Chapter 9: Processer Design Examples

Syllabus

- Objectives
- Bus
- Data transfer
- General-purpose input and output
- Timers
- Universal asynchronous receiver and transmitter
- A simple CPU design

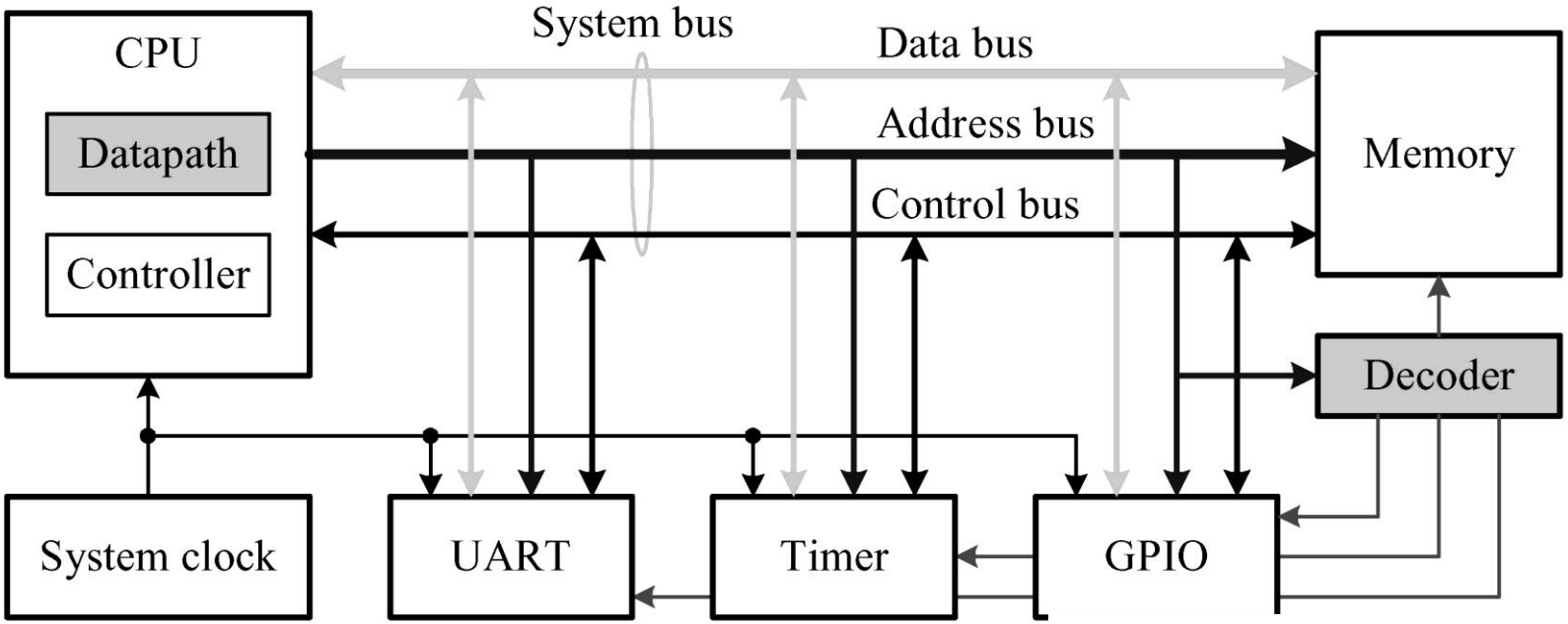
Syllabus

- Objectives
- Bus
  - A μp system architecture
  - Bus structures
  - Bus arbitration
- Data transfer
- General-purpose input and output
- Timers
- Universal asynchronous receiver and transmitter
- A simple CPU design

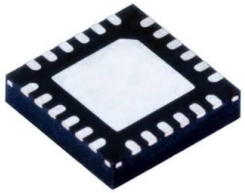
A Basic µP System

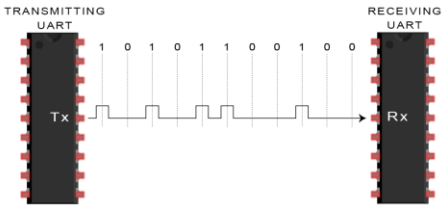
32-bit uP

16M SRAM



Clock Gen & Timer





ESP8266:

Add More GPIOs With MCP23008

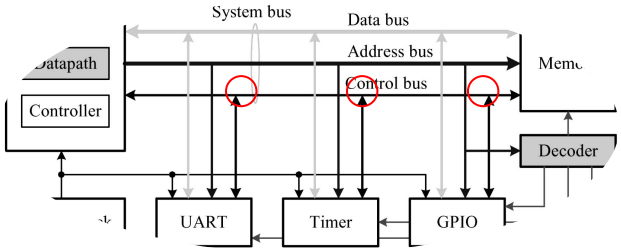
Syllabus

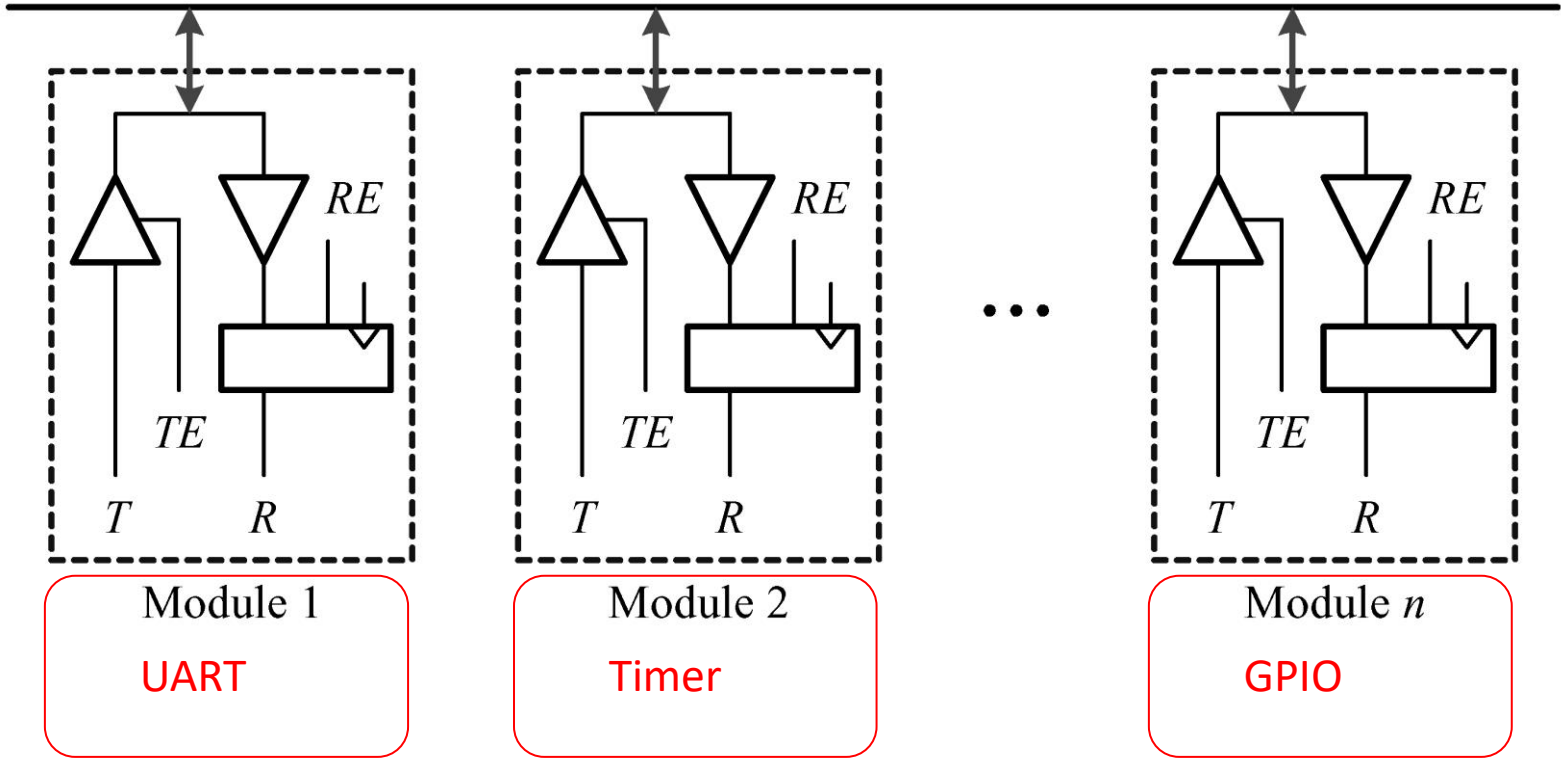
- Objectives
- Bus
  - A μp system architecture
  - Bus structures
  - Bus arbitration
- Data transfer
- General-purpose input and output
- Timers
- Universal asynchronous receiver and transmitter
- A simple CPU design

Bus Structures

- Tristate bus
  - using tristate buffers
  - often called bus for short
- Multiplexer-based bus
  - using multiplexers

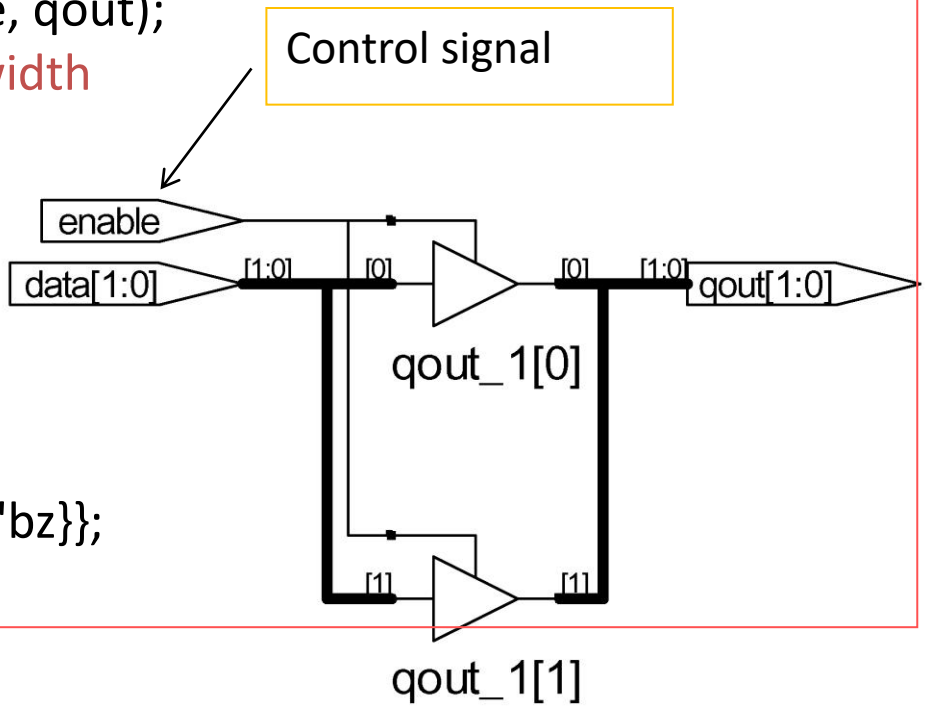
A Tristate Bus



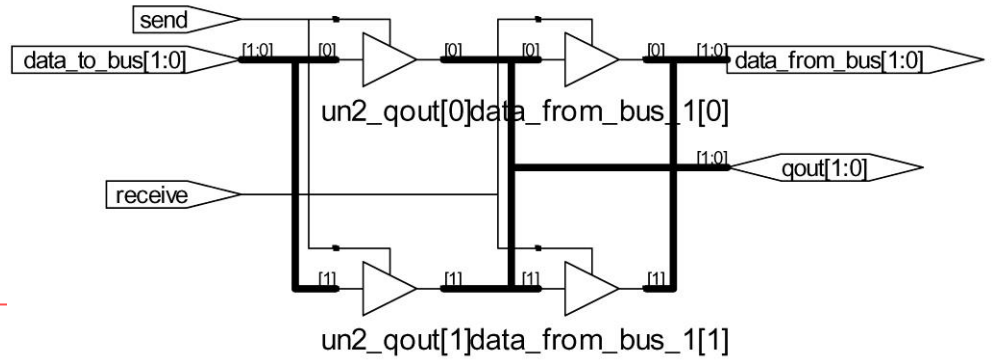


A Tristate Bus Example

```
// a tristate bus example
module tristate_bus (data, enable, qout);
parameter N = 2;   // define bus width
input   enable;
input   [N-1:0] data;
output [N-1:0] qout;
wire    [N-1:0] qout;

// the body of tristate bus
assign qout = enable ? data : {N{1'bz}};
endmodule
```

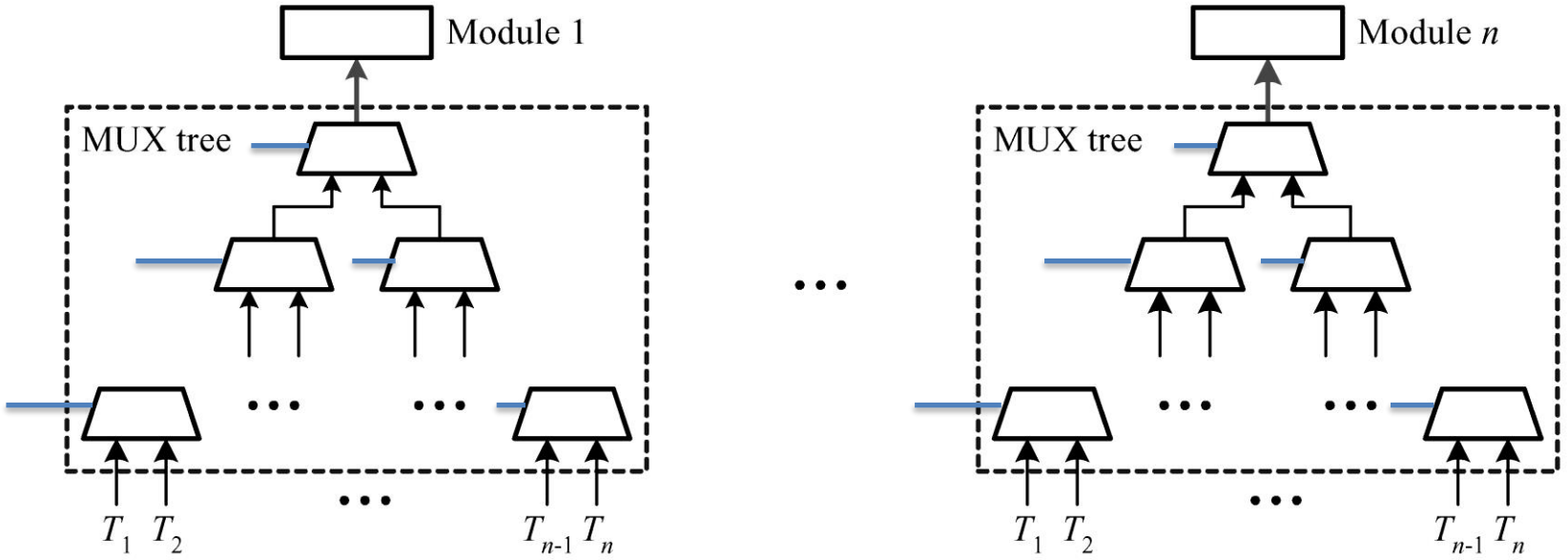


A Bidirectional Bus Example




```
// a bidirectional bus example
module bidirectional_bus (data_to_bus, send, receive, data_from_bus,
qout);
parameter N = 2;            // define bus width
input  send, receive;
input  [N-1:0] data_to_bus;
output [N-1:0] data_from_bus;
inout  [N-1:0] qout;        // bidirectional bus
wire   [N-1:0] qout, data_from_bus;
// the body of tristate bus
assign data_from_bus = receive ? qout : {N{1'bz}};
assign              qout = send ? data_to_bus : {N{1'bz}};
endmodule
```

A Multiplexer-Based Bus

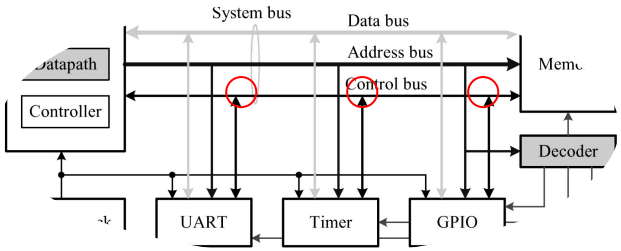


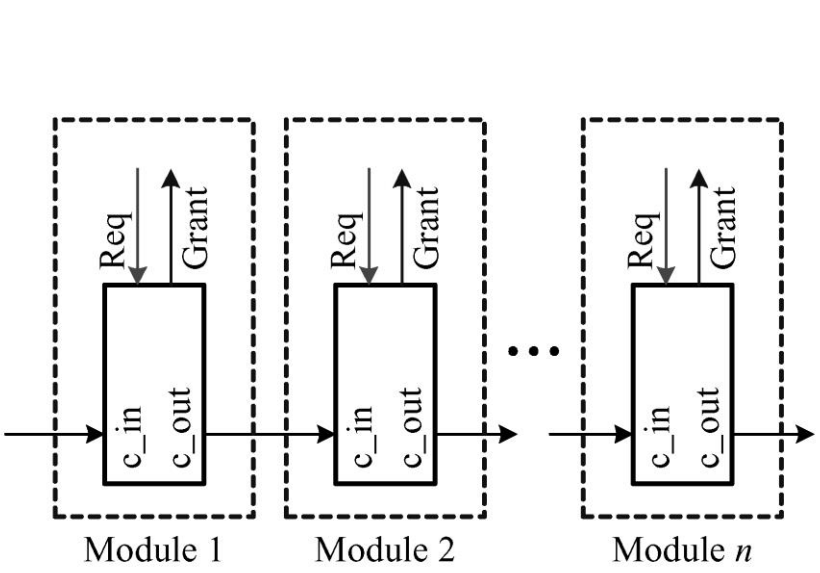
Syllabus

- Objectives
- Bus
  - A μp system architecture
  - Bus structures
  - Bus arbitration
- Data transfer
- General-purpose input and output
- Timers
- Universal asynchronous receiver and transmitter
- A simple CPU design

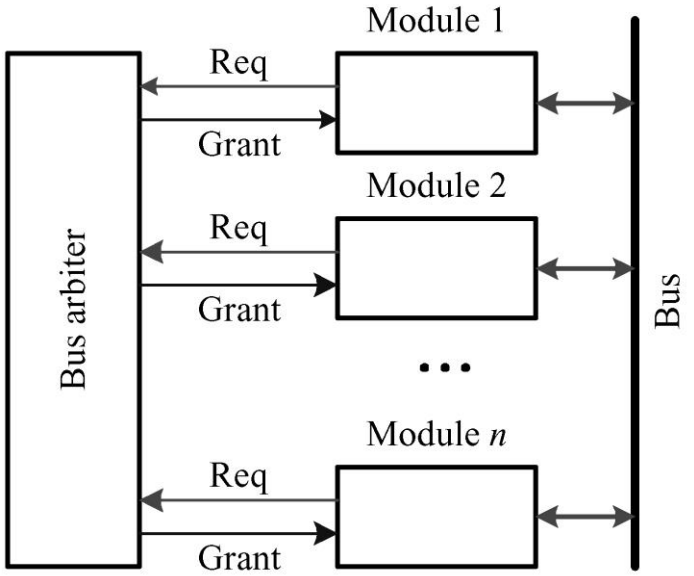
Daisy-Chain Arbitration

- Types of bus arbitration schemes
  - daisy-chain arbitration
  - radial arbitration



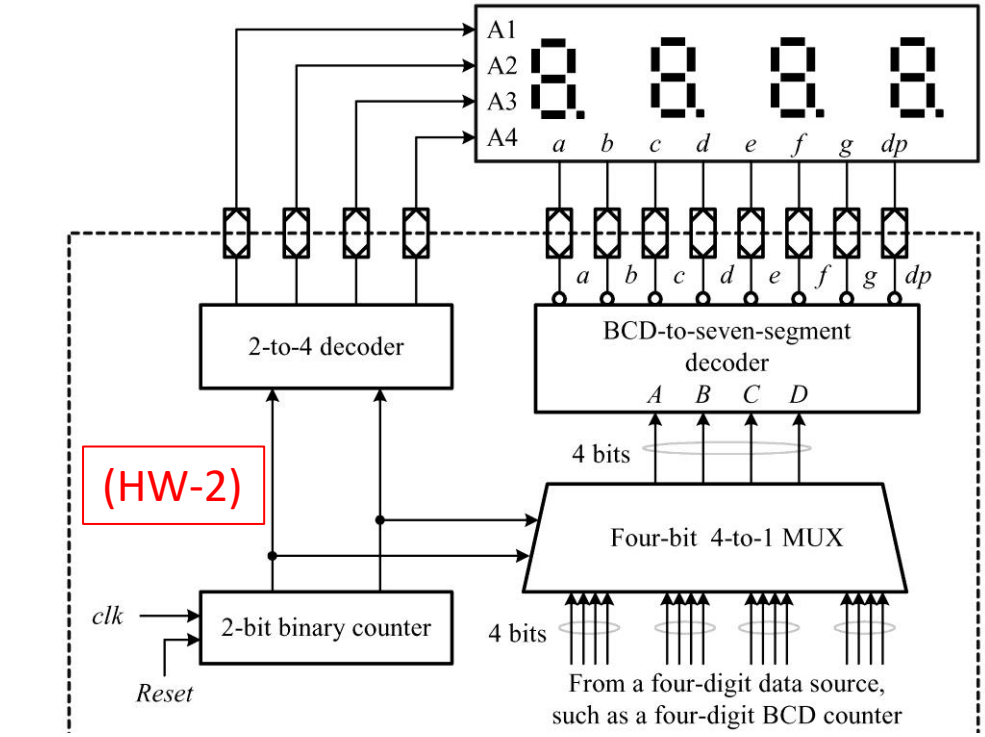


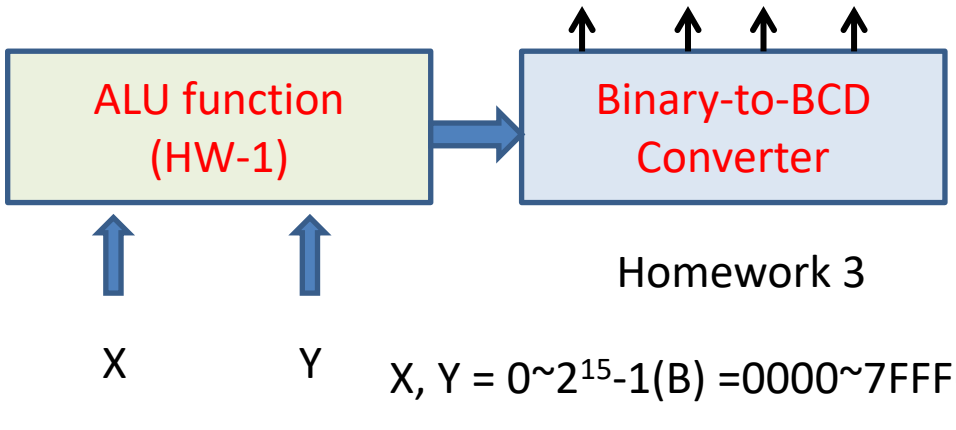
(a) Daisy-chain arbitration



(b) Radial arbitration

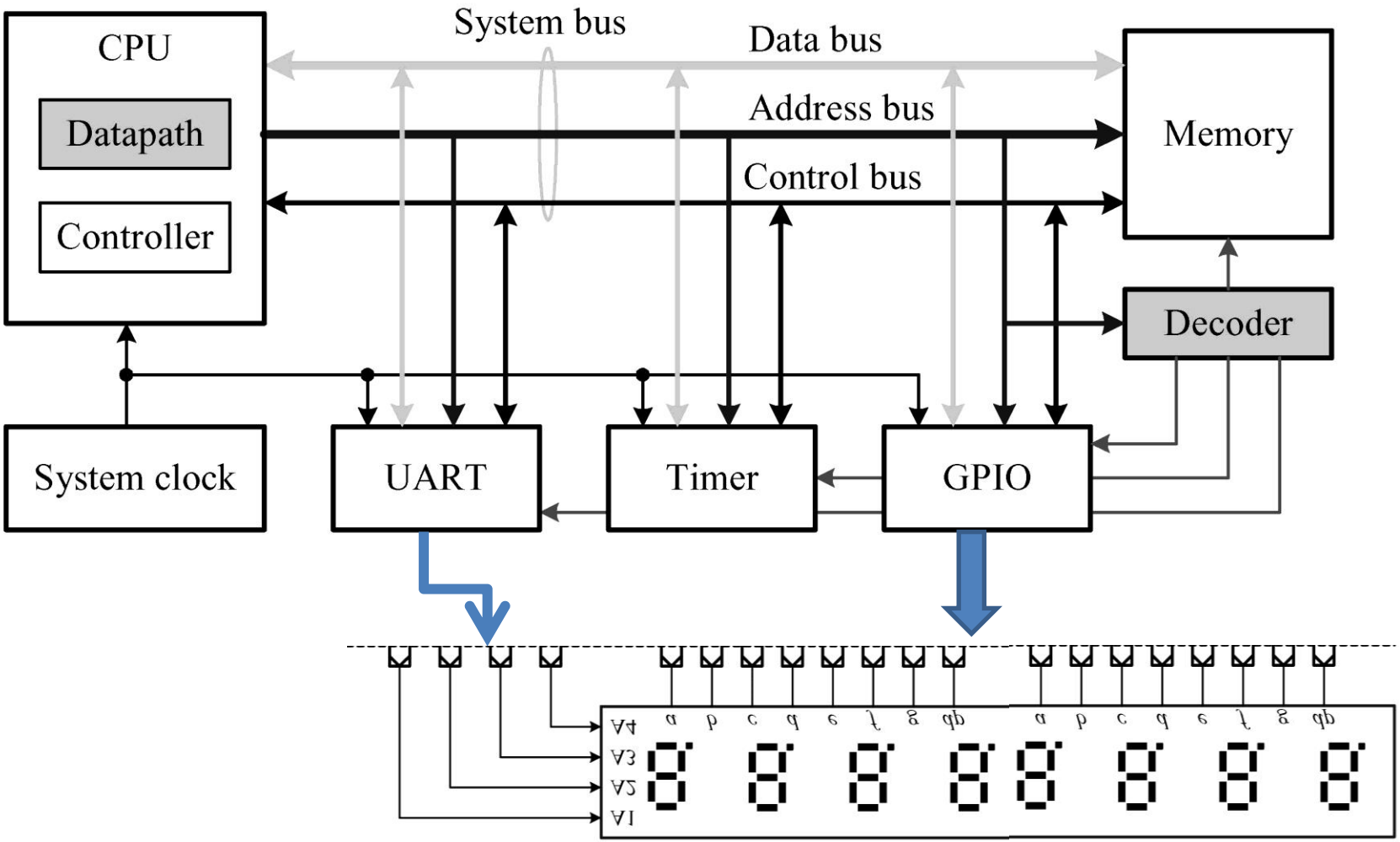
Homework 4



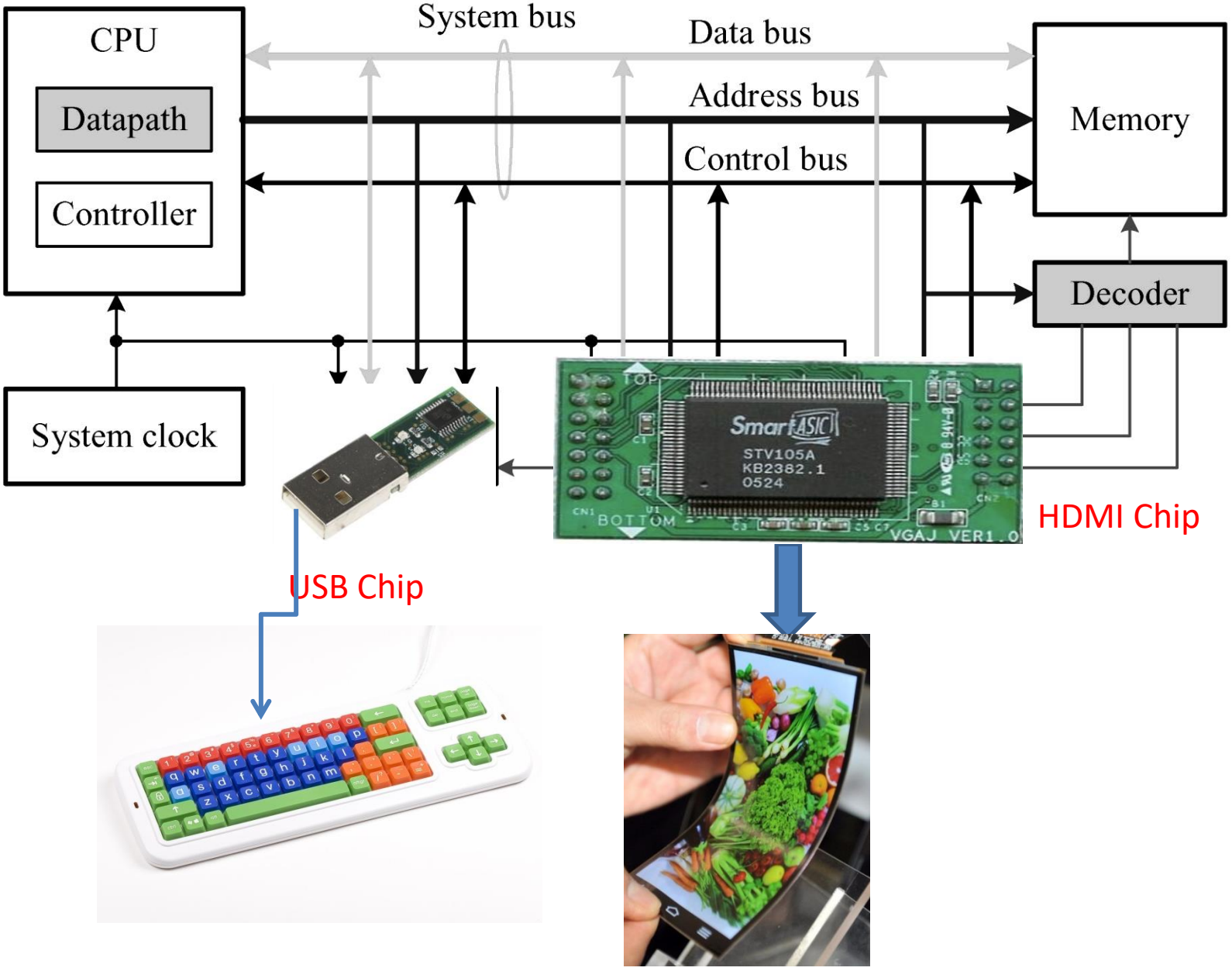


X, Y = $0 \sim 2^{15}-1$(B) =0000~7FFF(H)

uP for Homework Design



uP for Computer Design



Reference Design Examples

Basic Verilog Examples
https://web.csulb.edu/~rallison/Verilog_Examples_Table.htm

Useful verilog examples
http://jupiter.math.nctu.edu.tw/~weng/courses/IC_2007/PROJECT_MATH_CLASS1/3_DESIGN_COMPLEXITY/Verilog%20examples%20useful%20for%20FPGA%20&%20ASIC.htm

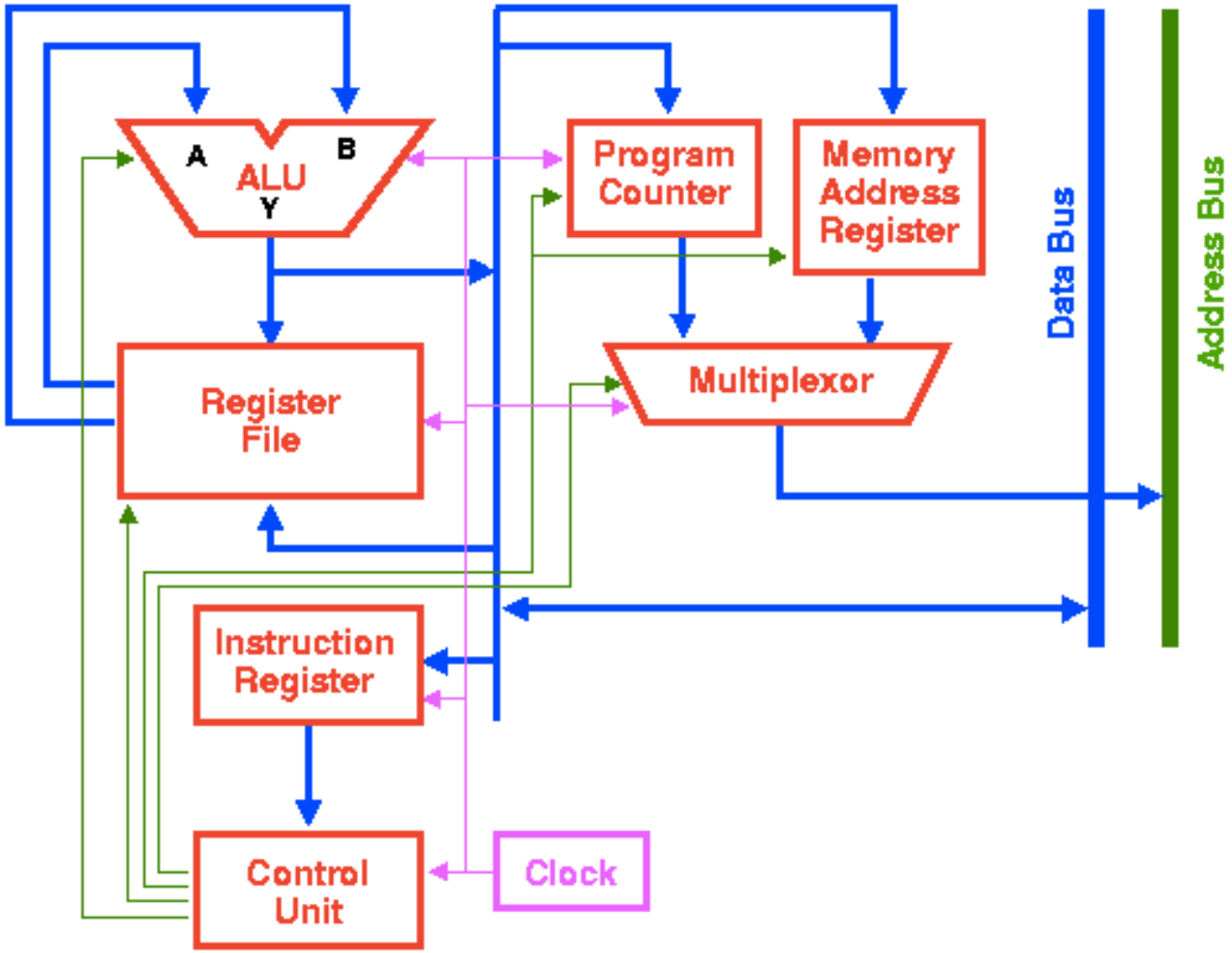
Design company
https://www.doulos.com/knowhow/verilog_designers_guide/

Design Examples
https://verilogguide.readthedocs.io/en/latest/verilog/designs.html

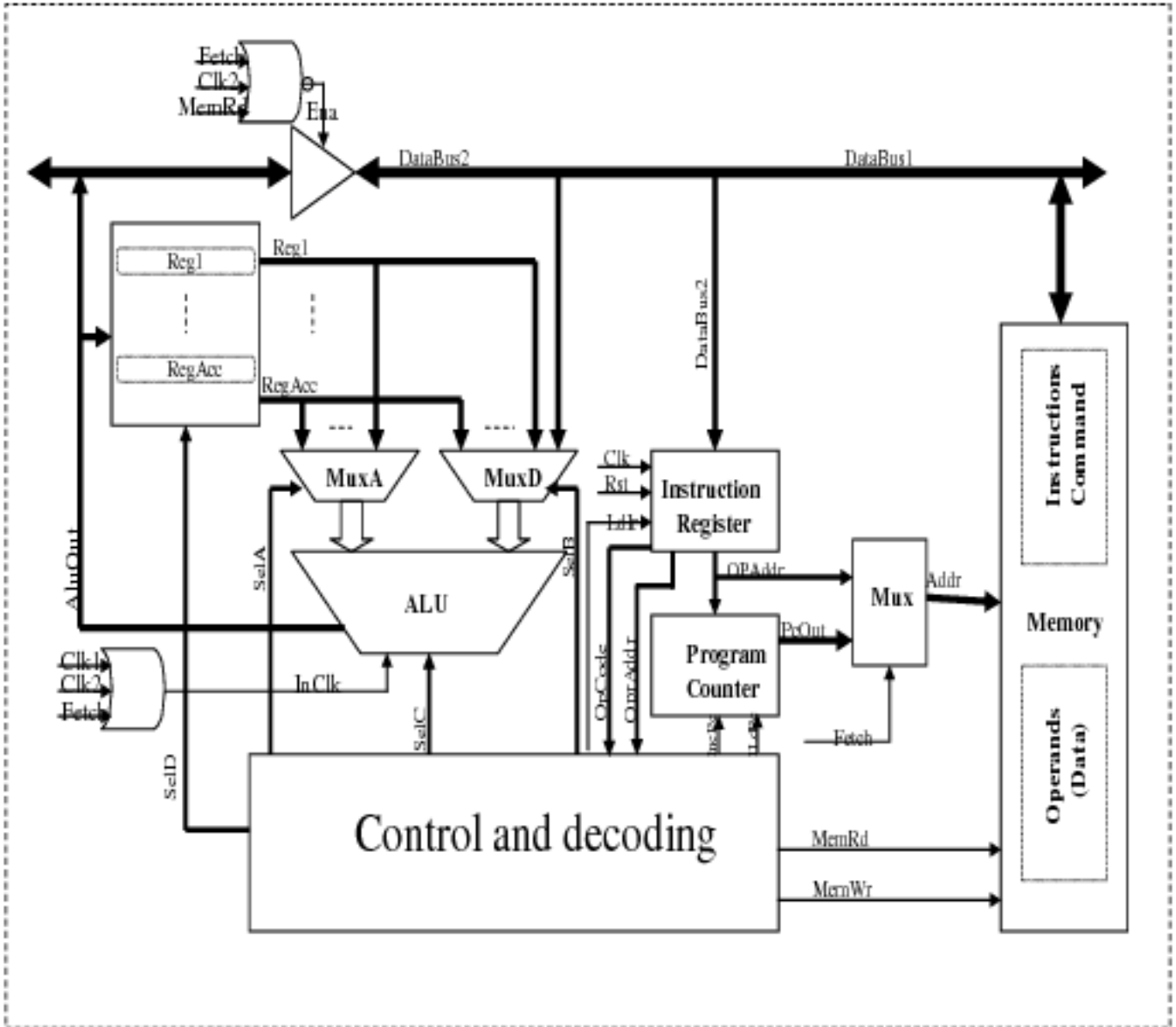
UART, Serial Port, RS-232 Interface
https://www.nandland.com/vhdl/modules/module-uart-serial-port-rs232.html

8-Bit RISC Processor I

https://linus5.blogspot.com/2016/07/risc-processor.html



8-Bit RISC Processor II



8-Bit RISC Processor III

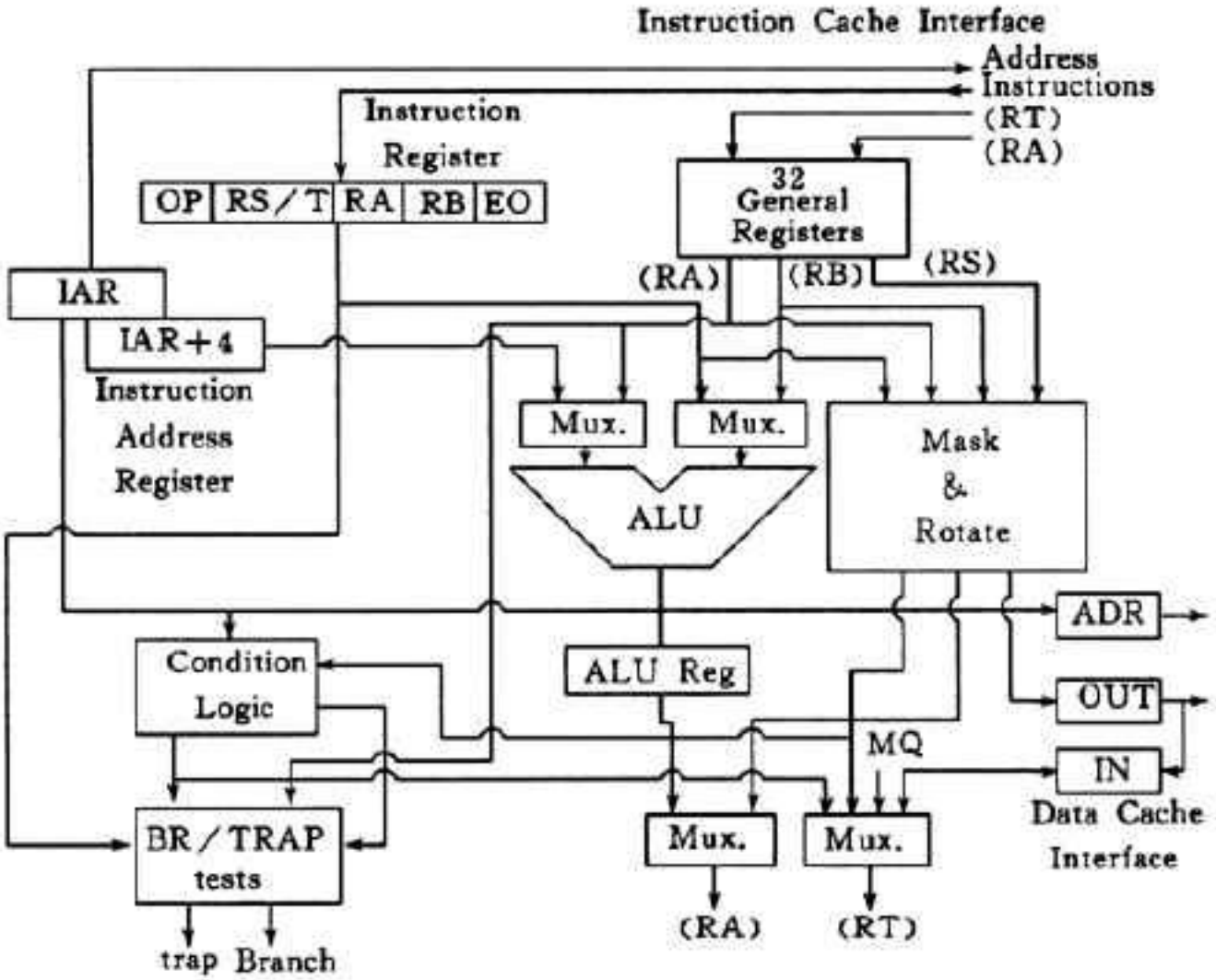


Fig. 3-2 IBM 801 Architecture

8-bit RISC CPU in Verilog

- https://electronicstopper.blogspot.com/2017/06/8-bit-risc-cpu-in-verilog.html

The Advantages of RISC will be discussed from a number of points of view

1. VLSI realization.
2. Computing speed.
3. Design cost and reliability.
4. HLL support.

網站畫面

8 bit RISC CPU in Verilog

June 20, 2017

You can download all my project files for this project HERE

下載CPU之Verilog程式

1. INTRODUCTION

1.1 RISC PROCESSORS

dropbox.com/sh/cegzwhcdsjbvz9t/AACwMmwoSyFMh9KoELC3iL2Ka?dl=0

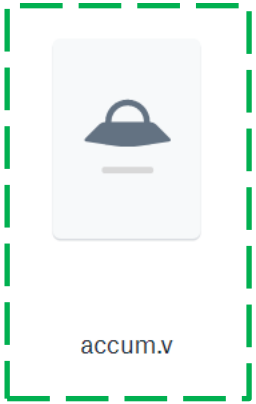
RISC_CPU

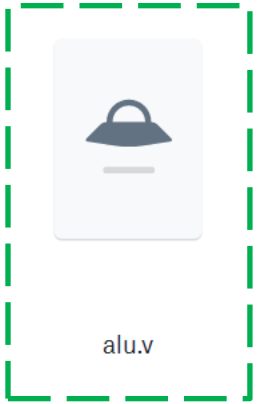
按名稱排序

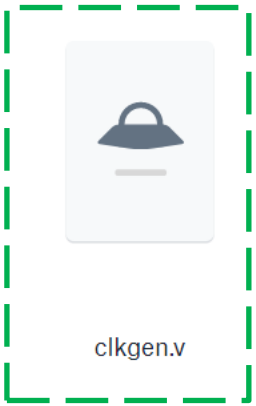

_ngo

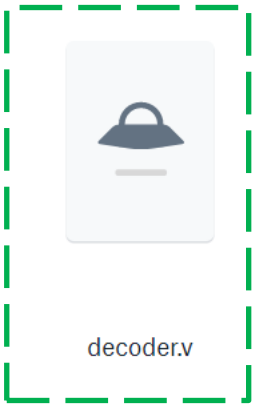

_xmsgs

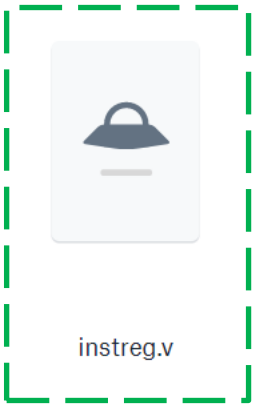

accum_summary.html

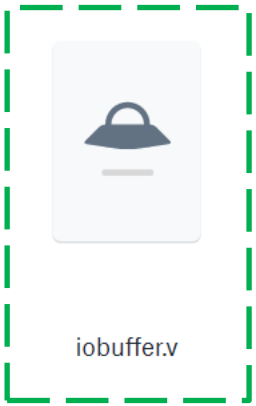

accum.v


alu.v


clkgen.v


decoder.v


instreg.v


iobuffer.v


iseconfig

memory.v

mux.v

netgen

par_usage_statistics.html

pc.v

RISC_CPU.gise

RISC_CPU.xise

Early computers had small and simple instruction sets, forced mainly by the need to minimize the hardware used to implement them. As the processor technology began to evolve beginning with 8,16 and 8,32 processors, the trend was to have more instruction capability in the processor. As a consequence, the control unit and instruction decoder became very complex. In some processors, the control unit occupies 50 to 60% of the chip area. This results in the reduction in space available to registers. Similarly, the instructions that access memory tend to slow down the code execution. This fact gave rise to RISC design philosophy that focused on a small set of frequently used instructions, thus simplifying the hardware design and improving the processor performance.

1.2 FEATURES OF RISC PROCESSOR

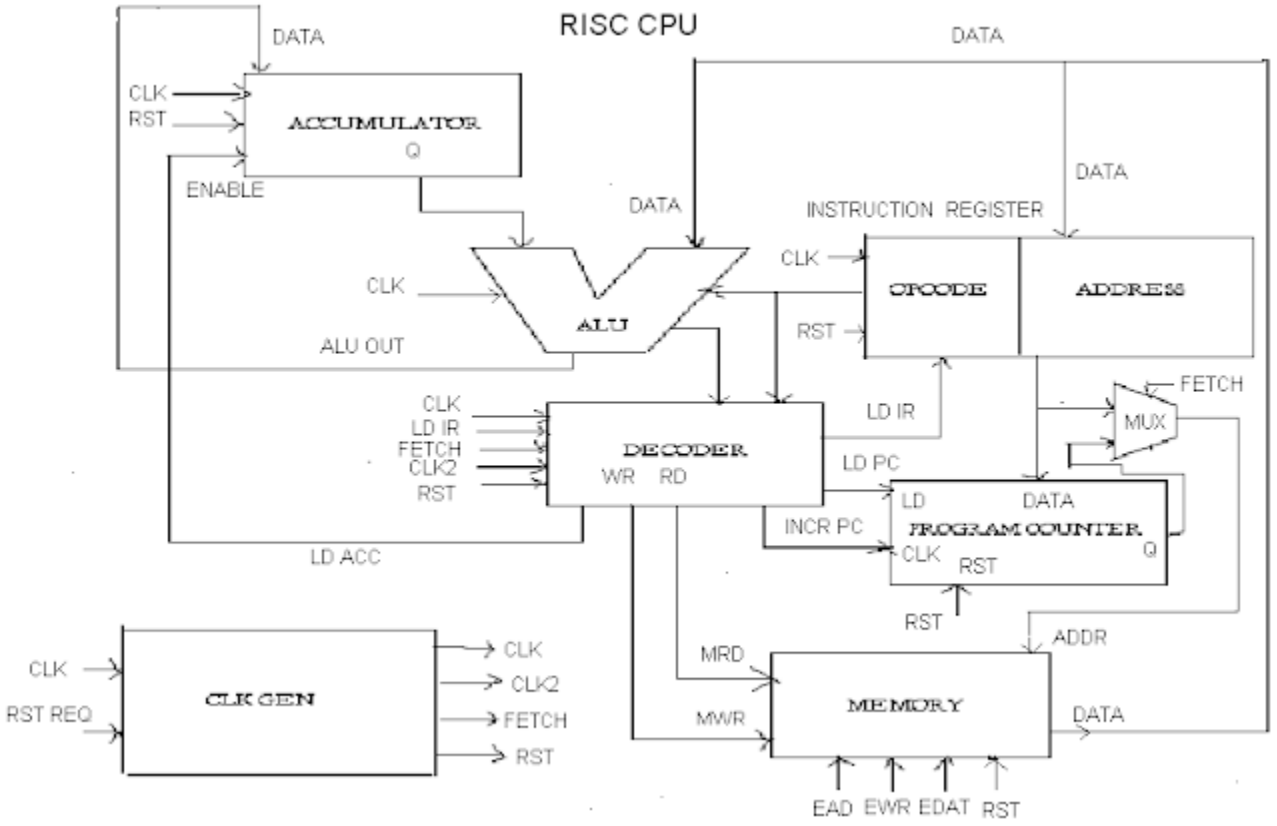
1. The number of instructions is minimized(less than 100).
2. The number of addressing modes is relatively few(less than 5).
3. The numbers of memory-reference instructions are minimized. Memory is accessed only by load and store instructions.
4. The processing is registered intensive, meaning it includes many more registers and most of the computing is performed using registers rather than memory. The number of registers range from 2 registers to more than 100 registers.
5. The instruction format is simplified and each instruction is executed in one cycle.
6. Design considerations include support for high level languages through appropriate selection of instructions and optimizations for compilers.
7. A relatively large, general purpose CPU register file.

risc_proc.doc

testbench.v

topmodule_bitgen.xwbt

8-bit RISC CPU Architecture



Top level module:

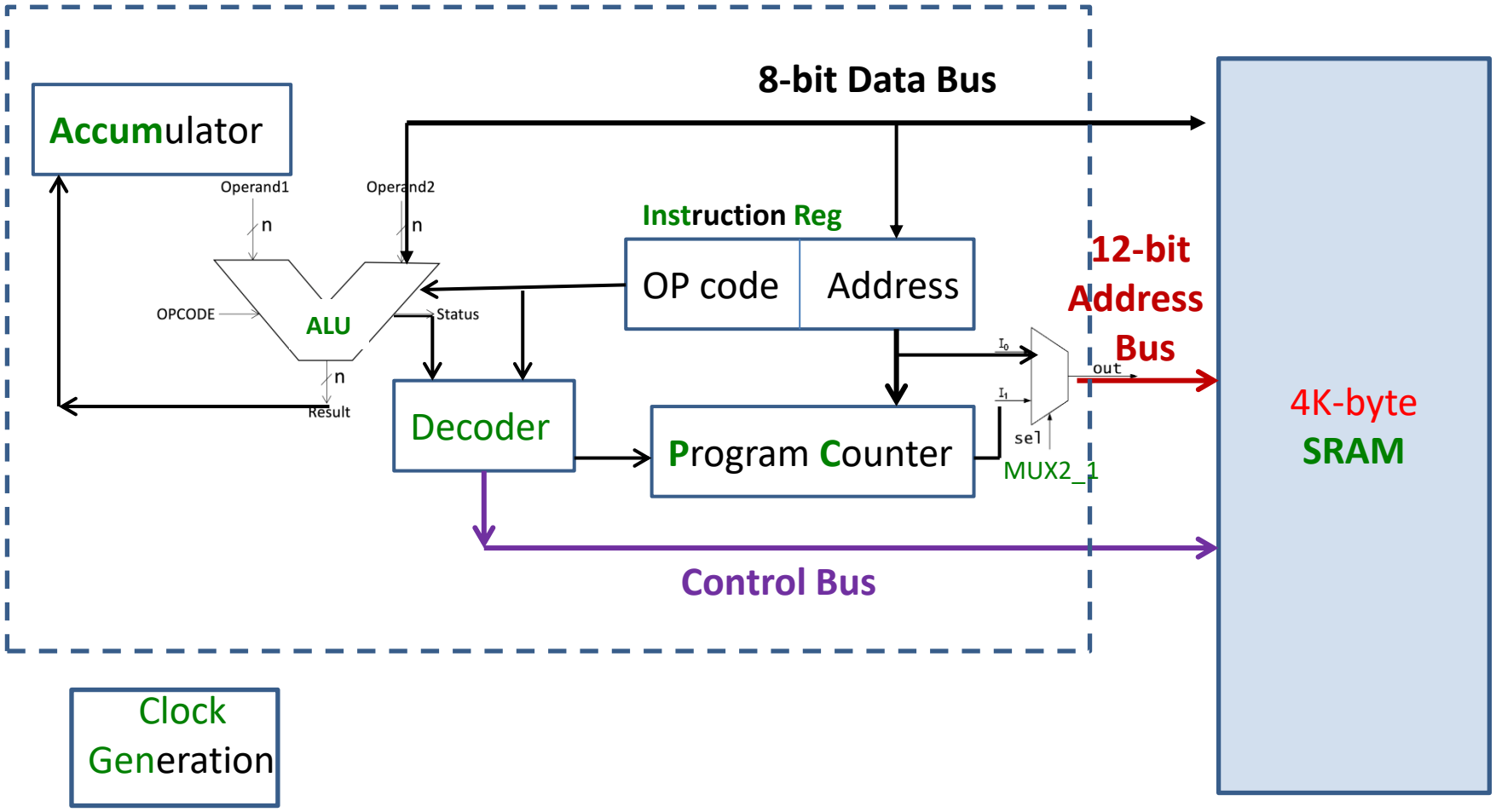
1. Clock Generator
2. Memory
3. Multiplexer
4. Program Counter
5. Instruction Register
6. Accumulator
7. Arithmetic and Logic Unit
8. Decoder
9. IO Buffer

Homework 5: 驗證9個module functions → 設計9個stimulus code
Homework 6: 修改testbench.v之測試資料

The major characteristics of a RISC processor are

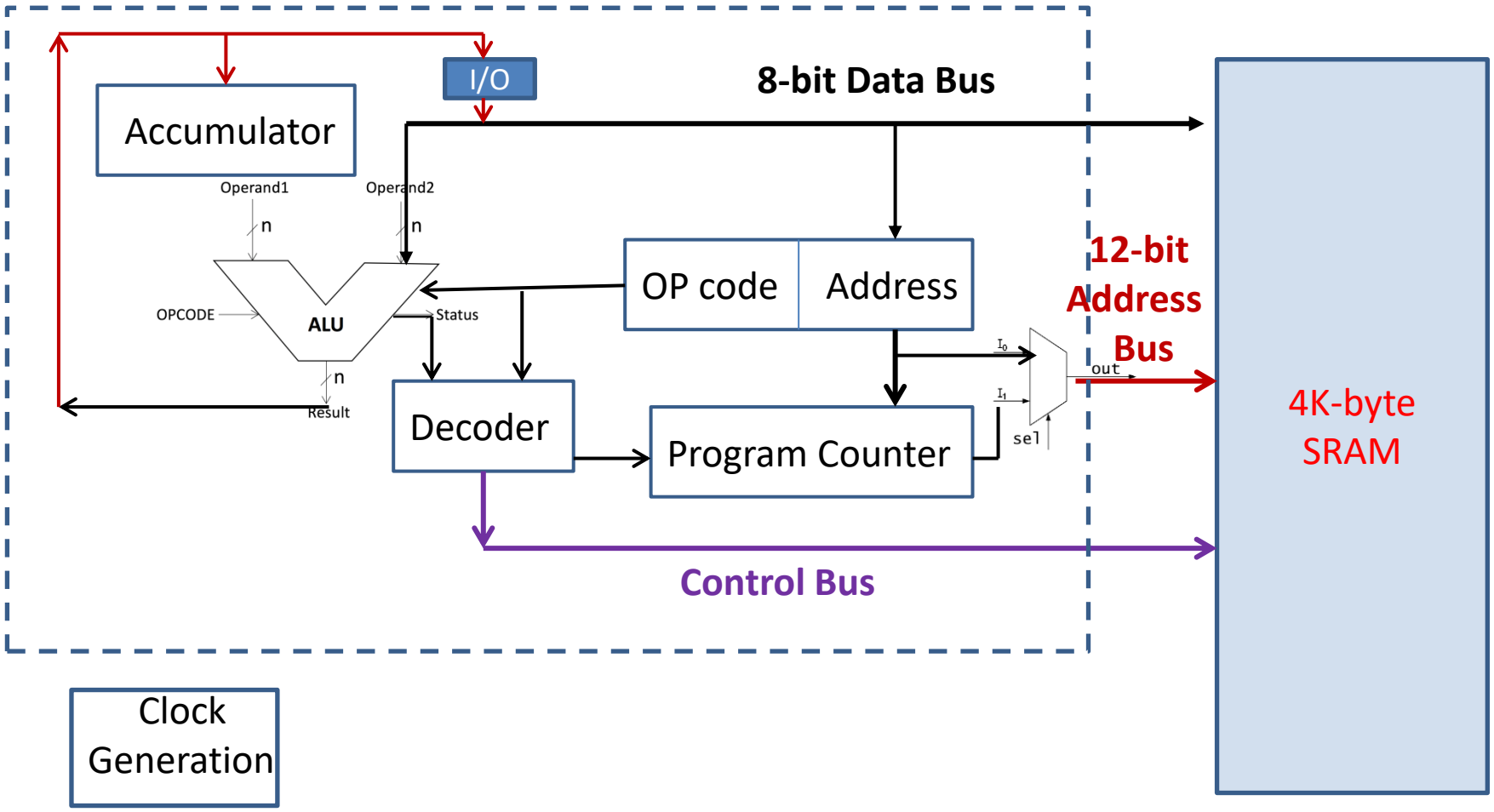
1. Relatively few instructions.
2. Relatively few addressing modes.
3. Memory access limited to load and store instructions.
4. All operations done within the registers of the CPU.
5. Fixed –length, easily decoded instruction format.
6. Single –cycle instruction execution.

Architecture of 8-Bit RISC CPU



```
module topmodule(clk,rstreq,ewr,ead,edat,zr);

input clk, rstreq, ewr;          input [4:0] ead;
input [7:0] edat;                output zr;          wire zr;    wire ldac, ldir;
wire clk1, clk2, fch, rst;       wire [7:0] alu_out, acc_out;    wire [7:0] mdat;
wire [4:0] adir, adpc, admem; wire [2:0] opcd;     wire ldpc, pclk, aclk, mrd, mwr;

iobuffer io1( .clk2(clk2), .mrd(mrd) , .fch(fch), .alu_out(alu_out), .mdat(mdat));
accum accum1( .clk(clk1), .rst(rst), .alu_out(alu_out), .ldac(ldac) , .acc_out(acc_out));
clkgen clkgen1( .clk(clk), .rstreq(rstreq), .clk1(clk1), .clk2(clk2), .fch(fch), .rst(rst));
decode decoder1( .clk1(clk1), .clk2(clk2), .fch(fch), .rst(rst), .opcd(opcd), .ldir(ldir),
        .ldir(ldir),.ldac(ldac),.mrd(mrd),.mwr(mwr),.ldpc(ldpc),.pclk(pclk),.aclk(aclk));
instreg instreg1( .clk(clk1) , .rst(rst), .ldir(ldir), .mdat(mdat), .adir(adir), .opcd(opcd));
alu alu1( .aclk(aclk) , .mdat(mdat) , .acc_out(acc_out) , .opcd(opcd) ,
          .alu_out(alu_out) , .zr(zr) );
memory mem1(.rst(rst) , .mrd(mrd) , .mwr(mwr) , .ewr(ewr), .mad(admem),
          .ead(ead), .edat(edat), .mdat(mdat));
pc pc1( .pclk(pclk), .rst(rst), .adir(adir), .ldpc(ldpc), .adpc(adpc));
mux2_1 mux2( .adir(adir), .adpc(adpc), .fch(fch) , .admem(admem));
endmodule
```

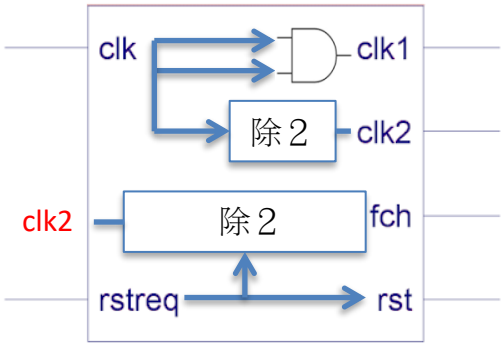
Architecture of 8-Bit RISC CPU

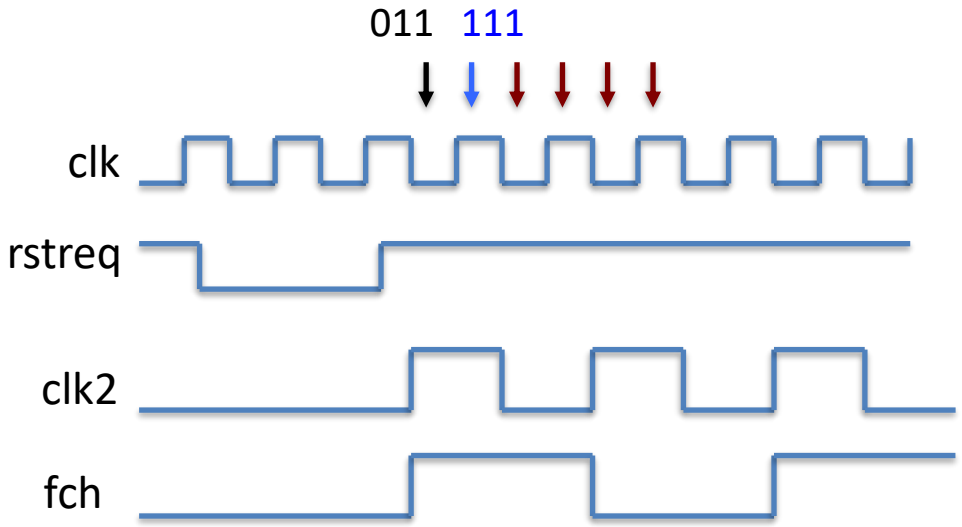


CLOCK GENERATOR Module

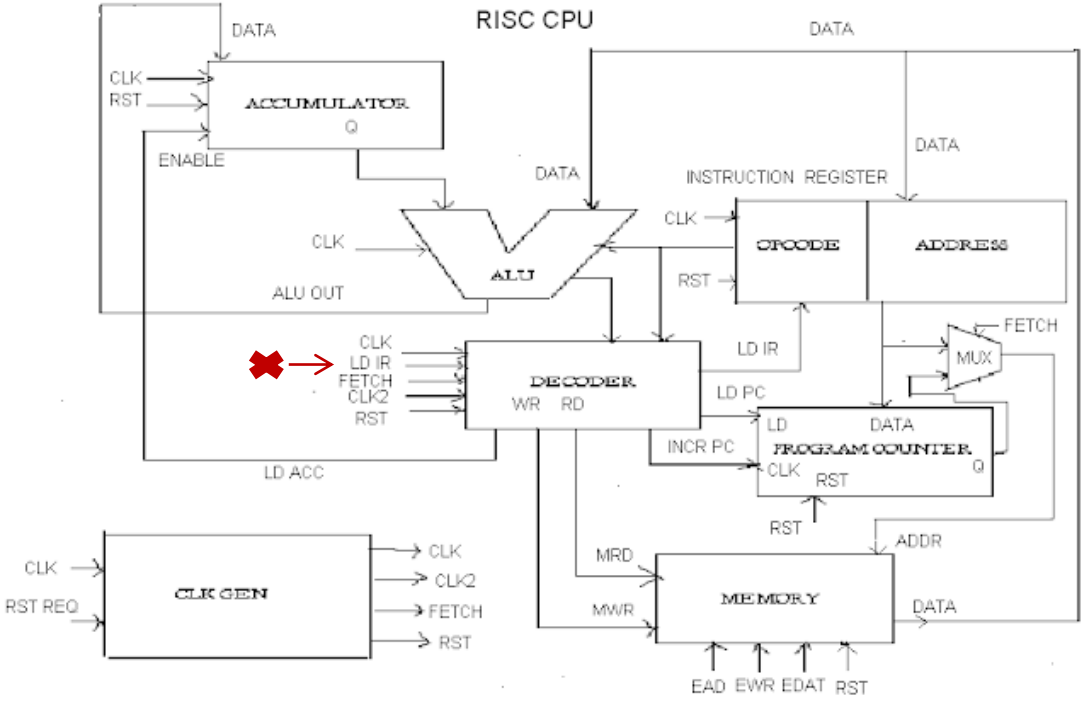
For every negative edge of clock, clock2 is generated

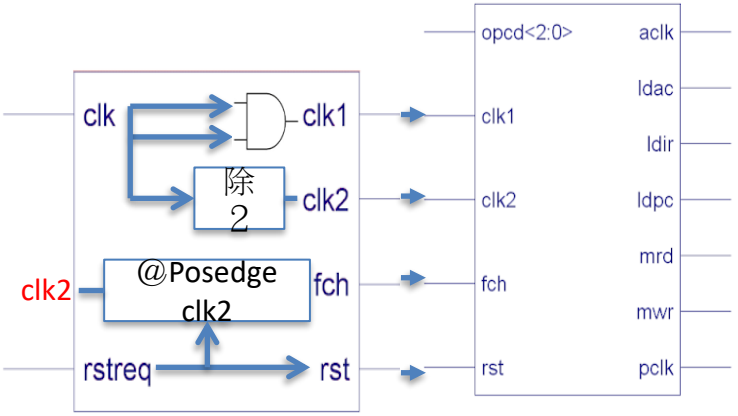
For every positive edge of clock2, fetch signal is generated.





Clock Gnerator → Decoder





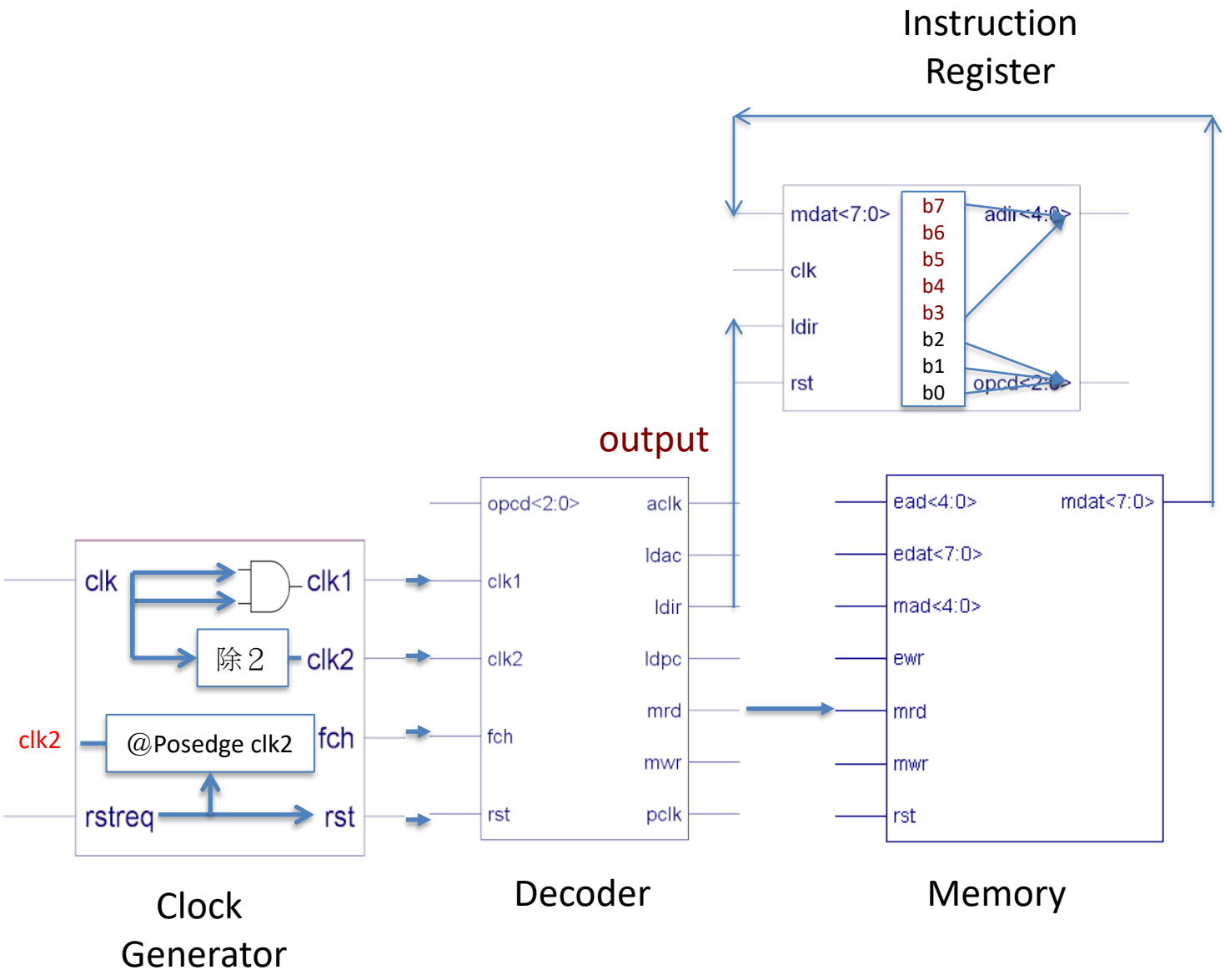
Clock Generator

Decoder

```
module clkgen(clk,rstreq,clk1,clk2,fch,rst);
```

CODE FOR CLOCK GENERATOR

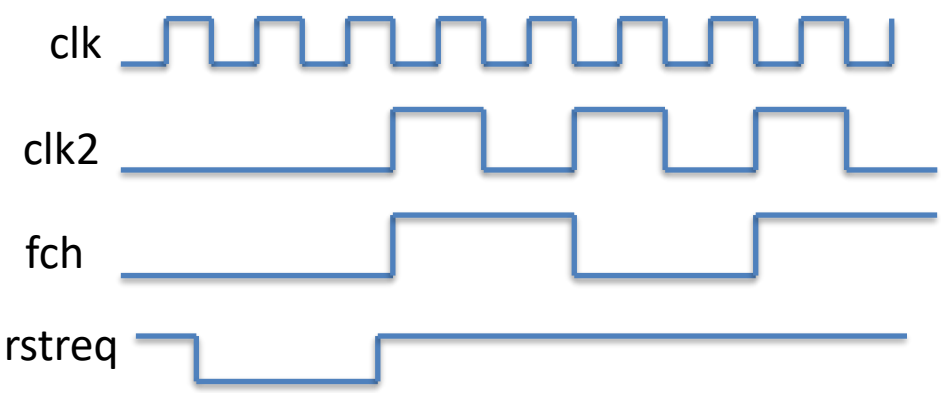

```
module clkgen(clk,rstreq,clk1,clk2,fch,rst);

input clk,rstreq;
output clk1,clk2,fch,rst;
wire c1;   reg c2,c3;              reg rst;

assign c1 = clk && clk;       // and delay for synchronous
assign clk1 = c1;
always @(negedge c1 or negedge rstreq) begin
        if (!rstreq)  begin c2 <= 0;    rst<=1; end  else begin c2 <= ~c2; clk2 <= c2 ; end

always @(posedge c2 or negedge rstreq) begin
        if (!rstreq)    c3<= 0      else begin c3<= ~ c3; fch<= c3 ; end
end
Endmodule                          //網站沒有
```


Instruction
Register
mdat<7:0>
clk
ldir
rst
b7
b6
b5
b4
b3
b2
b1
b0
adir<4:0>
opcd<2:2>
output
clk
clk1
除 2
clk2
clk2
@Posedge clk2
fch
rstreq
rst
opcd<2:0>
clk1
clk2
fch
rst
aclk
ldac
ldir
ldpc
mrd
mwr
pclk
ead<4:0>
edat<7:0>
mad<4:0>
ewr
mrd
mwr
rst
mdat<7:0>
Clock
Generator
Decoder
Memory

CPU工作劇本:

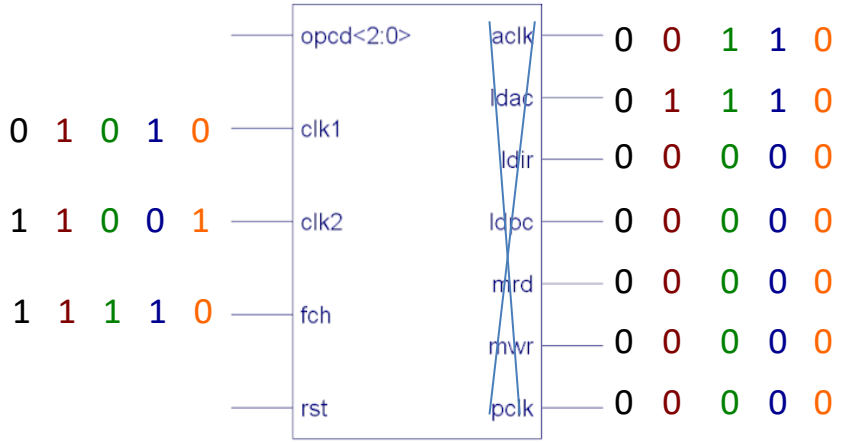
CPU2讀取 SRAM指令 | CPU2解碼 指令與執行 | CPU儲存 結果

011 111



Decoder Operation

Read data from memory




```
assign ldir = dout[0];
assign mrd  = dout[1];
assign mwr  = dout[2];
assign ldpc = dout[3];
assign ldac = dout[4];
assign pclk = dout[5];
```

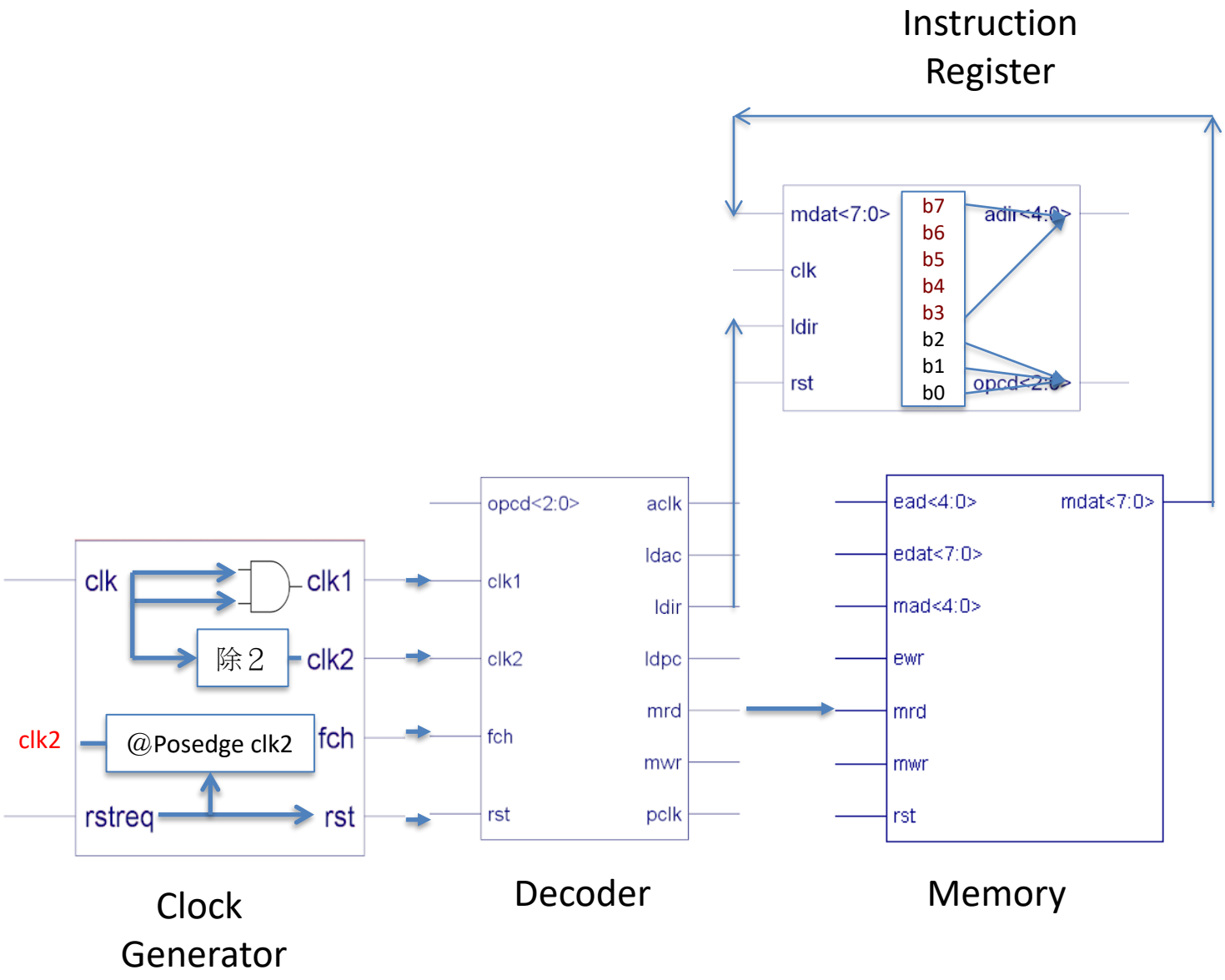
```
3'b100  : aclk = 0;
default : aclk = 1;
```

Code or Decoder

```
module decode(clk1,clk2,fch,rst,opcd,ldir,ldac,mrd,mwr,ldpc,pclk,aclk);

 input clk1, clk2, rst, fch;                    input [2:0] opcd;
output ldir, ldac, mrd, mwr, ldpc, pclk, aclk;
reg aclk;              wire [2:0] seq;         reg [7:0] dout;

assign seq = {clk1,clk2,fch};

always @(*) begin
  if(!rst)  dout = 0;   else begin
    case(seq)  3'b011 : dout = 6'b000000;
               3'b111 : dout = 6'b000010;
               3'b001 : dout = 6'b000011;
               3'b101 : dout = 6'b000011;
               3'b010 : dout = 6'b000000;
               3'b110 :  begin
                           case(opcd)  3'b010 : dout = 6'b100010;
                                       3'b011 : dout = 6'b100010;
                                       3'b100 : dout = 6'b100010;
                                       3'b101 : dout = 6'b100010;
                                       default: dout = 6'b100000;  endcase   end
```

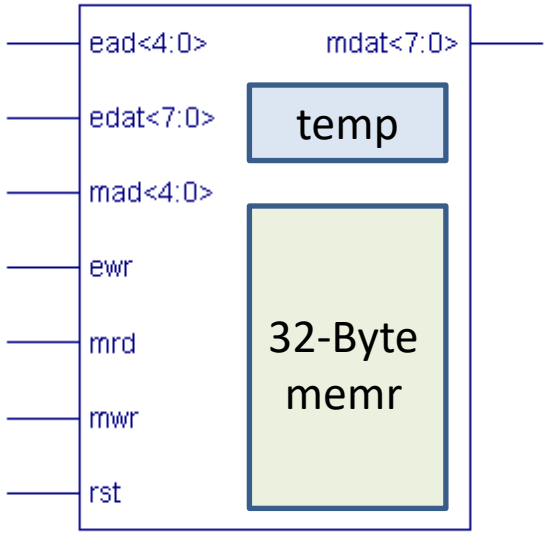
Read instruction from memory to instruction register

Decoding opcode from instruction register

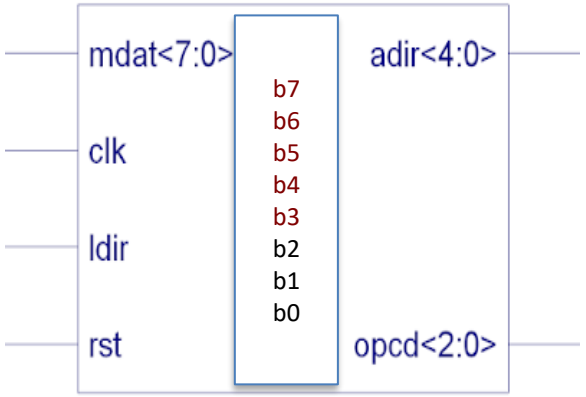

Instruction Register
mdat<7:0>
clk
ldir
rst
b7
b6
b5
b4
b3
b2
b1
b0
adir<4:0>
opcd<2:0>
clk
clk1
除 2
clk2
clk2
@Posedge clk2
fch
rstreq
rst
opcd<2:0>
clk1
clk2
fch
rst
aclk
ldac
ldir
ldpc
mrd
mwr
pclk
ead<4:0>
edat<7:0>
mad<4:0>
ewr
mrd
mwr
rst
mdat<7:0>
Clock Generator
Decoder
Memory

Memory Module

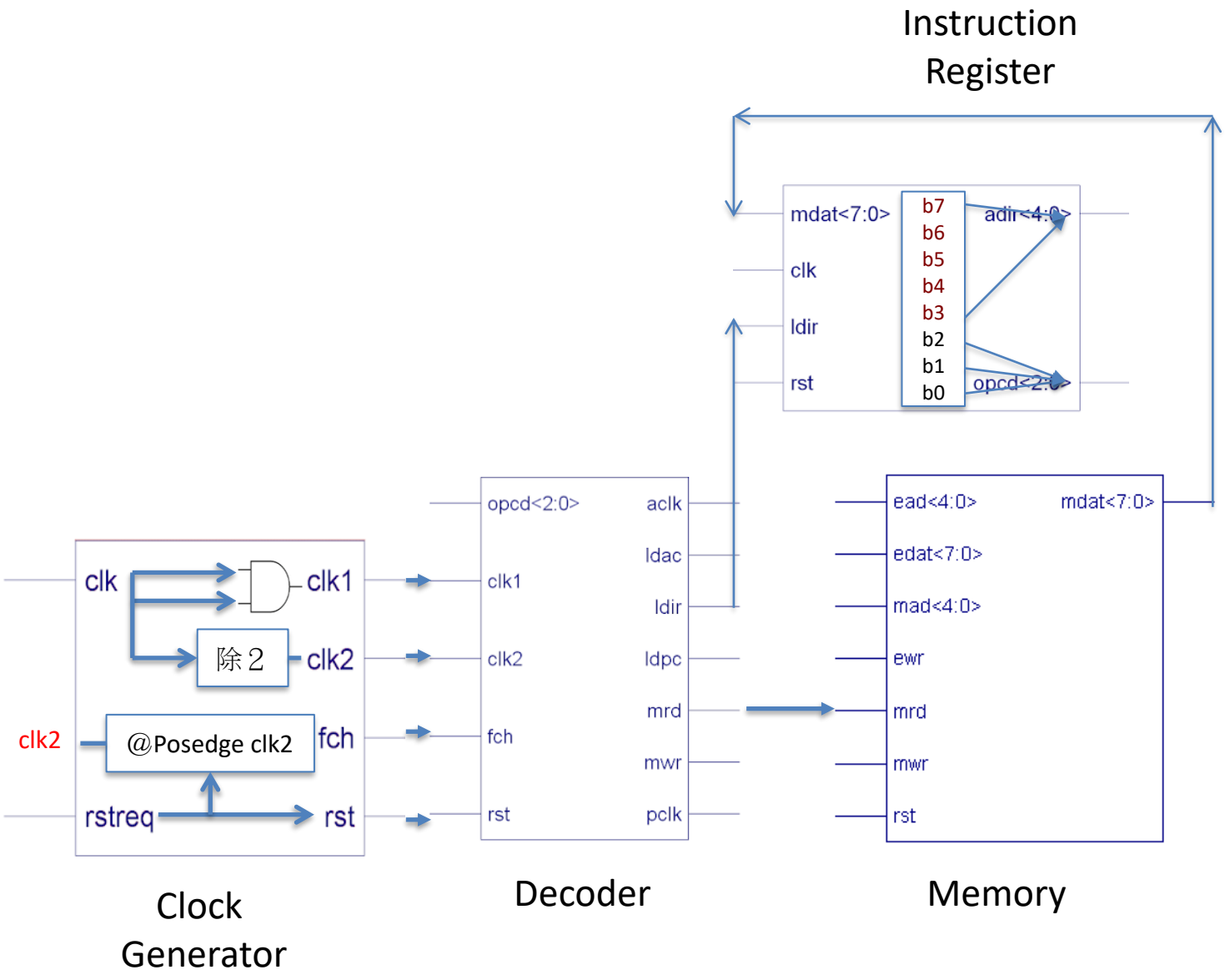
```
module  memory(rst,mrd,mwr,ewr,mad,ead,edat,mdat);

input rst, mrd, mwr, ewr;                    input [4:0] mad, ead;          input [7:0] edat,
inout [7:0] mdat;                  reg [7:0] memr [31:0] ;        reg [7:0] temp;
          // edata: external data,    mdata: CPU-memory data

always @(*) if (!rst) begin
          if (!rst) begin if(ewr == 1) begin memr[ead] <= edat; end end
          else begin if(mwr == 1) begin memr[mad] <= mdat; end
                    else if (mrd == 1) begin temp <= memr[mad]; end
                    else begin temp <= 'bzzzzzzzz; end
               end
                    end

assign mdat = temp;
endmodule
```

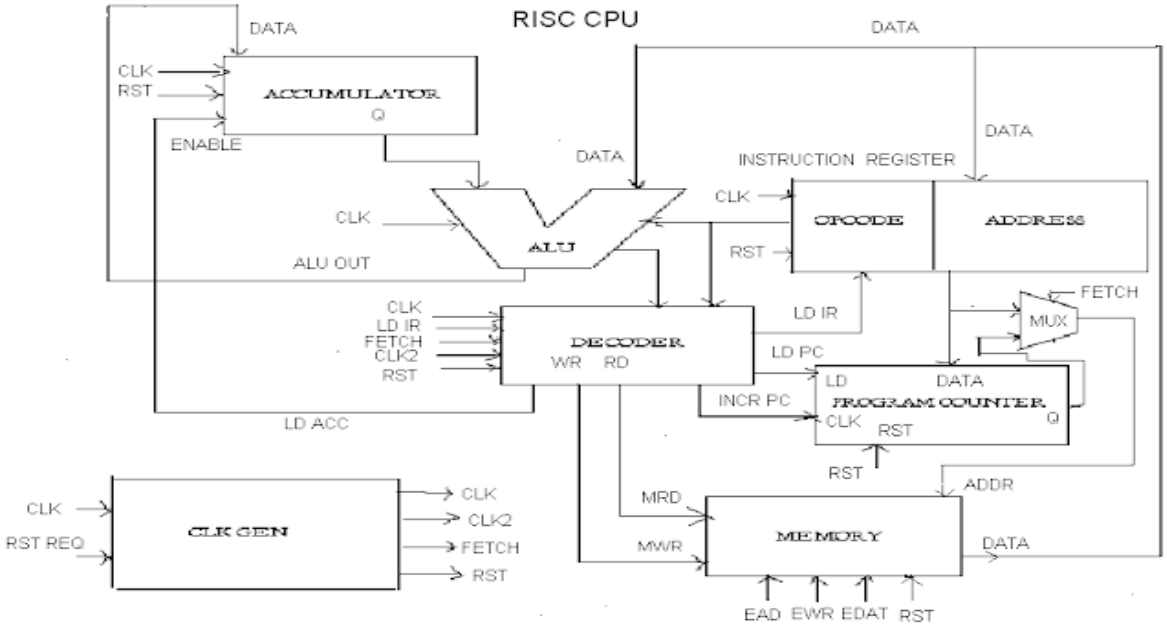


Instruction Register Module

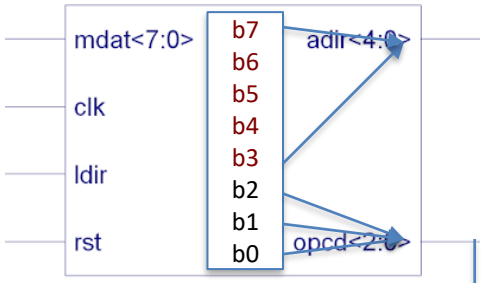


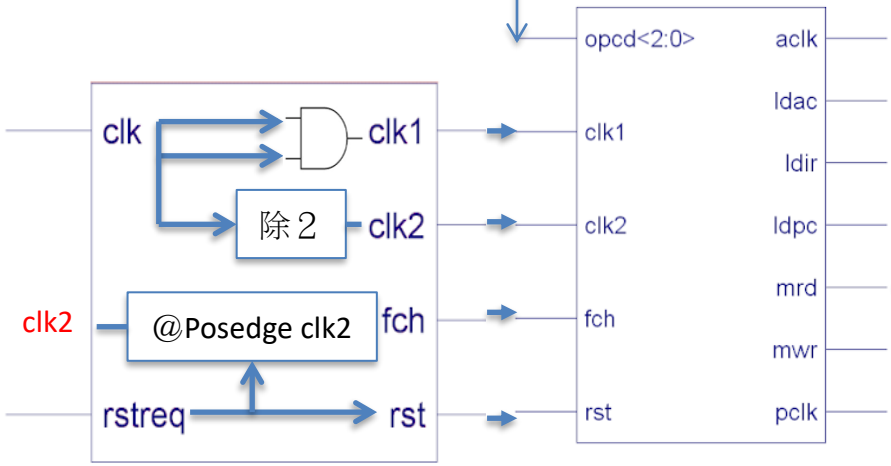

```
module  instreg(clk,rst,ldir,mdat,adir,opcd);

input clk, ldir, rst;                  input [7:0] mdat;
output [4:0] adir;                     output [2:0] opcd;
reg [4:0] adir;                        reg [2:0] opcd;

always @(posedge clk or negedge rst) begin
            if (!rst)                  begin opcd <= 0; adir <= 0; end
           else if (ldir == 1) begin  opcd <= mdat[7:3] ; adir <= mdat[2:0] ; end
      end

endmodule
```

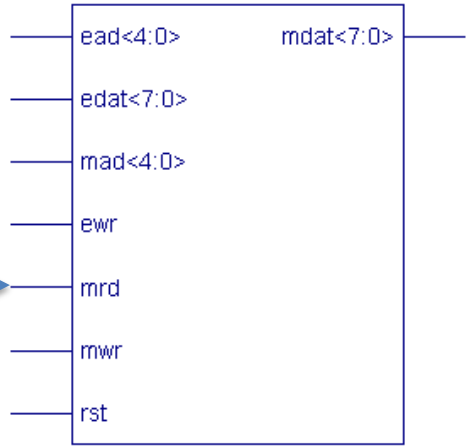

Instruction Register
mdat<7:0>
clk
ldir
rst
b7
b6
b5
b4
b3
b2
b1
b0
adir<4:0>
opcd<2:0>
clk
clk1
除 2
clk2
clk2
@Posedge clk2
fch
rstreq
rst
opcd<2:0>
clk1
clk2
fch
rst
aclk
ldac
ldir
ldpc
mrd
mwr
pclk
ead<4:0>
edat<7:0>
mad<4:0>
ewr
mrd
mwr
rst
mdat<7:0>
Clock Generator
Decoder
Memory



Instruction Register



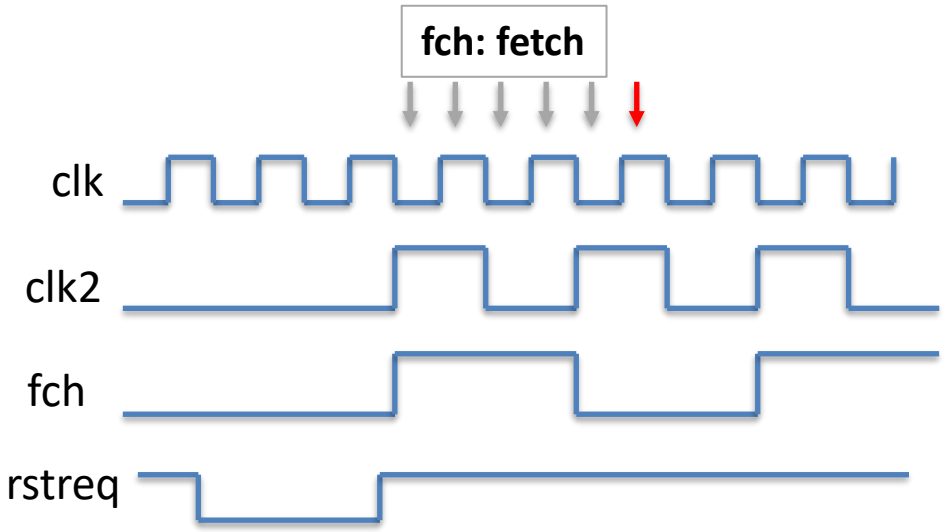




Clock Generator

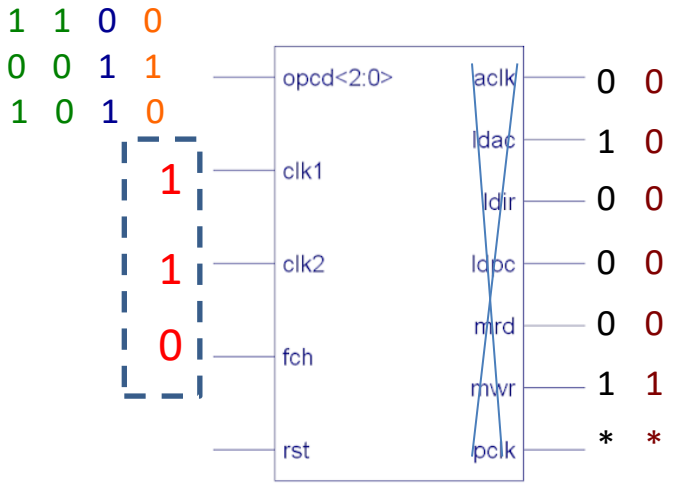
Decoder

Memory



Decoder Operation

Read data from memory




```
assign ldir = dout[0];
assign mrd  = dout[1];
assign mwr  = dout[2];
assign ldpc = dout[3];
assign ldac = dout[4];
assign pclk = dout[5];
```

```
3'b100  : aclk = 0;
default : aclk = 1;
```

default: dout = **6'b100000**; **3'b010** : dout = **6'b100010**;

Cpde for Decoder

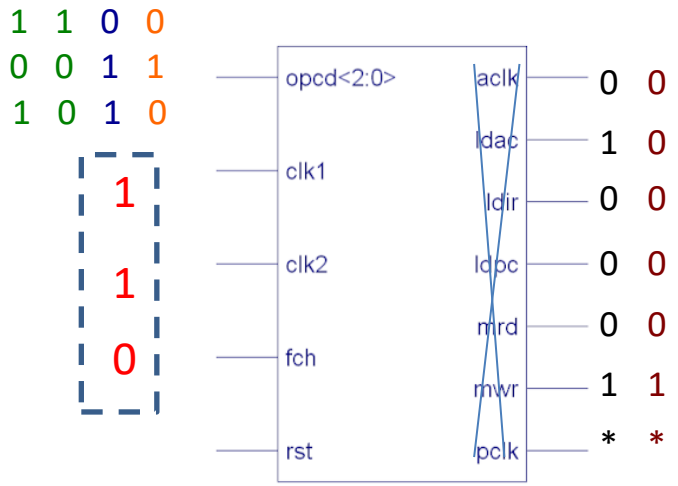
```
module decode(clk1,clk2,fch,rst,opcd,ldir,ldac,mrd,mwr,ldpc,pclk,aclk);

 input clk1, clk2, rst, fch;                    input [2:0] opcd;
output ldir, ldac, mrd, mwr, ldpc, pclk, aclk;
reg aclk;              wire [2:0] seq;         reg [7:0] dout;

assign seq = {clk1,clk2,fch};

always @(*) begin
  if(!rst)  dout = 0;   else begin
    case(seq)  3'b011 : dout = 6'b000000;
               3'b111 : dout = 6'b000010;
               3'b001 : dout = 6'b000011;
               3'b101 : dout = 6'b000011;
               3'b010 : dout = 6'b000000;
               3'b110 :  begin
                           case(opcd)  3'b010 : dout = 6'b100010;
                                       3'b011 : dout = 6'b100010;
                                       3'b100 : dout = 6'b100010;
                                       3'b101 : dout = 6'b100010;
                                       default: dout = 6'b100000;  endcase   end
```

Read instruction from memory to instruction register

Decoding opcode from instruction register

OPCODE for Decoder

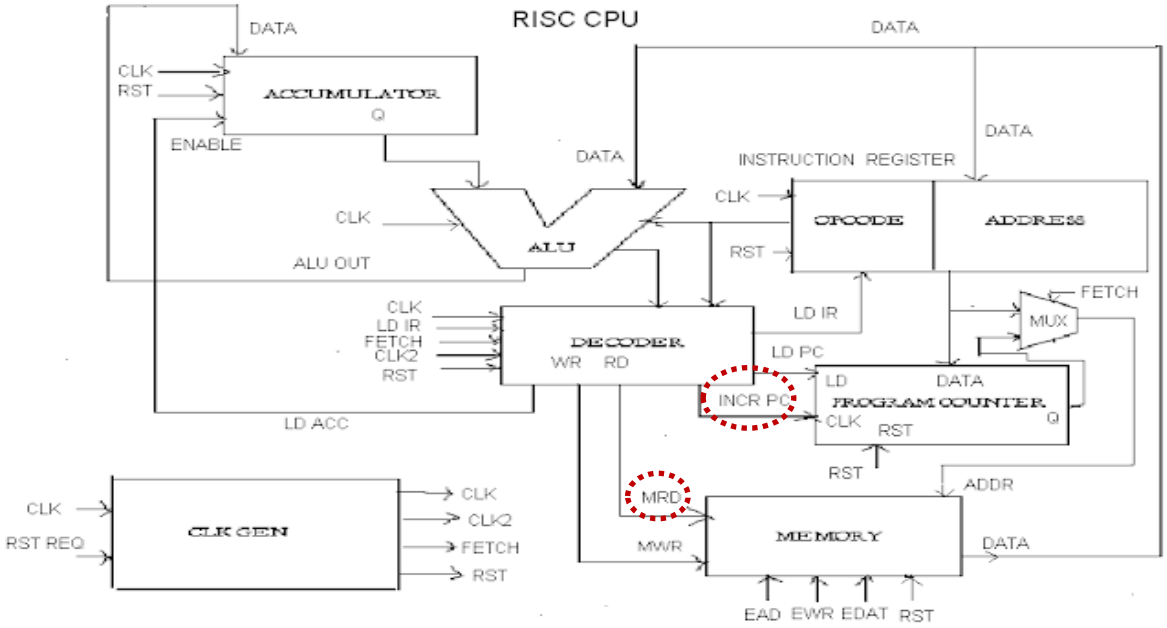
Instruction	Opcode	
Halt	000	//控制所有module電路皆Halt
Skip of zero	001	//跳到0結果
Add	010	// Adder operation by controlling ALU
And	011	// And operation by controlling ALU
Xor	100	// Xor operation by controlling ALU
Load	101	// Load data by controlling Bus
Store	110	// Store data by controlling Bus
Jump	111	// Jump to 副程式 by changing Program Counter

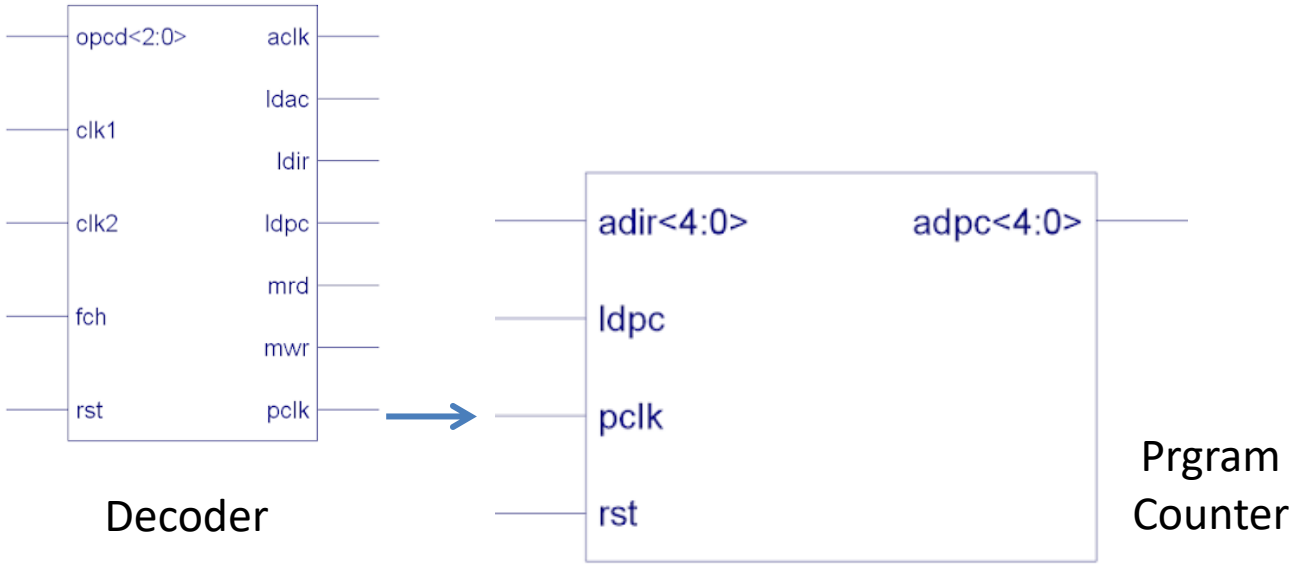



```
assign ldir = dout[0];
assign mrd  = dout[1];
assign mwr  = dout[2];
assign ldpc = dout[3];
assign ldac = dout[4];
assign pclk = dout[5];
            //program counter之clock
```

default: dout = **6'b100000**;

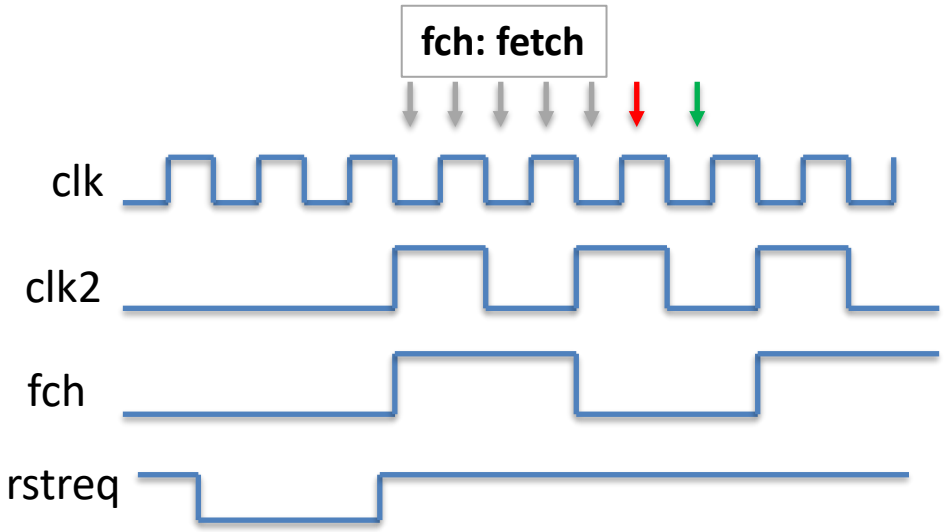
3'b010 : dout = **6'b100010**;


RISC CPU
DATA
CLK
RST
ACCUMULATOR
Q
ENABLE
DATA
DATA
INSTRUCTION REGISTER
DATA
CLK
ALU
ALU OUT
CLK
RST
OPCODE
ADDRESS
FETCH
MUX
CLK
LD IR
FETCH
CLK2
RST
DECODER
WR RD
LD IR
LD PC
LD
DATA
INCR PC
PROGRAM COUNTER
CLK
RST
Q
LD ACC
RST
ADDR
MRD
MWR
MEMORY
DATA
CLK
RST REQ
CLK GEN
CLK
CLK2
FETCH
RST
EAD EWR EDAT RST


opcd<2:0>
clk1
clk2
fch
rst
aclk
ldac
ldir
ldpc
mrd
mwr
pclk
Decoder
adir<4:0>
ldpc
pclk
rst
adpc<4:0>
Prgram Counter

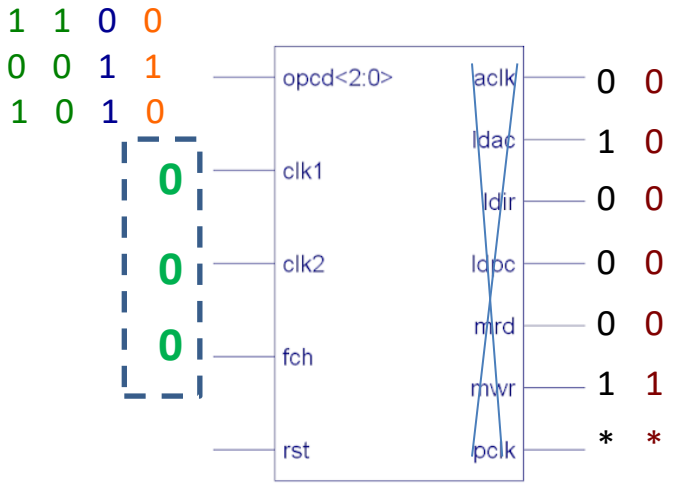
Cpde for Program Counter


```
module pc( pclk, rst, adir, ldpc, adpc);

input pclk, rst, ldpc;          input [4:0] adir;
output [4:0] adpc;
reg [4:0] temp1;

      always @ (posedge pclk or negedge rst) begin
            if (!rst) begin  temp1 <= 0; end
            else if ( ldpc == 1) begin  temp1 <= adir; end
            else begin temp1 <= temp1 + 1; end
      end

      assign adpc = temp1;

endmodule
```



Decoder Operation

Read data from memory




```
assign ldir = dout[0];
assign mrd  = dout[1];
assign mwr  = dout[2];
assign ldpc = dout[3];
assign ldac = dout[4];
assign pclk = dout[5];
```

```
3'b100  : aclk = 0;
default : aclk = 1;
```

default: dout = **6'b100000**; **3'b010** : dout = **6'b100010**;

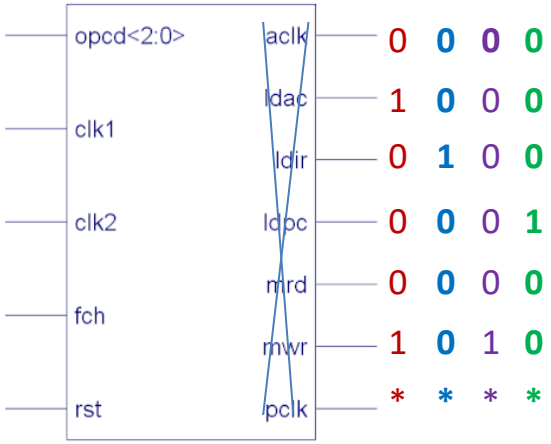
Decoder module 2

```
3'b000 : begin case(opcd)
                    3'b010 :        dout = 6'b010010;
                     3'b011 : begin dout = 6'b010010; end
                     3'b100 : begin dout = 6'b010010; end
                     3'b101 : begin dout = 6'b010010; end
                     3'b110 : begin dout = 6'b000100; end
                     3'b001 : begin dout = 6'b100000; end
                     3'b111 : begin dout = 6'b001000; end
                     default : begin dout = 6'b000000; end endcase
```

OPCODE for Decoder

Instruction	Opcode	
Halt	000	//控制所有module電路皆Halt
Skip of zero	001	//跳到0結果
Add	010	// Adder operation by controlling ALU
And	011	// And operation by controlling ALU
Xor	100	// Xor operation by controlling ALU
Load	101	// Load data by controlling Bus
Store	110	// Store data by controlling Bus
Jump	111	// Jump to 副程式 by changing Program Counter

1 0 1 1 1 0 0
1 0 1 0 0 1 1
1 1 0 1 0 1 0

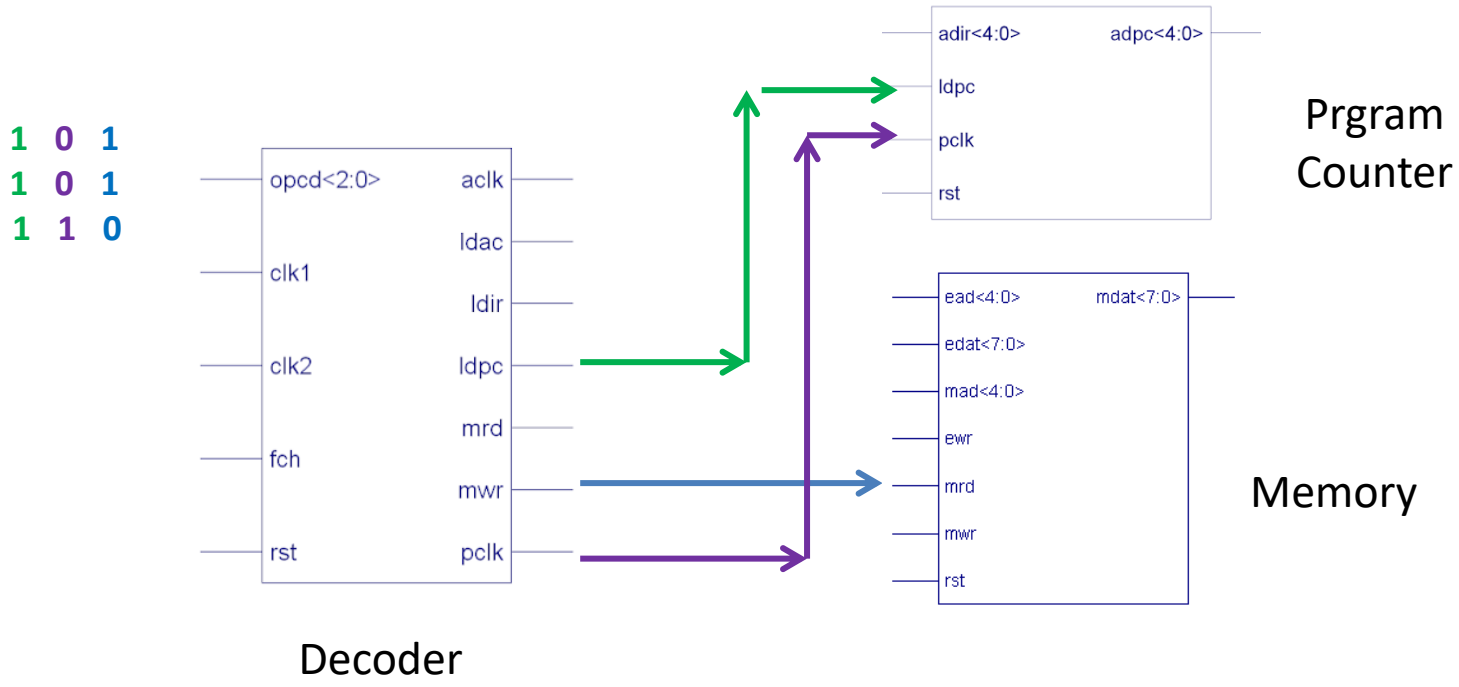



```
assign  ldir = dout[0];
assign  mrd  = dout[1];
assign  mwr  = dout[2];
Assign  ldpc = dout[3];
assign  ldac = dout[4];
assign  pclk = dout[5];

//program counter之clock
```

default: dout = **6'b100000**;

3'b010 : dout = **6'b100010**;

Decoder Output



Decoder module 3

```
3'b100 : begin case(opcd)
                        3'b010 :        dout = 6'b010010;
                         3'b011 : begin dout = 6'b010010; end
                         3'b100 : begin dout = 6'b010010; end
                         3'b101 : begin dout = 6'b010010; end
                         3'b001 : begin dout = 6'b100000; end
                         3'b111 : begin dout = 6'b101000; end
                         3'b110 : begin dout = 6'b000100; end
                         default : begin dout = 6'b000000; end endcase
endcase end end

assign ldir = dout[0];          assign mrd = dout[1];
assign mwr = dout[2];           assign ldpc = dout[3];
assign ldac = dout[4];          assign pclk = dout[5];
```

Decoder module 4

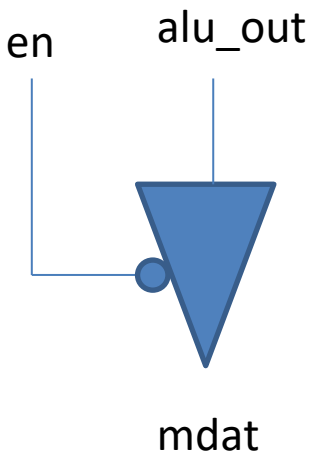
```
always @(*) begin
                    if(!rst) begin aclk = 0; end
                    else begin
                                case(seq) 3'b100 : aclk = 0;
                                            default : aclk = 1; endcase
                end
end

endmodul
e
```

ALU (Arithmatic Logic Unit) module 4

```
module alu (aclk,mdat,acc_out, opcd, alu_out, zr);

input aclk;          input [7:0] mdat, acc_out;          input [2:0] opcd;
output [7:0] alu_out;                                    output zr;
reg [7:0] a,alu_out;                                     wire zr;

always @ ( posedge aclk ) begin
                    case (opcd)   3'b000 : begin  a <= acc_out; end
                                  3'b001 : begin a <= acc_out; end
                                  3'b010 : begin a <= {mdat + acc_out}; end
                                  3'b011 : begin a <= {mdat & acc_out}; end
                                  3'b100 : begin a <= {mdat ^ acc_out}; end
                                  3'b101 : begin a <= mdat; end
                                  3'b110 : begin a <= acc_out; end
                                  3'b111 : begin a <= acc_out; end
                                  default : begin a <= 0; end
                endcase end
assign alu_out = a;
assign zr = &(a);
endmodul
e
```

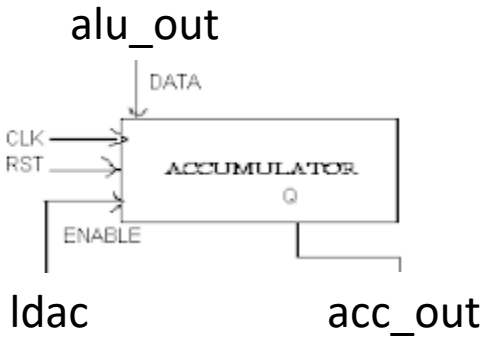
CODE FOR INPUT-OUTPUT BUFFER

```
module iobuffer(clk2,mrd,fch,alu_out,mdat);

input clk2,mrd,fch;
input [7:0] alu_out;
output [7:0] mdat;
wire en, en1;

assign en = mrd || fch || clk2;
assign en1 = ~en;
assign mdat = (en1) ? alu_out : 'hzz ;

endmodule
```



CODE FOR ACCUMULATOR

```
module accum(clk, rst, alu_out, ldac, acc_out);

input clk, rst, ldac;
input [7:0] alu_out;
output [7:0] acc_out;          reg [7:0] acc_out;

always @ (posedge clk or negedge rst) begin
        if (!rst) begin
                acc_out <= 0;
        end else if ( ldac == 1) begin
                acc_out <= alu_out[7:0];
        end
end
endmodule
```



CODE FOR MULTIPLEXER

```
module mux2_1(adir, adpc, fch, admem);

input [4:0] adir, adpc;          input fch;
output [4:0] admem;              reg [4:0] admem;

always @ (fch,adpc,adir) begin
        case (fch)
                1'b1 : admem = adpc;
                1'b0 : admem = adir;
        endcase  end
endmodule
```

TESTBENCH CODE

```
module testbench();

 reg clk, rstreq, ewr;          reg [4:0] ead;        reg [7:0] edat;
wire zr;                int i;

topmodule dut11(clk,rstreq,ewr,ead,edat,zr);
 initial begin
            rstreq = 1;
            #12; @(negedge clk); rstreq = 0;
            #12; ewr = 1'd1;
            #12; ead = 5'd1;
//int i ; //repeat(31) begin


endmodule
```