
endmodule

// Decoder 16 - bit
// Input 16 - bit In, Output O1,O2,O3,O4, Selection - Sel, Enable - En
module DEC_1_4(In,O1,O2,O3,O4,Sel,En);
input [15:0] In;
input [1:0] Sel;
output [15:0] O1,O2,O3,O4;

assign  O1 =  ((Sel = = 2'b00) && (En = 1'b1)) In : 16'h0000,
        O2 =  ((Sel = = 2'b01) && (En = 1'b1)) In : 16'h0000,
        O3 =  ((Sel = = 2'b10) && (En = 1'b1)) In : 16'h0000,
        O4 =  ((Sel = = 2'b11) && (En = 1'b1)) In : 16'h0000;
endmodule
```

Arithmetic and Logic Unit

The basic module of the adder can add four RR - 4 digits and the output is a carry and sum. This is done by using ROMs which store the

corresponding values of carry and sum and can be found in [1]. The size of each ROM is 4K × 6 bits. The input to each ROM is 12 bit through the address
lines and the output is the 6 – bit RR – 4 carry and sum. Since the size of the
ROM is very large, it cannot be directly implemented. Here we add the four RR – 4 number using normal signed bit addition to produce a signed 5 – digit binary number. This can be sent into the address lines of the reduced size ROM (32 × 6 bits) and the corresponding carry and sum can be obtained.

For example,

If 1, $\underline{2}$, 3, $\underline{1}$ are to be added then in the normal case, the input to the ROM is 12 bits. The underlined numbers represent negative RR – 4 numbers. But in this implementation we just add the four RR – 4 numbers using signed bit addition and the result is expressed as a five bit signed number. The sum of the above four numbers is 1 which is 00001 and the carry and sum is stored in this location. This reduces the size of the ROM for implementation purposes i.e, we require to make only 32 entries. But in reality the ROM is size is the same. The output of the ROM has to be rectified before it can be used for other operations, the logic of which can be found in [1]. Hence the basic adder consists of the ROM and the rectifying module.

Using this basic adder, the 9 – digit RR – 4 adder can be constructed. The ***increment*** and ***decrement*** operation can be easily implemented using the adder. The ***subtraction*** operation can be done using the adder by just negating the sign bits of the RR – 4 subtrahend. The ***multiplier*** uses the adders extensively for performing multiplication in O(log N) time using RR – 4 logic.

Control Registers

The control registers PC, IR, IX and MAR define the status of the coprocessor. Fig ** shows how these are connected to facilitate the transfer of data between them. The signals En_pc and Incr_pc are for enabling the PC register and incrementing the PC and similar is the case with IX and IR. Mar_out drives the address bus with the contents of MAR. The MBR which is a bi-directional port is also modeled using the *always* construct in slightly different manner.

Control Unit

The control unit is generally implemented using flip-flops and other state machine circuits. Here also the control unit for processor is implemented as a single state machine with various states. Based on the current state control signals are generated and given to the various components. The state machine is entirely specified for its next state and it runs continuously till the job is finished by the coprocessor. The implementation state diagram depends on the instruction which is begin currently executed. Based on this the different control

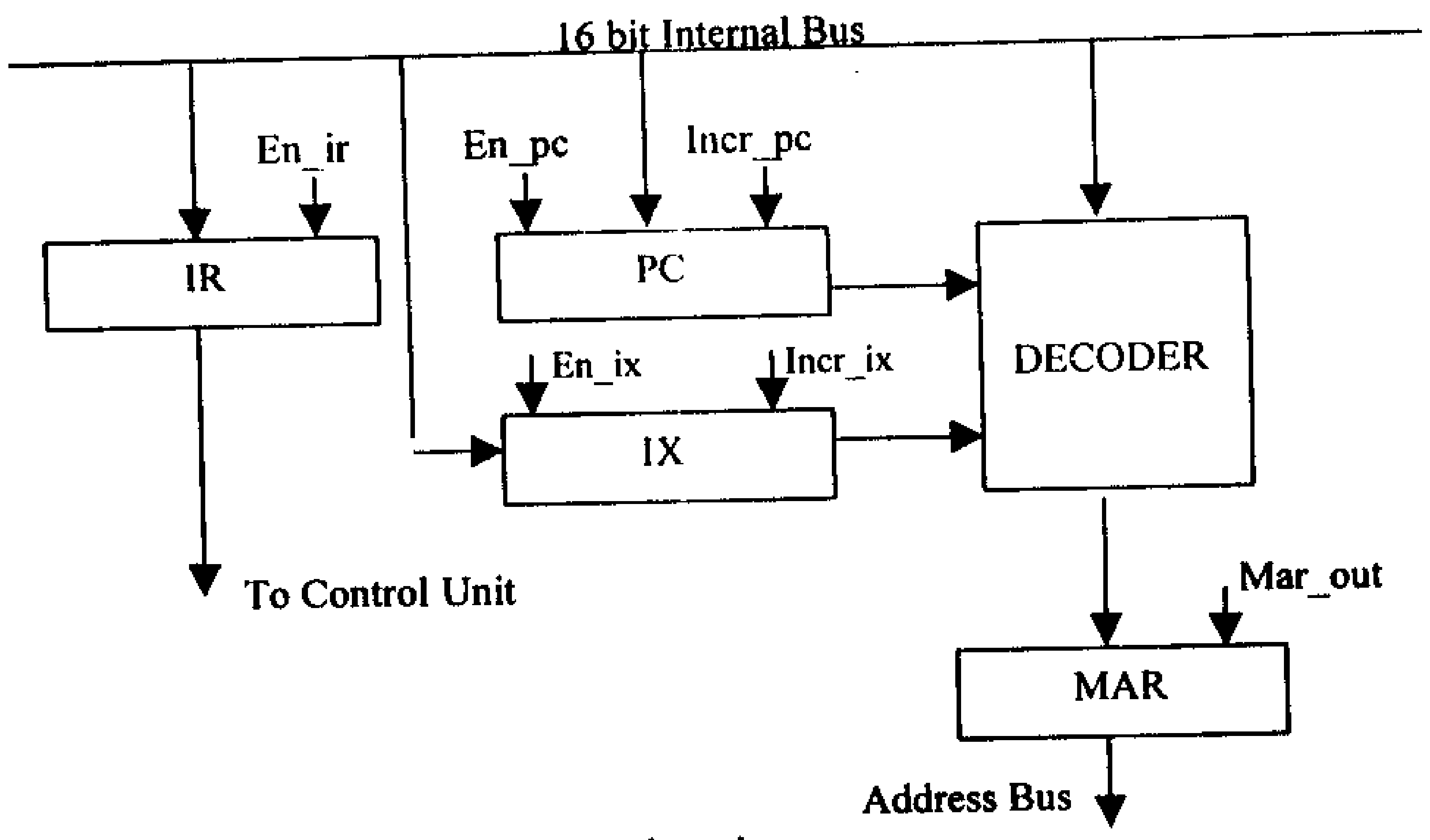
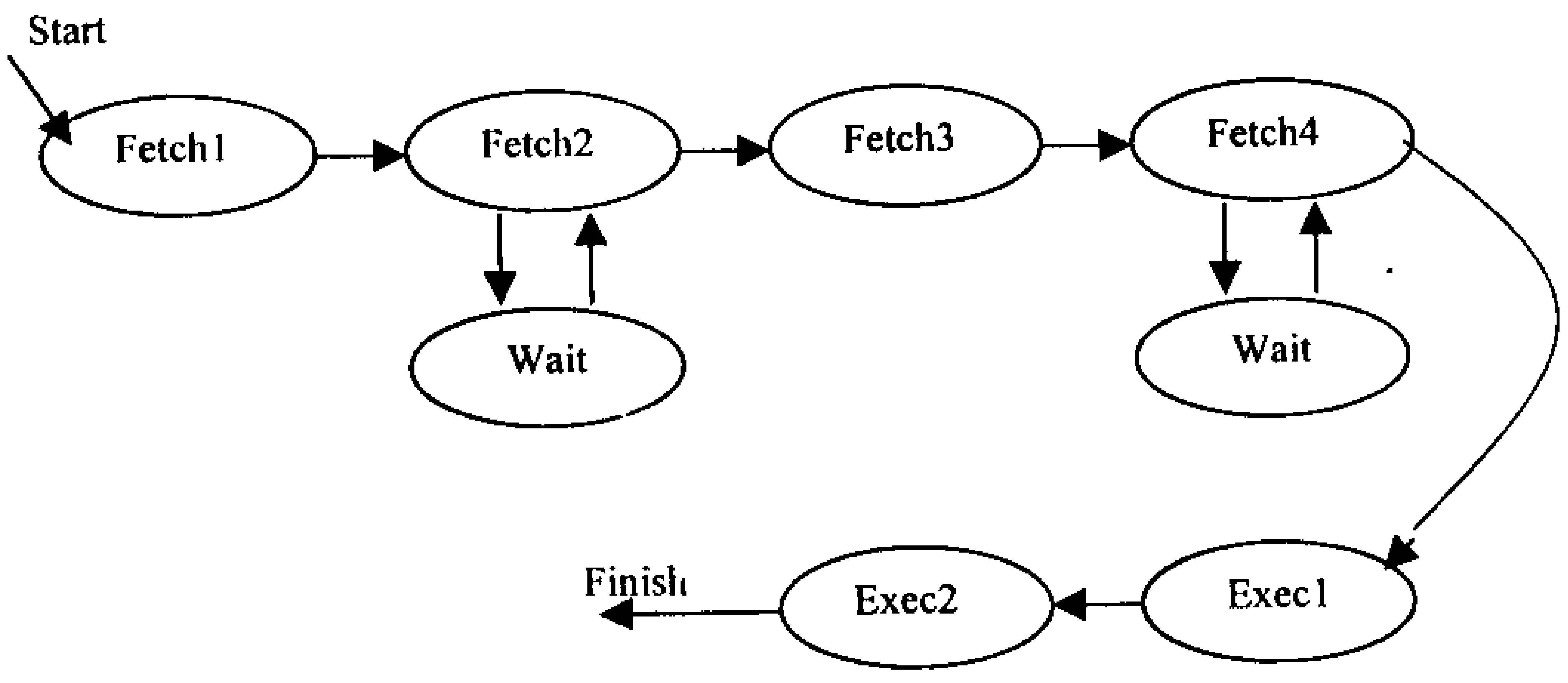


Fig. Control registers

signals are generated.

The following is a state diagram for *LOAD* instruction.



Each state represents a particular clock cycle in the micro-operation sequence. Based on these states the implementation uses another state machine which will produce the required control signals.

Results

The design explained in this report for RR – 4 coprocessor has been simulated using verilog language. The various modules - Binary to RR – 4 converter , RR – 4 to binary converter, the control unit , the register block ,all have been simulated . Later, the integration of all the modules into the RR – 4 coprocessor module also was simulated. The processor was able to add two signed bit input numbers. Exact simulation would require the external bus control logic which was excluded in this simulation. The simulation does not guarantee the timing constraints required for running of the coprocessor. Floating point operations also can be implemented in this design by including some extra modules.

Conclusion

The idea to implement the RR – 4 coprocessor is very good one. Computations can be performed at a faster rate especially, addition and multiplication. In normal processor, the delay is more for adding two binary numbers but the hardware required is less. In the case, of RR – 4 coprocessor, the various modules developed shows that the amount of hardware needed for the coprocessor is more. Hence, there is tradeoff between speed and space. Also it might be possible that the delays in these hardware elements can overwhelm the fastness in computation provided by the algorithm. A clear view can be obtained when the coprocessor is actually designed using the timing constraints.

References

[1] Mallika De, B.P.Sinha - Fast Parallel Multiplication Using Redundant Quarternary Number System ,Parallel Processing Letters.
[2] Verilog language FAQ - http://www.ovi.com/
[3] B.P.Sinha, P.K.Srimani - Fast parallel algorithm for binary multiplication and their implementation on systolic arrays, IEEE Transactions on Computers, Vol. 38, No.3, Mar. 1989, pp 424 – 431.
[4] Microcomputer Architecture – 8086/8088 Family – Liu and Gibson, Prentice Hall.

```
/* Module to convert two binary bit with sign bit and sign bit
from previous digit position.*/

module two_bit_rr4(a,sc,b,c);
 input sc;              /* sc is the sign bit from previous digit
position */
 input [2:0] a;      /* a[2] is the sign bit a[1],a[0] are the
binary bits */
 output [2:0] b,c; /* b and c are the RR - 4 carry and sum
respectively */

 assign b[2] = (a[2] & sc);
 assign b[1] = 0;
 assign b[0] = b[2] | ((~a[2] & (~sc)) & (a[1] | a[0]));
 assign c[2] = (~ sc) & (a[1] | a[0]);
 assign c[1] = ((~ a[0]) & a[1]) | (a[1] & (a[2] ^ sc)) |
((~a[1] & a[0]) & (~(a[2] ^ sc)));
 assign c[0] = a[0];
endmodule

/* Module to convert 16-bit binary to RR-4 CARRY VECTOR and
SUM VECTOR */
/* All input binary numbers are signed 2's complement number
*/

module bin_8_rr4(a,r,c);

 input [15:0] a;     /* input 16 bit binary number , a[15] is
the sign bit */
 output [23:0] r,c; /* r is sum vector and c is carry vector*/

 wire [16:0] t;

 assign t[15:0] = (a[15] == 1'b1) ? {1'b1,~a[14:0]} :
{1'b0,a[14:0]} ;
 assign t[16]   = (a[15] == 1'b1) ? 1'b0: 1'b1;

/* Instantiation of two bit binary to RR - 4 converters */

 two_bit_rr4 B1({a[15],t[1:0]},t[16],c[2:0],r[2:0]);
 two_bit_rr4 B2({a[15],t[3:2]},r[2],c[5:3],r[5:3]);
 two_bit_rr4 B3({a[15],t[5:4]},r[5],c[8:6],r[8:6]);
 two_bit_rr4 B4({a[15],t[7:6]},r[8],c[11:9],r[11:9]);
 two_bit_rr4 B5({a[15],t[9:8]},r[11],c[14:12],r[14:12]);
 two_bit_rr4 B6({a[15],t[11:10]},r[14],c[17:15],r[17:15]);
 two_bit_rr4 B7({a[15],t[13:12]},r[17],c[20:18],r[20:18]);
 two_bit_rr4 B8({a[15],t[15:14]},r[20],c[23:21],r[23:21]);

endmodule

/* Stimulus for testing the above module */

module stimulus;

reg [15:0] in;
wire [23:0] sum,carry;
integer i;

initial begin

     #1 in = 16'b0010111011011010;
     #1 in = 16'b0001101001101011;
     #1 in = 16'b1001010101101010;
     #1 in = 16'b0101101011010100;
  end

bin_8_rr4 B1(in,carry,sum);
initial begin

     i = $fopen("m1.dat");
   $fdisplay(i,"Input 16-bit signed number          RR-4 Carry
RR-4 Sum");
   $fmonitor(i,"  %b         %b           %b ",in,carry,sum);
 end

endmodule



□
/* Module for carry propagation free addition of two rr - 4
numbers */

module carry_prop_free_add(d,a,b,c,s);

input [2:0] d,a,b; /* d - carry from previous digit either
1,0,-1 */
                   /* a,b - RR - 4 digits to be added */
output [2:0] c,s;  /* c - Carry to next digit, s - sum of
d,a,b */
```

```
wire t1,t2;
wire [2:0] temp1,temp2,x,y;

assign t1 = (a[2] & d[2]) | (b[2] & (~d[2]));
assign temp1 = (t1 == 1'b0) ? a:b;


/* Module adds two RR-4 digits to produce a sum nd carry */
module rr4_two_digit_sum s1(d,temp1,x,y);

assign t2 = ~t1;
assign temp2 = (t2 == 1'b1) ? b:a;

rr4_two_digit_sum s2(temp2,y,c,s);
endmodule


/* Module to convert 16 - bit binary number to 9 digit RR - 4
number */

`include "bin_16_sum_carry.v"
`include "rr4_2digit_sum.v"
`include "carry_prop_free.v"

module final_rr4(p,s);

input [15:0] p;  /* Input 16 - bit binary number with a[15] as
sign bit */
output [26:0] s; /* Output 9 digit RR -4 number */

wire [2:0] t;
wire [26:0] l;
wire [23:0] r,c;

/*Module converts the input binary number to rr - 4 carry and
sum vector */

bin_8_rr4 B11(p,r,c);

assign  t  =(p[15]  ==  1'b1)  ?  3'b  001:3'b  000;  /*  Two's
complement addition adjustment */

/*Module performs the carry propagation free addition rr -4
sum and carry vectors */

carry_prop_free_add c1(3'b 000,r[2:0],t,l[2:0],s[2:0]);
carry_prop_free_add c2(l[2:0],r[5:3],c[2:0],l[5:3],s[5:3]);
carry_prop_free_add c3(l[5:3],r[8:6],c[5:3],l[8:6],s[8:6]);
carry_prop_free_add c4(l[8:6],r[11:9],c[8:6],l[11:9],s[11:9]);
carry_prop_free_add
c5(l[11:9],r[14:12],c[11:9],l[14:12],s[14:12]);
carry_prop_free_add
c6(l[14:12],r[17:15],c[14:12],l[17:15],s[17:15]);
carry_prop_free_add
c7(l[17:15],r[20:18],c[17:15],l[20:18],s[20:18]);
carry_prop_free_add
c8(l[20:18],r[23:21],c[20:18],l[23:21],s[23:21]);
carry_prop_free_add
c9(l[23:21],3'b000,c[23:21],l[26:24],s[26:24]);

endmodule

/* Stimulus for the above module */

module stimulus;

reg [15:0] a;
wire [26:0] s;
integer myfile;

initial
 begin
   #1 a= 16'b0011001101011011;
   #1 a= 16'b0110010100101001;
   #1 a = 16'b1010100100100l0;
   #1 a= 16'b1001010010101110;
   #1 a = 16'b010101111010111;
 end

final_rr4 r1(a,s);

initial
  begin
    myfile = $fopen("m3.dat");
    $fdisplay(myfile,"   Input   16   -   bit   binary   number
RR - 4 NUMBER     ");
    $fmonitor(myfile, "  %b              %b ",a,s);
  end
endmodule

/* ROM module takes an 12 - bit input and gives a 6 - bit
output*/
/* Implements the addition of 4 rr-4 digits */
`include "sign_add.v"     /* Module to add two sign-digit
number of 4-bits */
```

```
`include  "sign_add_4.v"      /*  Module  to  add  two  sign-digit
number of 5-bits */

/* Signed addition of input rr-4 numbers is done to reduce the
size of
   ROM for implementation purpose */
module rom_mem(a,b);

input [11:0] a; /* Input 12 - bits i.e, 4 rr - 4 digits */
output [5:0] b; /* Out put rr - 4 sum and carry */

reg [5:0] romadd[0:31]; /* ROM memory */

wire [3:0] temp1,temp2;
wire [4:0] rom_in;

sign_add s1(a[5:3],a[2:0],temp1);
sign_add s2(a[11:9],a[8:6],temp2);

sign_add_4 s3(temp1,temp2,rom_in);

initial
   begin
   $readmemb("in.dat",romadd);  /*  Binary  patterns  are  read
from the file */
   end                          /* in.dat to load the ROM with
appropriate values */

assign b = romadd[rom_in];     /* Output the carry and sum */

endmodule

/*Stimulus for the above module */

module stimulus;

reg [11:0] in;
wire [5:0] out;

integer myfile ;

initial begin
  #1  in = 12'b001011010101;
  #1  in = 12'b101010001110;
  #1  in = 12'b101010011111;
  #1  in = 12'b110001101011;
  #1  in = 12'b001010011101;
  end
     rom_mem r1(in,out);
initial begin
   myfile = $fopen("m4.dat");
  $fdisplay(myfile, " Input1   Input2   Input3   Input4   Carry
Sum");
  $fmonitor(myfile, "    %b       %b        %b       %b         %b
%b",in[11:9],in[8:6],in[5:3],
   in[2:0],out[5:3],out[2:0]);
end
endmodule

/* Module to perform signed addtion of two 4-bit numbers */

   module sign_add_4(a,b,c);
   input [3:0] a,b;
   output [4:0] c;
   wire flag;
   wire [1:0] t;
   wire [2:0]t1,t2;
   wire [3:0] t3;

   assign  t[1:0] = {a[3],b[3]};
   assign t1[2:0] = (t == 2'b10)? (~a[2:0] + 1) : a[2:0];
   assign t2[2:0] = (t == 2'b01)? (~b[2:0] + 1) : b[2:0];
   assign t3[3:0]= t1[2:0] + t2[2:0];
   assign flag = (~ t3[3]) & (a[3] ^ b[3]);
   assign c[4] =  (a[3] & b[3]) |((~t3[3])& (a[3] | b[3]));
   assign c[2:0] = (flag == 1'b1) ? (~t3[2:0] + 1) : t3[2:0];
   assign c[3] =  t3[3] & (~(a[3]^b[3]));

   endmodule


  /* Module for signed addition of two three bit numbers */

   module sign_add(a,b,c);

   input [2:0] a,b;
   output [3:0] c;
   wire flag;
   wire [1:0] t,t1,t2;
   wire [2:0] t3;

   reg [1:0] out;
   assign  t[1:0] = {a[2],b[2]};
assign t1[1:0] = (t == 2'b10)? (~a[1:0] + 1) : a[1:0];
assign t2[1:0] = (t == 2'b01)? (~b[1:0] + 1) : b[1:0];
assign t3[2:0]= t1[1:0] + t2[1:0];
assign flag = (~ t3[2]) & (a[2] ^ b[2]);
assign c[3] =  (a[2] & b[2]) |((~t3[2])& (a[2] | b[2]));
assign c[1:0] = (flag == 1'b1) ? (~t3[1:0] + 1) : t3[1:0];
assign c[2] =  t3[2] & (~(a[2]^b[2]));

endmodule
```