N26F300
VLSI SYSTEM DESIGN
(GRADUATE LEVEL)

[Material partly adapted from lecture notes of Prof. KJ Lee]

Fall 2010

Design of SISC Pipeline CPU

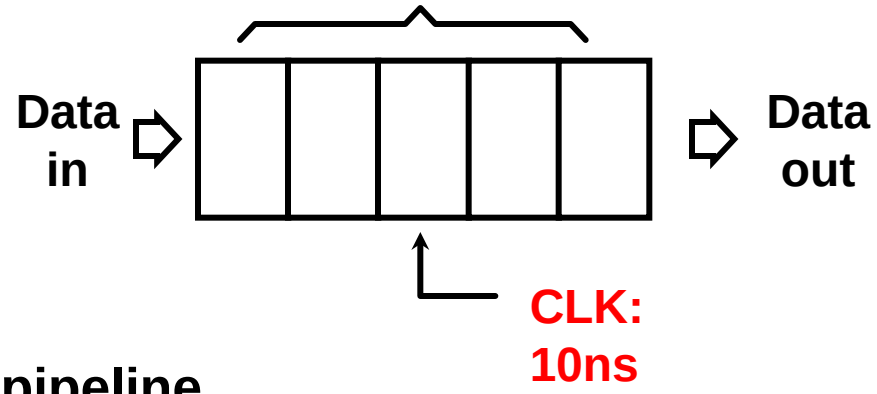
Outline

- Concept of pipeline
- Simultaneous events
- Fetch unit
- Execution unit
- Write unit
- Data dependency
- A complete list of Verilog code

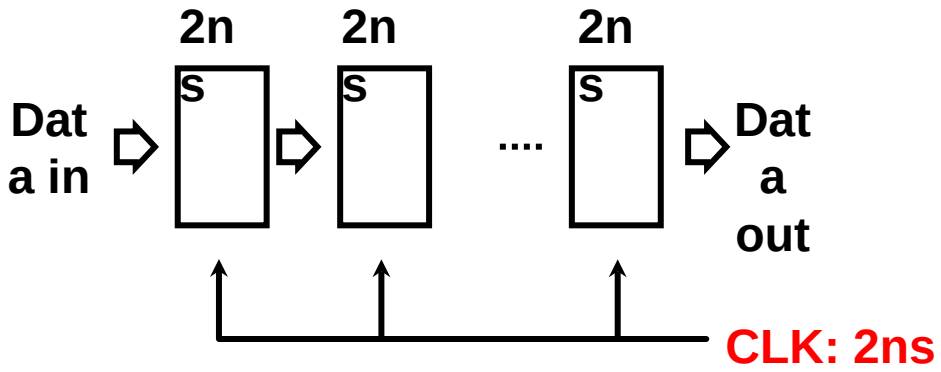
Ref : Sternheim, "Digital design and synthesis with Verilog HDL"

Concept of Pipeline

Without pipeline



With pipeline



Pipeline CPU

Cycle #	Without Pipeline	With Pipeline		
1	F1	F1		
2	E1	F2	E1	
3	W1	F3	E2	W1
4	F2	F4	E3	W2
5	E2	F5	E4	W3
6	W2	.	E5	W4
7	F3	.	.	W5
8	E3	.	.	.
9	W3	.	.	.
10	F4	.	.	.
11	E4	.	.	.
12	W4	F12	.	.
13	F5	F13	E12	W11
14	E5	F14	E13	W12
15	W5	F15	F14	W13

Maximum throughput of an n-staged pipeline

$$= \frac{1}{\text{maximum of delays of stages}}$$

F=Fetch, E=Execute, W=Write

Without pipeline: 5 instructions executed in 15 cycles. (#instructions)*3

With pipeline: 5 instructions executed in 7 cycles. (#instructions)+2

Factors affecting pipeline efficiency

1 Branch instructions

⇒ The prefetched instruction must be discarded.

2 Exception (Reset, interruption, etc.)

3 Load & Store instructions

⇒ These instructions are necessary for accessing memory to transfer data to or from the register file inside the processor. (Multi-ported memories and register files to keep the pipeline full)

4 Data dependent instructions

⇒ An instruction, say *I*, requires the results from the instruction(s) right before *I*.

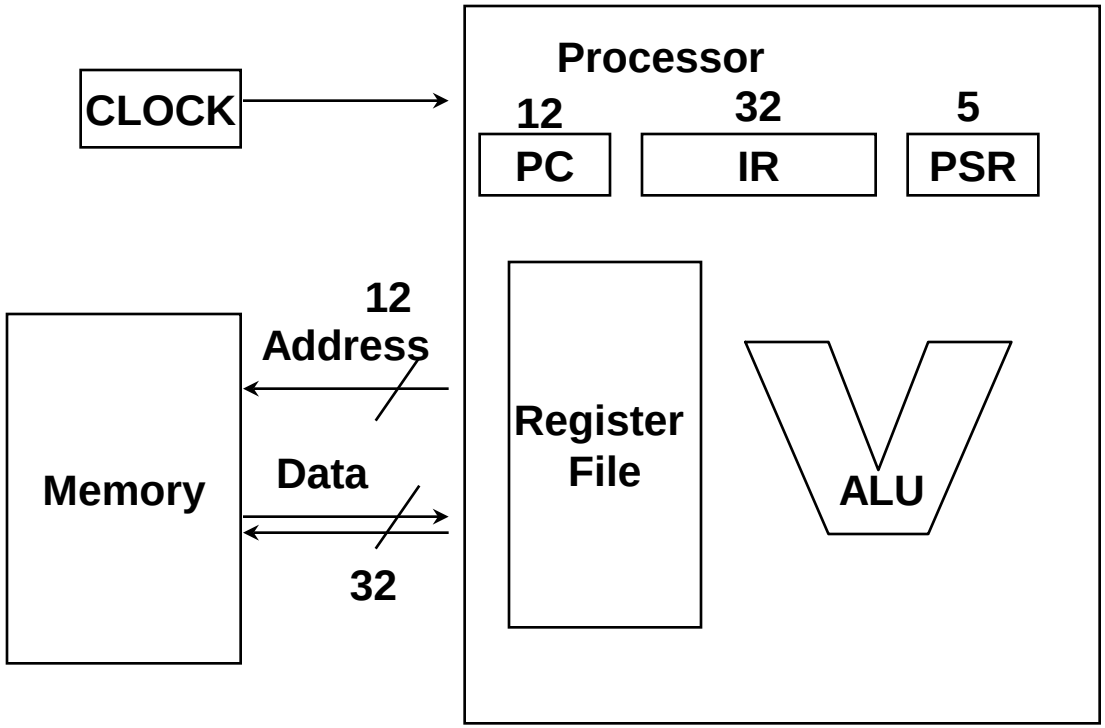
Flushing a pipeline : to ensure the proper completion of all instructions.

A small instruction set computer (SISC)

- 10 instructions
- 32-bit instruction register
- 12-bit address register
- 5-bit processor status register (PSR)
- 33-bit computation results

Instruction set model

An instruction set model of a processor describes the effect of executing the instructions and the interactions among them.



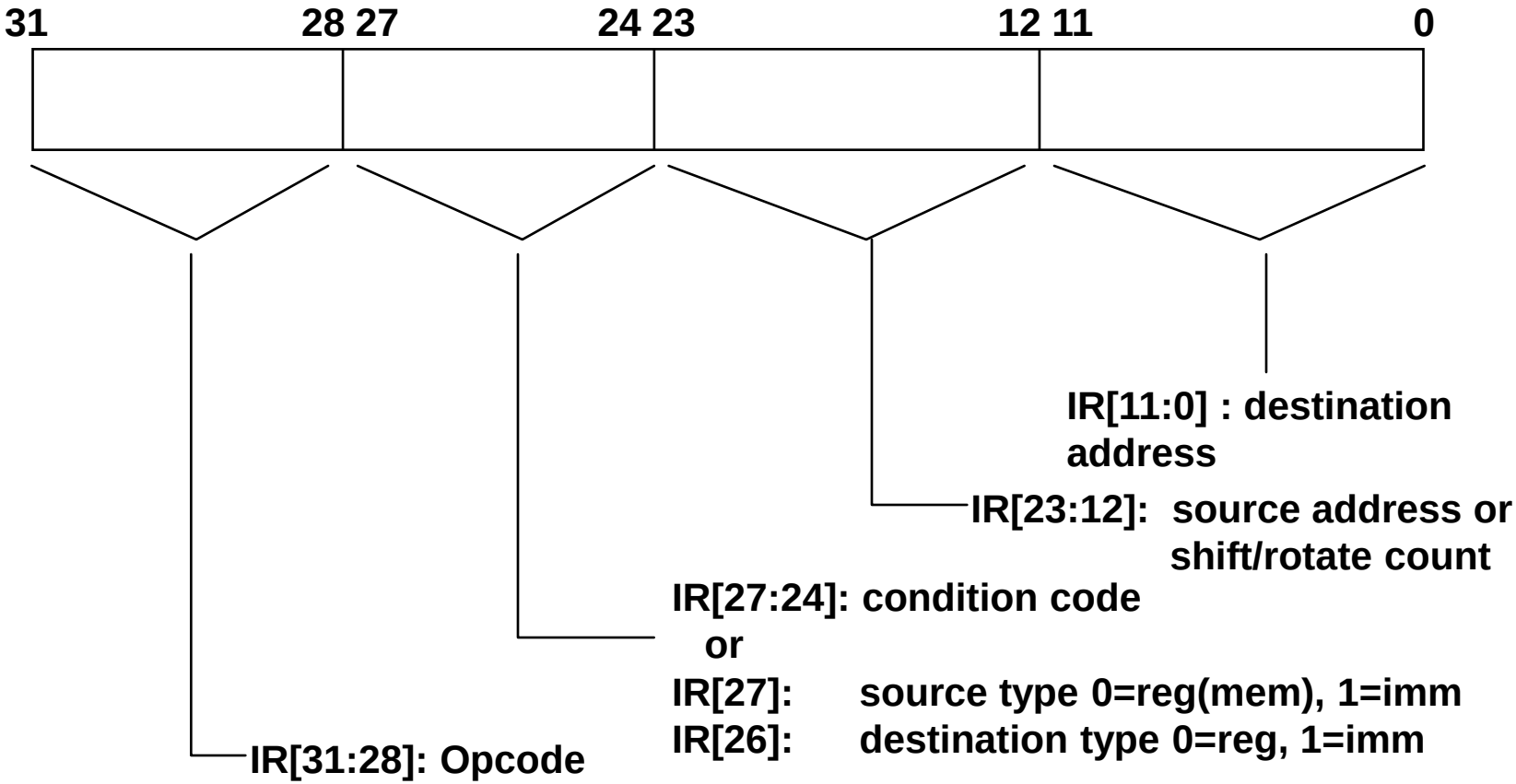
Instructions

Name	Mnemonic	Opcode	Format(inst dst, src)
NOP	NOP	0	NOP
BRANCH	BRA	1	BRA mem, cc
LOAD	LD	2	LD reg, mem1
STORE	STR	3	STR mem, src
ADD	ADD	4	ADD reg, src
MULTIPLY	MUL	5	MUL reg, src
COMPLEMENT	CMP	6	CMP reg, src
SHIFT	SHF	7	SHF reg, cnt
ROTATE	ROT	8	ROT reg, cnt
HALT	HLT	9	HLT

Operand addressing


```
mem    Memory address
mem1   Memory address or immediate value
reg    Any register index
src    Any register index,
        or immediate value
cc     Condition code
cnt    Shift/rotate count,
       >0: right shift,
       <0: left shift,
       +/-32
```

Instruction Format

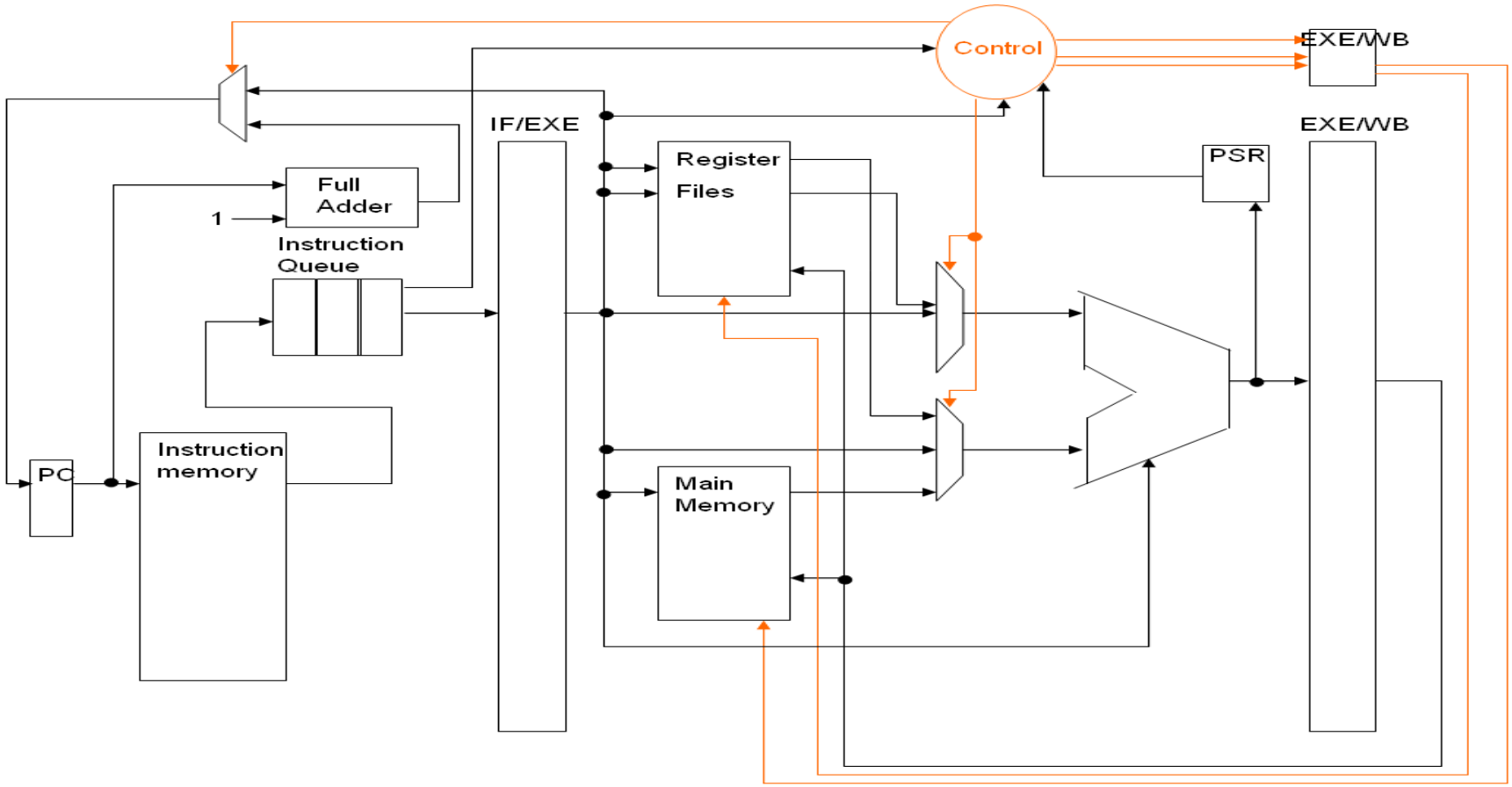


Condition codes

A	Always	0
C	Carry	1
E	Even	2
P	Parity	3
Z	Zero	4
N	Negative	5

Processor status register	
PSR [0]	Carry
PSR [1]	Even
PSR [2]	Parity
PSR [3]	Zero
PSR [4]	Negative

Block Diagram of SISC



Two-phase clock

1. phase1_loop : triggered on the positive edge of clock

- **Execute 3 simultaneous events:**

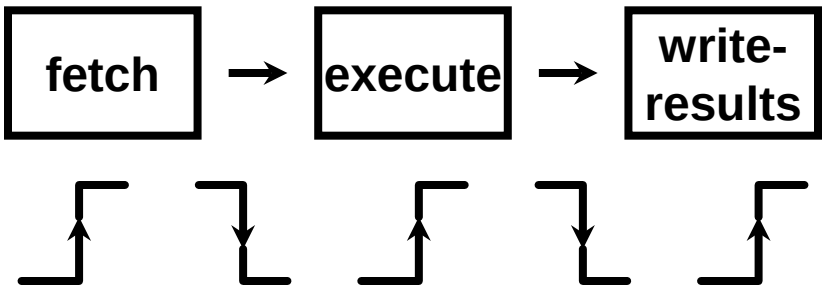
  do_fetch ,

  do_execute,

  do_write_results

2. phase2_loop : triggered on the negative edge of clock

- **Transfer information between pipeline stages and update some data registers and conditional codes.**



Additional declarations for pipeline modeling


```
parameter QDEPTH=3; // Instr Queue Depth
// Instr queue, and instr register for write
reg    [WIDTH-1:0]   IR_Queue [0:QDEPTH-1], wir ;

// Copy of result, and Execute and fetch pointers
reg    [WIDTH:0]   wresult;
reg    [2:0]          eptr, fptr, qsize;

// Various Controls/Flags
reg    mem_access, branch_taken, halt_found ;
reg    result_ready ;
reg    executed, fetched ;
wire  queue_full ;

event do_fetch, do_execute, do_write_results ;
```

Event <event_name>

- Declare an abstract event that can be used to trigger execution of a statement or a block of statements
- Used in high-level modeling
- Not supported by synthesis tools

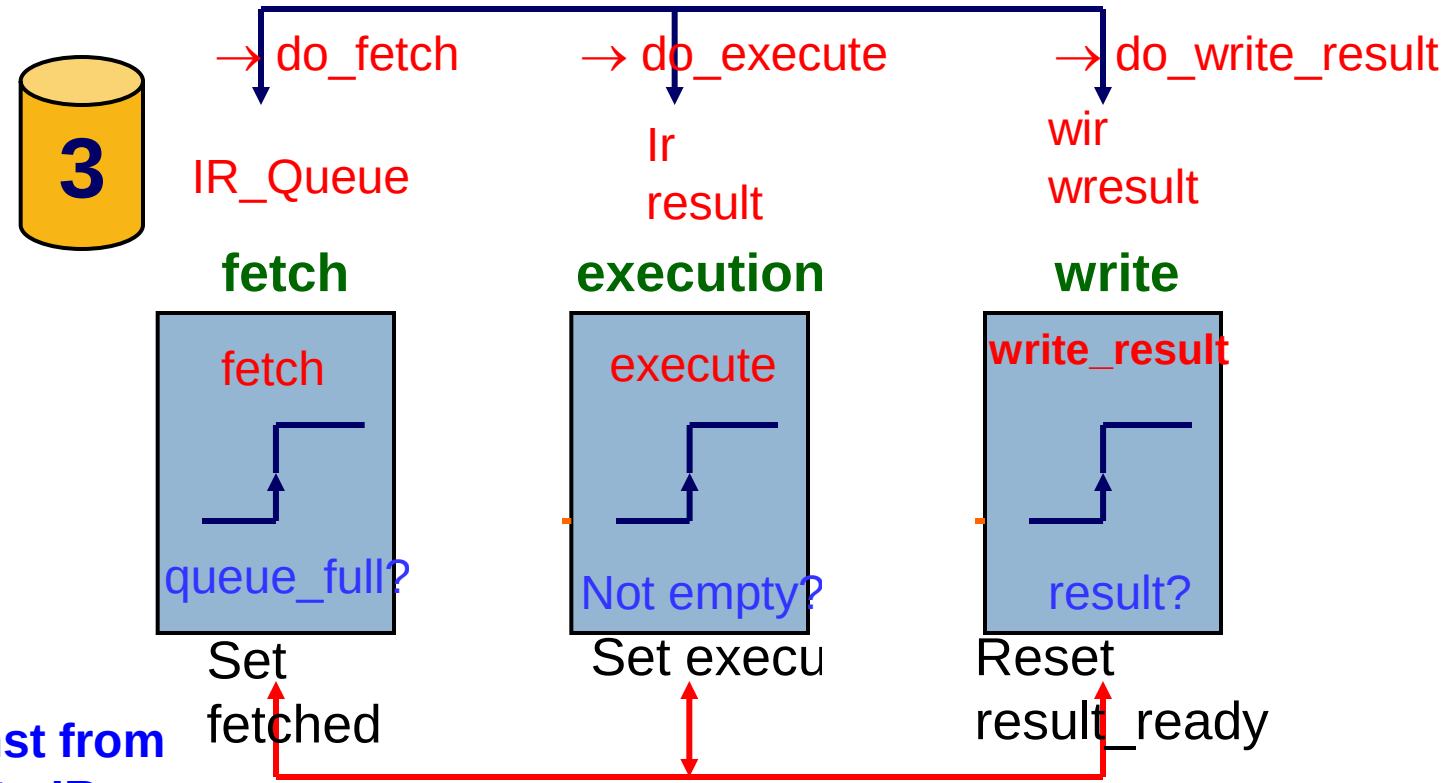
• An event is triggered by the symbol "->"

• The triggering of the event is recognized by the symbol @

```
event do_fetch, do_execute, do_write_results ;
.....
        if  (!queue_full && !mem_access)
                -> do_fetch;
.....
        always @ (do_fetch):  fetch_block
                fetch;
```

Execution of pipeline structure

concurrent processes ⇦ positive edge clock (phase 1)



Triggering simultaneous events

Phase1_loop (at positive edge of clock): to trigger three simultaneous events (The order of if statements is not important.)

```
always @(posedge clock) begin: phase1_loop
  if  (!reset)  begin
     fetched = 0;
     executed=0;
     if  (!queue_full && !mem_access)
         -> do_fetch;
     if  (qsize || mem_access)
         -> do_execute;
     if (result_ready)
         -> do_write_result;
  end
end
```

The mem_access flag signals the fetch unit to stall a cycle because a load or a store instruction is in progress.

The fetch unit

Fetch : memory → instruction queue

Queue depth = 3 = # pipeline stages

```
IR_Queue  [0 : QDEPTH-1];
reg   [2:0]   eptr,    // end
              fptr,     // first (current position to store next fetched instruction)
              qsize;  // size of queue entries with instructions
```

```
task  fetch;
begin
   IR_Queue [fptr] =  MEM [pc];
   fetched = 1;
end
endtask;
```

qsize, eptr, and fptr are updated in task set_pointers.

To invoke fetch:

always @ (do_fetch): fetch_block
fetch;

The execution unit

- Decodes the current instruction and executes it.
- Assume that all instructions except load and store can be executed in one cycle.
- All arithmetic instructions can be executed in one cycle, i.e., no memory access for ADD, MUL, SHF, etc. The operands are from register files or are immediate.

Load Instruction

Load instruction: need two cycles (Store similar)

- **Cycle 1: reserve memory by setting mem_access flag**
- **Cycle 2: access memory.**

Instruction Access:

```
if  (qsize && mem_access == 0) ir = IR_Queue [eptr];
```

LOAD Instruction:

```
`LD : begin
      if (mem_access == 0)
          mem_access == 1; // Reserve next cycle
      else begin
        ..... // Memory access in next cycle
      end
```

In 1st cycle, the fetch unit still reads the next instruction from memory.

In 2nd cycle, the fetch unit is idle while the execution unit will access memory.

Branch


```
`BRA :
    begin
        if (debug )  $write ("Branch..");
        if (checkcond (`CCODE) == 1)
        begin
            pc = `DST;
            branch_taken = 1;
        end
    end
```

When a branch is taken or the halt instruction is executed, the instruction queue should be "flushed".

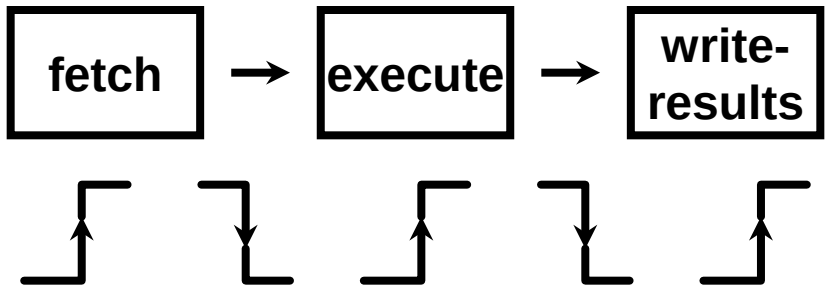
Flushing the pipeline


```
task flush_queue;
begin
    // pc is already modified by branch execution
    fptr = 0;
    eptr = 0;
    qsize = 0;
    branch_taken = 0;
end
endtask
```

Write Unit

Execute Unit and Write Unit communicate via two registers "wresult" and "wir" and a "result-ready" flag. This is done in phase2_loop, i.e., at the negative edge of clock.

```
task copy_results;
begin
    if ((`OPCODE >= `ADD) && (`OPCODE < `HLT)) begin
        setcondcode (result);
        wresult = result;
        wir = ir;
        result_ready = 1;
    end
end
endtask
```



Copying result and instruction

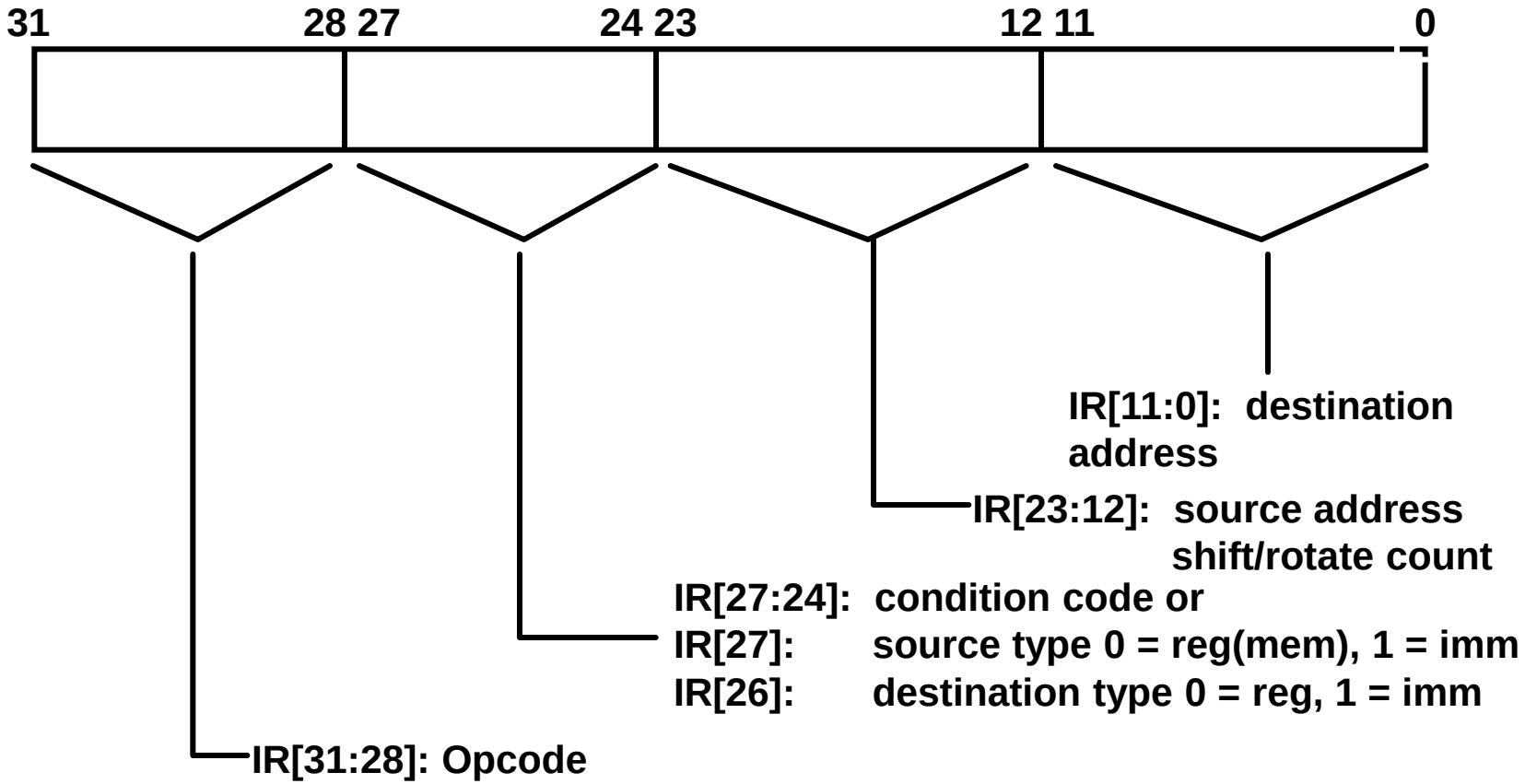
Simplify the write unit by copying the current instruction instead of keeping a pointer to the instruction in the instruction queue.

Write Unit (cont.)

Then the write unit writes the result to the destination register in the next cycle.

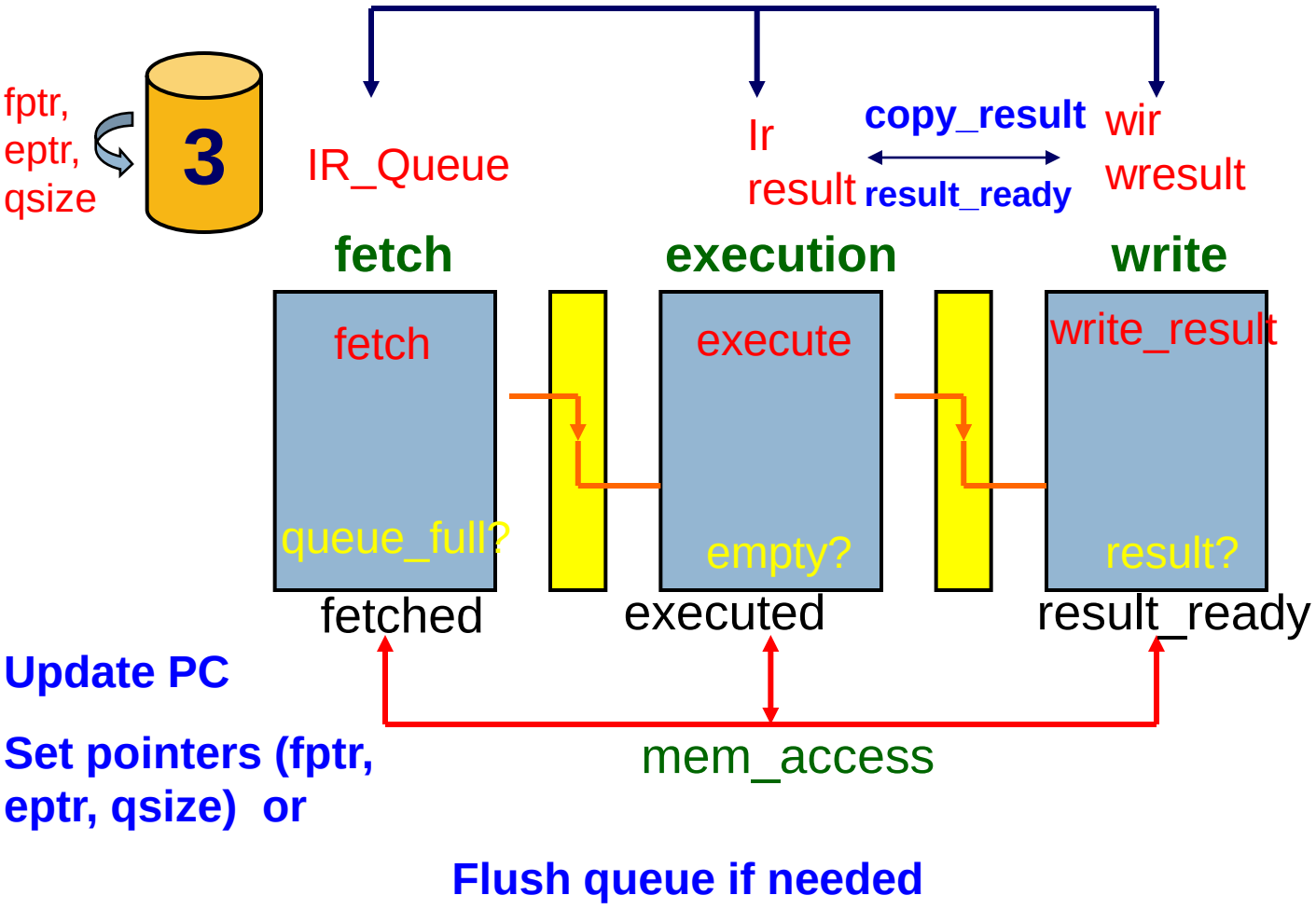
```
task write_result ;
begin
  if ((`WOPCODE >= `ADD) && (`WOPCODE  < `HLT)) begin
      if (`WDSTTYPE == `REGTYPE)
          RFILE [`WDST] = wresult;
      else $display ("Error: destination error.");
      result_ready = 0;
   end
end
endtask
```

Instruction Format



Execution of Phase 2

negative edge clock (phase 2)



Phase-2 Control Operation

1. Update PC based on whether the branch was taken or not.
2. Set conditional codes from the newly computed result.
3. Copy ir and result to wir and wresult.
4. Update eptr, fptr and qsize.

Phase-2 Control operations

```
task set_pointers;   // Manage queue pointers
begin
   case ({fetched, executed})
     2'b00 : ;              // idle fetch cycle
     2'b01 : begin          // No fetch
                qsize = qsize - 1;
                eptr = (eptr+1) % QDEPTH;
             end
    2'b10 : begin           // No execute
                qsize = qsize + 1;
                fptr = (fptr+1) % QDEPTH;
             end
```

Cont'd

```
    2'b11 : begin        // Fetch and execute
                eptr = (eptr+1) % QDEPTH;
                fptr = (fptr+1) % QDEPTH;
            end
  endcase
end
end task

always @ (negedge clock)  begin : phase2_loop
   if (!reset)  begin
        if (!mem_access && !branch_taken)
            copy_results;
        if (branch_taken) pc = `DST;
        else if (!mem_access) pc = pc+1;...
        if (branch_taken || halt_found)
            flush_queue;
        else set_pointers;
        if (halt_found) begin
            $stop;
            halt_found = 0;
        end
  end
end
```

Interlock (data dependency, hazard)

Program Segment 1: No problem

```
I1:  ADD    R1, R2  // R1 = R1+R2
I2:  CMP    R3, R2  // R3 = ~R2
```

Program Segment 2: Interlock problem

// a hazard or a race condition

```
I3:  ADD    R1, R2  // R1 = R1+R2
I4:  CMP    R3, R1  // R3 = ~R1
```

The problem appears when an instruction that modifies the contents of a register in the register file is followed too closely by another instruction that attempts to read the same register. (due to the concurrency between the execution unit and the write unit)

Use NOP to solve interlock


```
I3:   ADD   R1, R2   // R1 = R1+R2
IX:   NOP            // Avoid interlock
I4:   CMP   R3, R1   // R3 = ~R1
```

Advantage: no modification of the architecture and the design is required

Disadvantages: one more cycle and higher compiler complexity

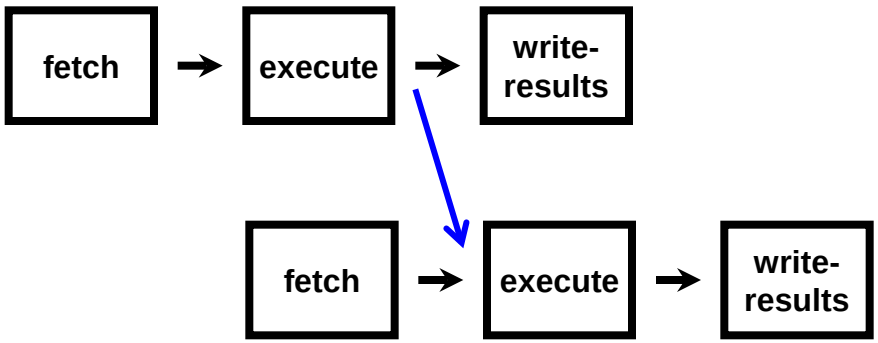
The "bypassing" technique

Use additional logic to recognize the interlocking problem and forward the required result directly to one of the operands for the current instruction.

```
reg   bypass1, bypass2;

function  [31:0] getsrc;
input  [31:0] i;
begin
   if (bypass1) getsrc = result;
   else if (`SRCTYPE === `REGTYPE)
          getsrc = RFILE[`SRC];
   else getsrc = `SRC; // immediate type
end
endfunction

function [31:0] getdst;
input [31:0]  i;
```



Cont'd.

The “bypassing” technique (cont.)


```
begin
    if (bypass2) getdst = result;
    else if(`DSTTYPE === `REGTYPE)
            getdst = RFILE[`DST];
    else $display (“Error : Immediate data cannot be destination.”);
end
endfunction

always @(do_execute) begin : execute_block
     if (!mem_access) begin
        ir = IR_Queue[eptr];       // SRC=REG type  WDST=REG type
        bypass1 = (`SRC == `WDST) && ~ir[27] && ~wir[26]);
        bypass2 = (`DST == `WDST) && ~ir[26]  && ~wir[26]);
    end       // ADD   R1, R2;   CMP    R3, R1;   ADD   R1, #1;
    execute;
    if (!mem_access) executed = 1;
end
```

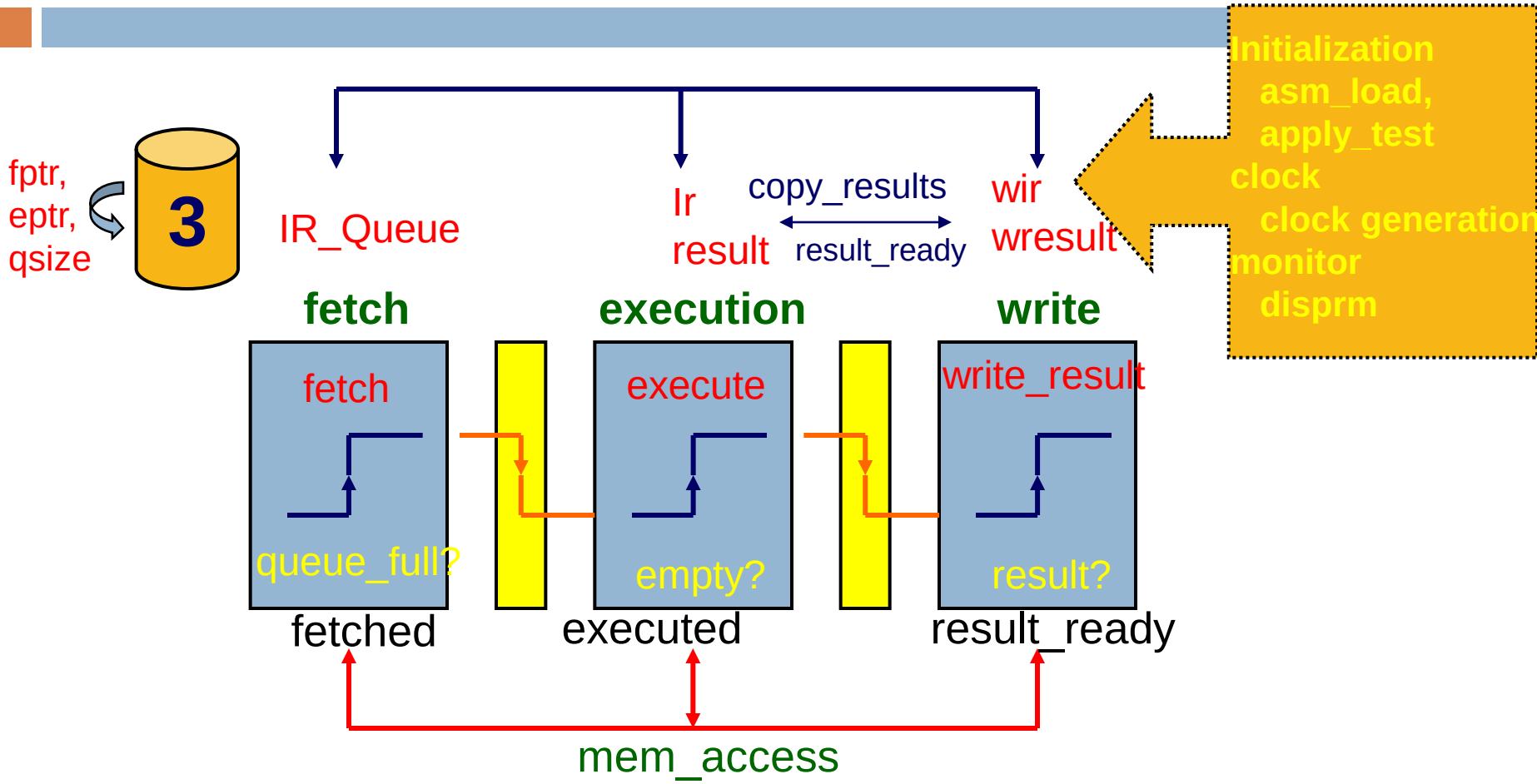
The “bypassing” technique (cont.)

Will there be a data dependency problem in the following code?

```
LD    R1, MEM[10]
ADD   R1, R2
```

Since R1 will receive right data at the execution cycle when the LD instruction is executed the second time, no bypassing is needed in our design.

Initialization



negative edge clock (phase 2) ⇨ fptr, eptr, qsize, branch_taken

Processor Model With Pipeline


```
/* ==================================
 *
 *  Model of SISC processor with pipeline
 *
 *     pipeline_model.v
 *
 *==================================*/

module pipeline_model ;

// Declare parameters

parameter CYCLE = 10;                                  // Cycle Time
parameter HALFCYCLE = (CYCLE/2)              // Half Cycle Time
parameter WIDTH = 32;                                  // Width of datapaths
parameter ADDRSIZE = 12;                        // Size of address fields
parameter MEMSIZE = (1<<ADDRSIZE);     // Size of max memory
parameter MAXREGS = 16;                        // Maximum number of registers
parameter SBITS = 5;                    // Status register bits
parameter QDEPTH = 3;                          // Instruction Queue Depth

// Declare Registers and Memory

reg [WIDTH-1:0] MEM [0:MEMSIZE-1],        // Memory
                RFILE [0:MAXREGS-1],          // Register file
```

Cont.

```
                ir,                 // Instruction Register
                src1, src2;         // Alu operation registers
reg   bypass1, bypass2;     // Solve data dependency
reg [WIDTH:0]  result;      // ALU result register
reg [SBITS-1:0] psr;        // Processor Status Register
reg [ADDRSIZE-1:0] pc;      // Program counter
reg            dir;         // rotate direction
reg            reset;       // System Reset
integer        i;           // useful for interactive debugging

// Declare additional registers for pipeline control

reg [WIDTH-1:0] IR_Queue [0:QDEPTH-1],     //Instruction Queue
                wir;        // Instruction Register for write stage
reg [2:0] eptr, fptr, qsize;    // Book keeping pointers
reg             clock;          // System Clock
reg [WIDTH:0] wresult;      // ALU result register for write stage

// Various Flags - control lines

reg   mem_access, branch_taken, halt_found;
reg   result_ready;
reg   executed, fetched;
reg   debug;
wire  queue_full;
event  do_fetch, do_execute, do_write_results;
```

Cont.

```
// General definitions

`define TRUE            1
`define FALSE           0
`define DEBUG_ON   debug  = 1
`define DEBUG_OFF  debug = 0

// Define Instruction fields

`define OPCODE   ir[31:28]
`define SRC          ir[23:12]
`define DST          ir[11:0]
`define SRCTYPE  ir[27]          // source type, 0 = reg (mem for LD), 1 = imm
`define DSTTYPE   ir[26]         // desti. type,   0 = reg, 1 = imm
`define CCODE      ir[27:24]
`define SRCNT      ir[23:12]      // Shift/rotate count  -: left, +:  right

// Define for Write instructions

`define WOPCODE wir[31:28]
`define WSRC        wir[23:12]
`define WDST        wir[11:0]

// Operand Types

`define REGTYPE     0
`define IMMTYPE      1
```

Cont.

```
// Define opcodes for each instruction

`define NOP         4'b0000
`define BRA         4'b0001
`define LD          4'b0010
`define STR         4'b0011
`define ADD         4'b0100
`define MUL         4'b0101
`define CMP         4'b0110
`define SHF         4'b0111
`define ROT         4'b1000
`define HLT         4'b1001

// Define Condition Code fields

`define CARRY       psr[0]
`define EVEN        psr[1]
`define PARITY      psr[2]
`define ZERO        psr[3]
`define NEG         psr[4]

// Define Condition Codes

// Condition Code set when ...
```

Cont.

```
`define CCC      1              // Result has carry
`define CCE      2              // Result is even
`define CCP      3              // Result has odd parity
`define CCZ      4              // Result is Zero
`define CCN      5              // Result is Negative
`define CCA      0              // Always

`define RIGHT    0              // Rotate/Shift Right
`define LEFT     1              // Rotate/Shift Left

// Continuous assignment for queue_full

assign queue_full = (qsize == QDEPTH);

// Functions for ALU operands and result

function [WIDTH-1:0] getsrc;
input [WIDTH-1:0] in;
begin
     if (bypass1) getsrc = result;
     else if(`SRCTYPE === `REGTYPE) getsrc = RFILE[`SRC];
     else getsrc = `SRC;       // immediate type
end
endfunction

function [WIDTH-1:0] getdst;
input [WIDTH-1:0] in;
```

Cont.

```
begin
    if (bypass2) getdst = result;
    else if (`DSTTYPE === `REGTYPE) getdst = RFILE[`DST];
    else begin                    // immediate type
        $display ("Error:Immediate data can't be destination.");
    end
end
endfunction

// Functions/tasks for Condition Codes

function checkcond;          // Returns 1 if condition code is set.
input [4:0] ccode;
begin
  case (ccode)
    `CCC : checkcond = `CARRY;
    `CCE : checkcond = `EVEN;
    `CCP : checkcond = `PARITY;
    `CCZ : checkcond = `ZERO;
    `CCN : checkcond = `NEG;
    `CCA : checkcond = 1;
  endcase
end
endfunction
```

Cont.

```
task clearcondcode;          // Reset condition codes in PSR
begin
   psr = 0;
end
endtask

task setcondcode;            // Compute the condition codes and set PSR
input [WIDTH:0] res;
begin
   `CARRY = res [WIDTH];
   `EVEN  = ~res [0];
   `PARITY = ^res;
   `ZERO = ~(|res);
   `NEG = res [WIDTH-1] ;
end
endtask

// Function and Tasks

task fetch;
begin
    IR_Queue [fptr] = MEM [pc];
    fetched = 1;
end
endtask
```

Cont.

```
task execute;
begin
    if (!mem_access) ir = IR_Queue [eptr] ;      // New  IR required?
    case (`OPCODE)
    `NOP: begin
              if (debug) $display ("Nop...");
          end
    `BRA: begin                                          // BRA  mem, cc
              if (debug) $write ("Branch...");
              if (checkcond (`CCODE) == 1) begin
                   pc = `DST;
                   branch_taken = 1;
              end
          end
    `LD:   begin                                         // LD   reg, mem1
               if ((mem_access == 0) begin
                    mem_access = 1;                  // Reserve next cycle
               end
               else begin                                // Memory access
                    if (debug) $display ("load...");
                    clearcondcode;
                    if (`SRCTYPE) begin
                      RFILE [`DST] = `SRC;
                    end
                    else RFILE [`DST] = MEM [`SRC];
                    setcondcode ({1'b0, RFILE [`DST]});
                    mem_access = 0;
               end
          end
```

Cont.

```
    `STR:  begin                                              // STR   mem, src
           if (mem_access == 0) mem_access = 1;   // Reserve next cycle
              else   begin                                    // Memory access
                  if (debug) $display ("Store...");
                  clearcondcode;
                  if (`SRCTYPE) MEM [`DST] = `SRC;
                  else MEM [`DST] = RFILE [`SRC];
                  mem_access = 0;
              end
             end

    // ADD, MUL, CMP, SHF, and ROT are
    // modeled similarly.

    `HLT:  begin
                $display ("Halt...");
                halt_found = 1;
           end
     default:   $display ("Error: Wrong Opcode in instruction.");
   endcase
   if (!mem_access) executed = 1;                    // Instruction executed?
end
endtask
```

Cont.

```
task write_result;
begin                          //ADD   reg, src
  if ((`WOPCODE >=  `ADD) && (`WOPCODE < `HLT)) begin
      if (`WDSTTYPE == `REGTYPE) RFILE [`WDST] = wresult;
      else $display ("Error: destination error.");
      result_ready = 0;
  end
end
endtask

task flush_queue;
begin                          // pc is already modified by branch execution
  fptr = 0;
  eptr = 0;
  qsize = 0;
  branch_taken = 0;
end
endtask

task copy_results;
begin
   if ((`OPCODE >= `ADD) && (`OPCODE< `HLT)) begin
      setcondcode (result);
      wresult = result;
      wir = ir;
      result_ready = 1;
   end
end
endtask
```

Cont.

```
task apply_reset;
begin
   result = 1;
   #2 reset = 0;
   pc = 0;
   mem_access = 0;
   qsize = 0;
   fptr = 0;
   eptr = 0;
   branch_taken = 0;
end
endtask

task disprm;
input rm;
input [ADDRSIZE-1:0] adr1, adr2;
begin
    if (rm == `REGTYPE) begin
          while (adr2 >= adr1) begin
               adr1 = adr1+1;              // display ...
          end
    end
    else begin
          while (adr2 >= adr1) begin
               adr1 = adr1+1;             // display ...
          end
     end
end
endtask
```

Cont.

```
task set_points;
begin
  case ({fetched, executed})
    2'b00: ;
    2'b01: begin
             qsize = qsize-1;
             eptr = (eptr+1) % QDEPTH;
           end
    2'b10: begin
             qsize = qsize+1;
             fptr = (fptr+1) % QEPTH;
           end
    2'b11: begin
             eptr = (eptr+1) % QDEPTH;
             fptr = (ftpr+1) % QDEPTH;
           end
  endcase
end
endtask

initial begin:  asm_load
     clock = 0;
     $readmemb ("sisc.asm", MEM);
     $monitor ("%d %b %d %h %h %h", $time, clock,pc,RFILE[0],RFILE[1],RFILE[2]);
      apply_reset;
end

always begin:  system_clock
     #5 clock = ~clock;
end
```

Cont.

```
always @ (posedge clock) begin :  phase1_loop
      if (!reset) begin
         fetched = 0;
         executed = 0;
         if (!queue_full && !mem_access)
            -> do_fetch;
         if (qsize || mem_access)
            -> do_execute;
         if (result_ready)
            -> do_write_results;
     end
end

always @ (do_fetch) begin : fetch_block
          fetch;
end

always @ (do_execute) begin: execute_block
    If (!mem_access) begin
       ir = IR_Queue [eptr];
       bypass1 = (`SRC == `WDST) && ~ir[27] && ~wir[26];
       bypass2 = (`DST == `WDST) && ~ir[26] && ~wir[26];
   end
   execute;                        // Duplicated mem_acess check (redundancy)
   if (!mem_access) executed = 1;
end
```

Cont.

```
always @ (do_write_results) begin:  write_result_block
       if (!mem_access)
           write_result;
end

always @ (negedge clock) begin:  phase2_loop
  if (!reset) begin
       if (!mem_access && !branch_taken) begin
           copy_results;
       end

       if (branch_taken) pc = `DST;            // Duplicated assignment
       else if (!mem_access) pc = pc+1;

       if (branch_taken || halt_found)
           flush_queue;
       else set_points;

      if (halt_found) begin
           halt_found = 0;
           &finish;
      end
  end
end
endmodule
```

Sample of sisc.asm


```
0010_1000_0000_0000_0000_0000_0000_0001 // LD R1, #0
0010_1000_0000_0000_0001_0000_0000_0001 // LD R1, #1
0011_0000_0000_0000_0001_0000_0000_1001 // STR Mem[9], R1
0010_1000_0000_0000_0010_0000_0000_0001 // LD R1, #2
0010_0000_0000_0000_1001_0000_0000_0001 // LD R1, Mem[9]
//0001_0000_0000_0000_0000_0000_0000_0100 // BRA addr[4], ALWAYS
1001_1111_1111_1111_1111_1111_1111_1111 // HLT
```