Design of a Pipelined RISC Microprocessor

Sunggu Lee,

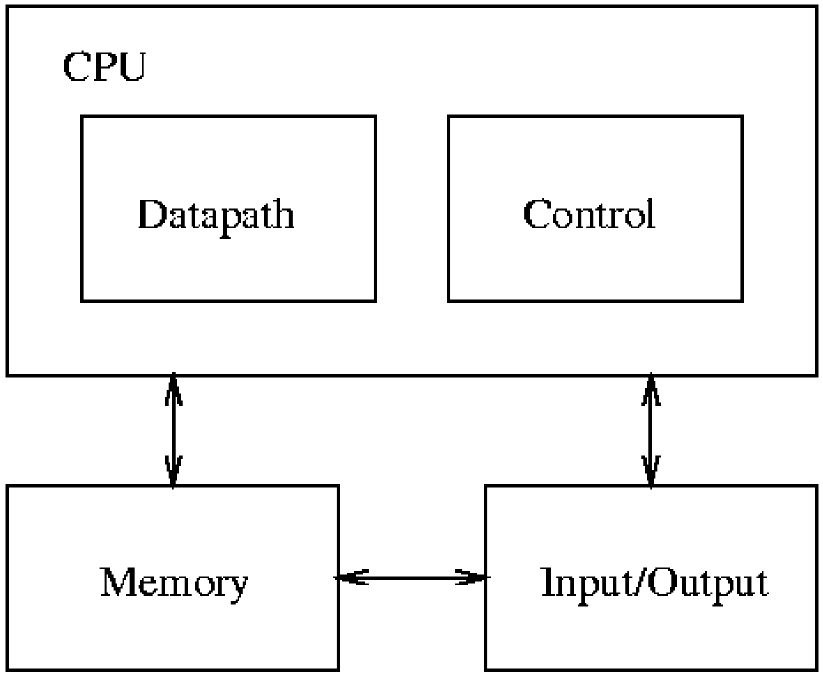
Advanced Digital Logic Design Using Verilog, State Machines, and Synthesis for FPGA, Chap. 9, Neilson, 2006.

Outlines

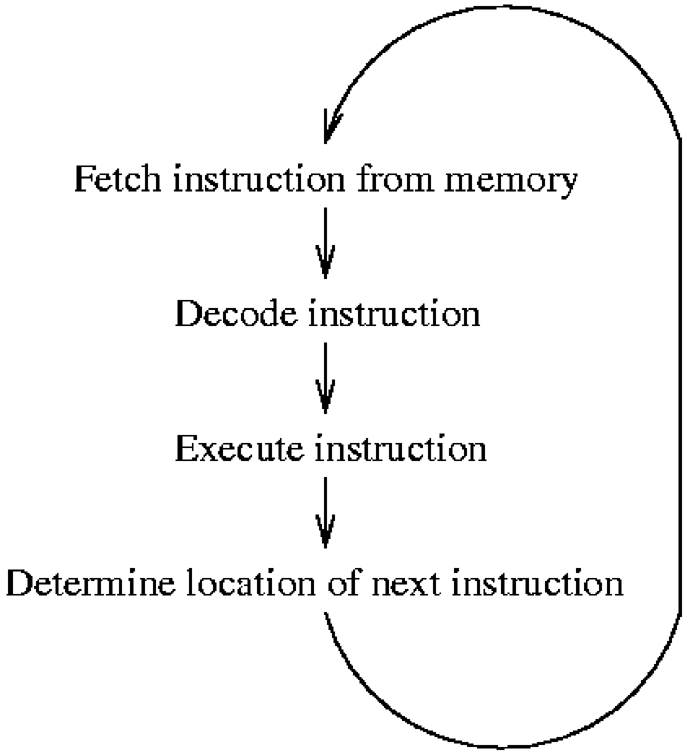
- Basic operation of a microprocessor
  - How to design a basic microprocessor circuit
- Design of an instruction pipeline for a RISC microprocessor
- Instruction set and circuit implementation of a commercial microprocessor
  - Uses THUMB microprocessor as an example
- Pipelined microprocessor design

Basic Computer System

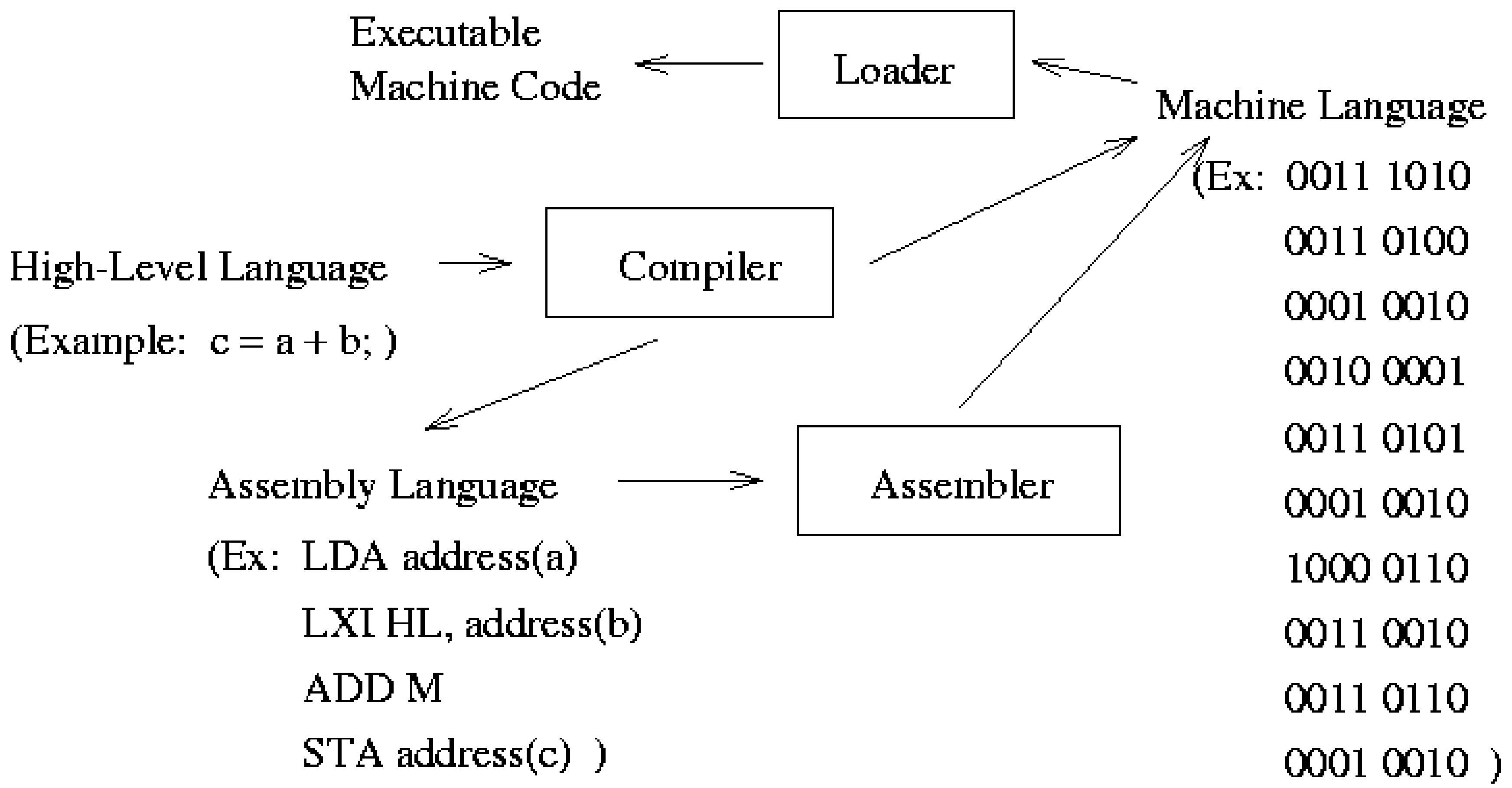
- Block Diagram



- Basic Operation


Fetch instruction from memory
Decode instruction
Execute instruction
Determine location of next instruction

Process of Preparing a Computer Program for Execution



THUMB Instruction Set

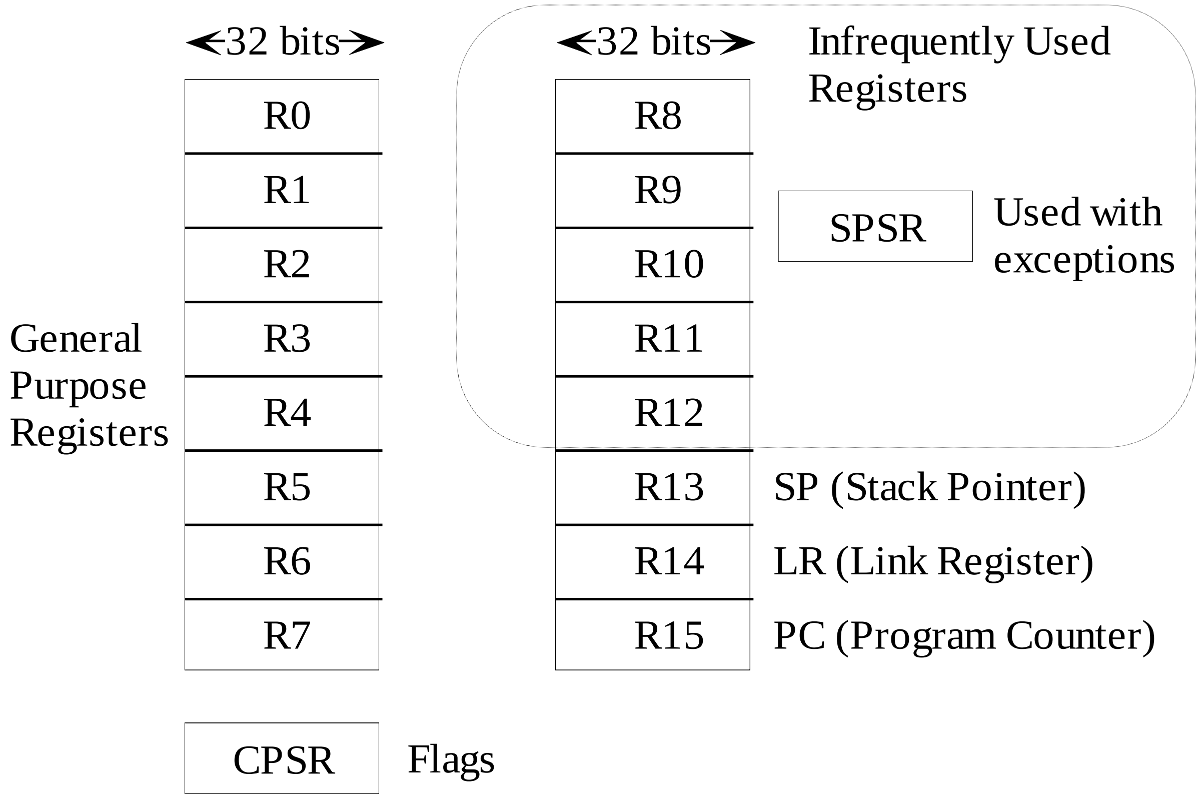
- Representative of the basic instructions supported in a modern RISC microprocessor architecture
  - Addressing modes supported
    - Immediate, direct, register, register indirect, indexed
  - Register file
    - A large set of general-purpose registers, used for temporary data storage
    - THUMB uses a small register file of 16 registers
      - Only 8 are accessed directly most of the time
  - Load-store architecture
    - Only LOAD and STORE instructions can access data memory
    - All other instructions operate from registers
  - Instruction set designed to facilitate pipelining

- Pseudocode (RTL) solution:
  - 1. Fetch M[PC] into instruction register IR
  - 2. Increment PC by 2
  - 3. Switch (IR) begin

```
     case “MOV Rd, Rn”:  Rd ← Rn;
     case “ADD Rd, Rn, Rm”:  Rd ← Rn + Rm;
     case “B #immed3”:   PC ← PC + immed3;
     …
   end;
```

  - 4. Repeat from Step 1

THUMB Programming Model (1/3)



Programming model for THUMB microprocessor

THUMB Programming Model (2/3)

- General purpose registers
  - R0 through R7: normal visible set
  - R8 through R15: “high” set of registers, infrequently accessed directly
- SP (Stack Pointer) → R13
  - Points to the top of the “stack” region
- PC (Program Counter) → R15
  - Points to the address of the instruction to be executed
- LR (Link Register) → R14
  - Used to save “return address” during a subroutine call

THUMB Programming Model (3/3)

- Condition code bits (flags)
  - Stored in two “status” registers
    - CPSR (Current Program Status Register)
      - Contains most widely used flag bits
    - SPSR (Saved Program Status Register)
      - Used for interrupt (exception) processing
- Commonly used condition code (flag) bits
  - negative (N), zero (Z), carry (C), overflow (O)

THUMB Instruction Set (1/4)

- Subset of ARM instruction set
- Main instruction types
  - Branch
  - Data processing
  - Load/store
  - Exception-related

THUMB Instruction Set (2/4)

- Branch Instructions
  - Unconditional branch
    - Branches to new location “PC+offset” (offset is positive or negative)
      - PC-relative or relative addressing
  - Conditional branch
    - Checks a set of flag bits (refer to condition code table)
    - If (condition satisfied)
      then PC $\leftarrow$ new branch address
      else PC $\leftarrow$ next sequential address location
  - Subroutine CALL
    - Must save previous PC value before branching
      - Previous PC value can be saved on stack (PUSH) or in LR
    - Subroutine call in THUMB $\rightarrow$ “branch with link” (save PC in LR)

THUMB Instruction Set (3/4)

- Data processing instructions
  - Data movement instructions
    - Read from memory or a register, store in register
    - Read from a register, store into register or memory
  - Arithmetic instructions
    - Add, subtract, negate, multiply, divide, compare, test
    - Condition codes set based on operation result
  - Logical instructions
    - OR, AND, exclusive-OR
  - Shift and rotate instruction
    - Logical shift and rotate → shift in ‘0’ bits
    - Arithmetic shift → preserve “sign” (copy msb bit during right-shift)

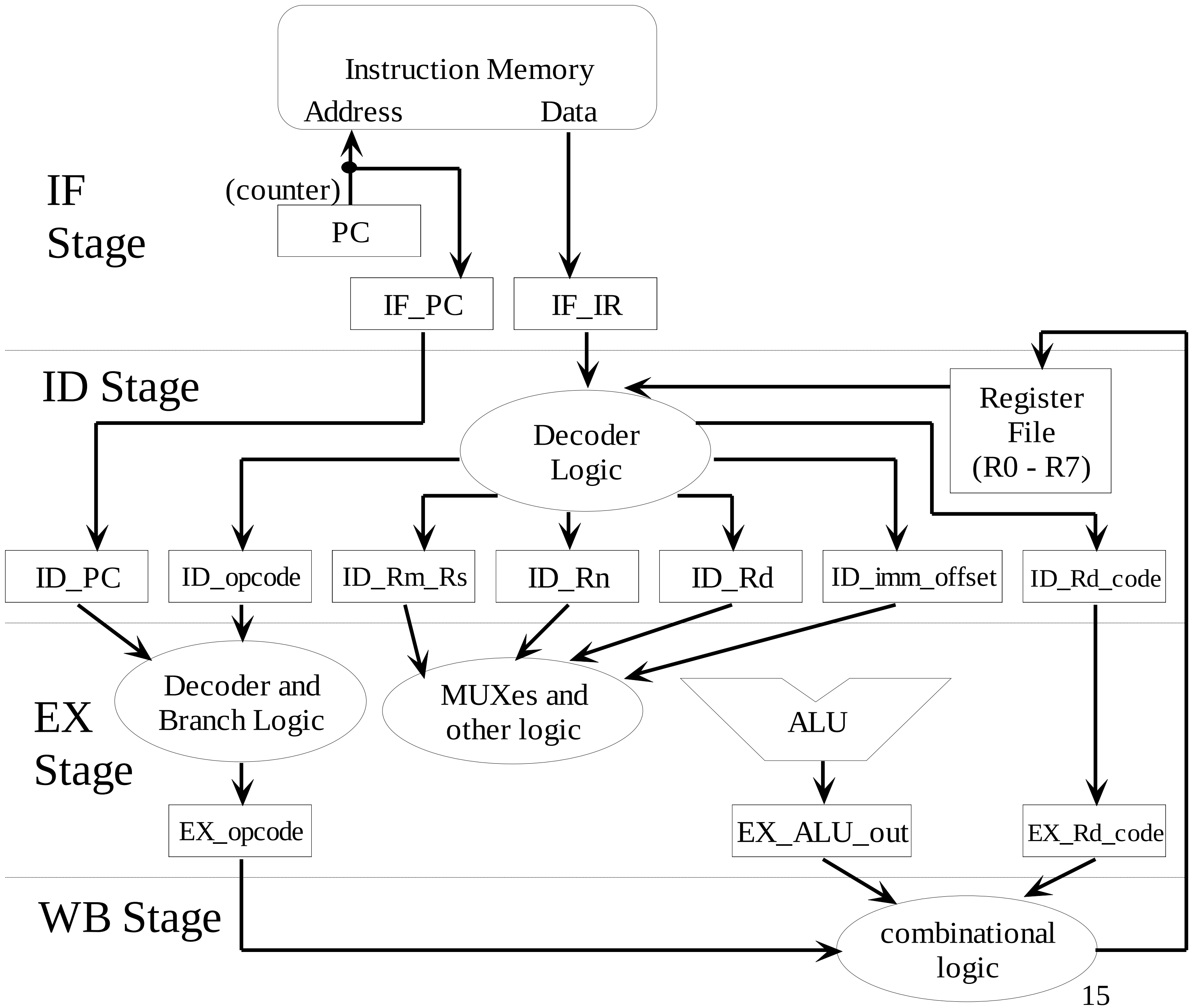
THUMB Instruction Set (4/4)

- Interrupt and Other Instructions
  - SWI: software interrupt
    - Causes a SWI exception (interrupt) to occur
      - Handled by operating system subroutine
    - Typically used to call operating system services
      - Switch from “user” mode to “operating system” mode
  - BKPT: breakpoint
    - Used to generate software breakpoints
    - Typically used with debugging hardware or software

Pipelined THUMB Verilog Code

- Written using one “always” block for each stage of the 4-stage instruction pipeline used in the THUMB implementation
  - IF: instruction fetch
  - ID: instruction decode (and operand fetch)
  - EX: execute
  - WB: write back (store results in destination registers)
- Based on block diagram design of the figure (next slide)
  - Pipeline registers used to transfer data between pipeline stages
  - Note: A single variable (even a “for loop” index variable) must not be assigned values in two or more always blocks!
    - Results in assignment conflicts and unsynthesizable code

Pipelined THUMB Block Diagram

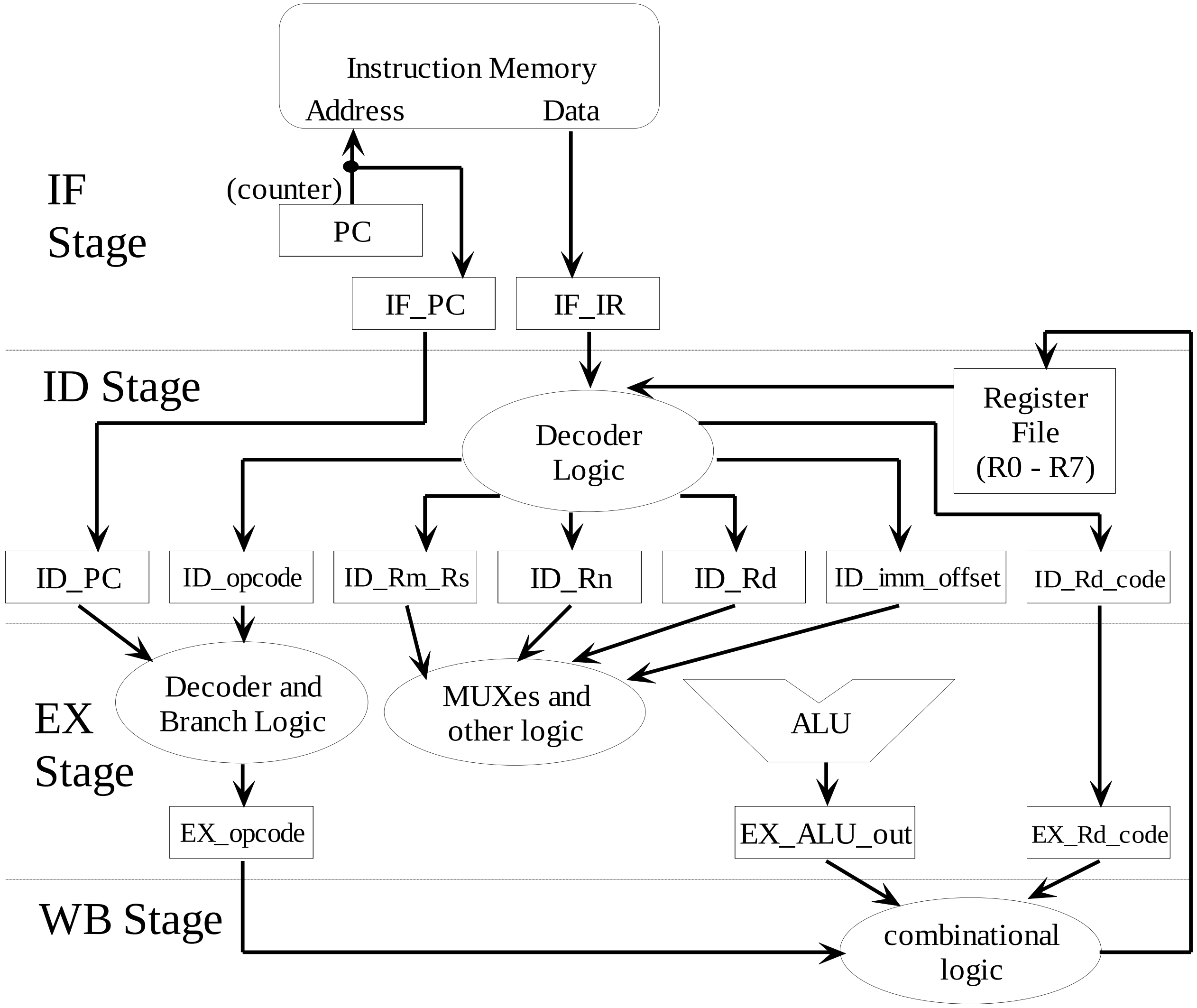


Verilog Codes for pipelined THUMB

- pipelined THUMB code by Sunggu Lee
  - complete Verilog code for a 16-bit pipelined RISC microprocessor
- test bench

Verilog THUMB implementation

- 4 pipeline stages using separate **always** blocks



constant definitions

- define opcode of 59 instructions using 6 bits
- undefined instruction 0xDE00 can be used as a NULL operation (NOP)

```
// OPCODE DEFINITIONS
// Undefined Instruction Thumb Opcode
`define UNDEFINED_INSTRUCTION 16'hDE00 // 16 = `HWORD_SIZ
E
// Internal Encoding for Thumb Instructions Supported in "thumb.v"
`define ADC       6'b00_0000
`define ADD_1      6'b00_0001
`define ADD_2      6'b00_0010
`define ADD_3      6'b00_0011
`define ADD_5      6'b00_0100
`define ADD_6      6'b00_0101
`define ADD_7      6'b00_0110
`define AND       6'b00_0111
.
.
.
`define SUB_1      6'b11_0101
`define SUB_2      6'b11_0110
`define SUB_3      6'b11_0111
`define SUB_4      6'b11_1000
`define SWI       6'b11_1001
`define TST       6'b11_1010
`define UNDEF      6'b11_1111
////////////////////////////////////////////////////////////////
```

THUMB module

- signal declarations
- a function definition *condition_passed*
  - for conditional branch instruction
- continuous assignment **assign**
- 4 **always** blocks for 4 pipeline stages

signal declaration

```
output read_instruction_n;  // enable read from instruction mem
 output [`WORD_SIZE-1:0] instruction_address; // instr. address
 input [`HWORD_SIZE-1:0] instruction;  // current instruction
 output read_data_n;     // enable read from data memory
 output write_data_n;    // enable write to data memory
 output [`WORD_SIZE-1:0] data_address; // address of data
 inout [`WORD_SIZE-1:0] data;          // current data
 input reset_n;    // active-low RESET signal
 input clk;        // clock signal

 // SIGNAL DECLARATIONS for chip inputs and outputs
 reg read_instruction_n, read_data_n, write_data_n;
 wire [`WORD_SIZE-1:0] instruction_address;
 reg [`WORD_SIZE-1:0] data_address;
 wire [`HWORD_SIZE-1:0] instruction;
 wire [`WORD_SIZE-1:0] data;
 wire reset_n;
 wire clk;

 // SIGNAL DECLARATIONS for internal registers and wires
…

 // SIGNAL DECLARATIONS used to aid in the Verilog description
 …
```

assign

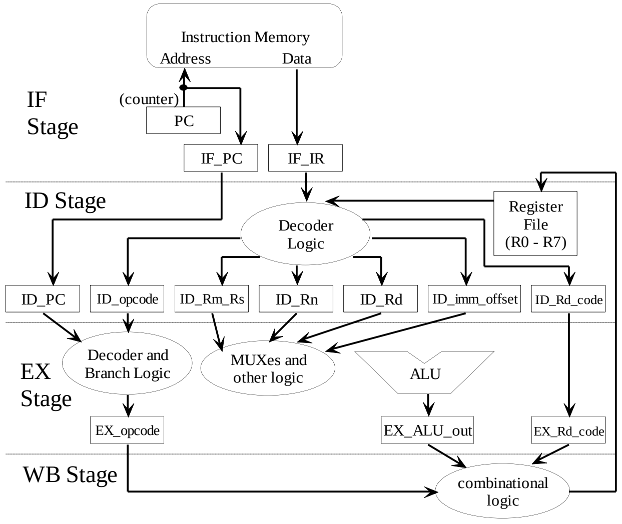
```
// ASSIGN STATEMENTS

 assign  instruction_address = PC;  // set instr. address to PC
 assign  data = (~write_data_n) ? DR : 'bz;  // tri-state data
                                     // DR: data to be stored back to memory
```

function

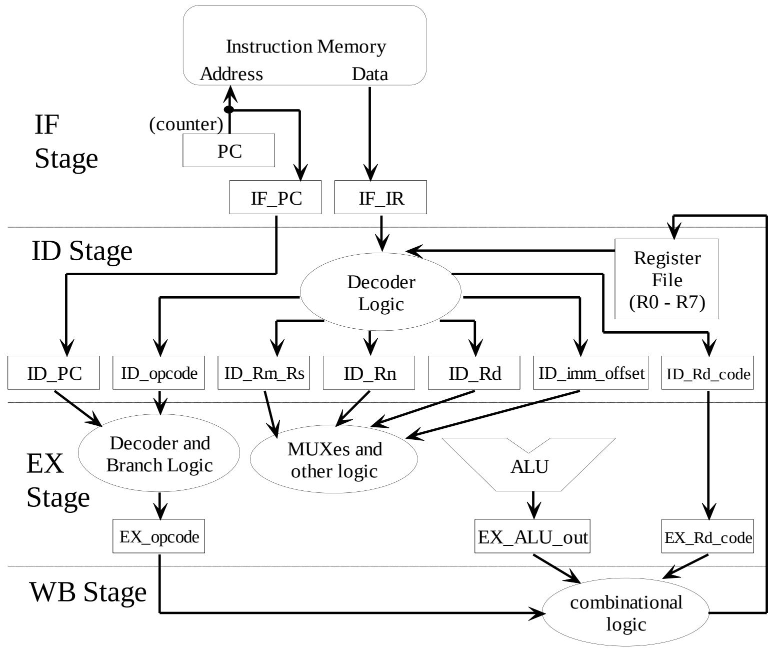
```
// Function to check condition codes
  function condition_passed;
    input [3:0] cond_code;  // 4-bit Thumb/ARM condition code
  begin
    case (cond_code)  // codes in Table 3-1 of [ARM 2000]
      4'b0000: condition_passed = Z_Flag;  // EQ (equal)
      4'b0001: condition_passed = ~Z_Flag; // NE (not equal)
      4'b0010: condition_passed = C_Flag;  // CS/HS (carry set)
      4'b0011: condition_passed = ~C_Flag; // CC (carry clear)
      4'b0100: condition_passed = N_Flag;  // MI (minus)
      4'b0101: condition_passed = ~N_Flag; // PL (plus)
      4'b0110: condition_passed = V_Flag;  // VS (overflow)
      4'b0111: condition_passed = ~V_Flag; // VC (no overflow)
      4'b1000: condition_passed = (C_Flag & (~Z_Flag)); // HI
      4'b1001: condition_passed = ((~C_Flag) & Z_Flag); // LS
      4'b1010: condition_passed = (N_Flag == V_Flag);  // GE
      4'b1011: condition_passed = (N_Flag != V_Flag);  // LT
      4‘b1100: condition_passed = ~Z_Flag & (N_Flag == V_Flag); // GT (greater
than)
      4‘b1101: condition_passed = Z_Flag & (N_Flag != V_Flag); // LE (less or eq)
      4'b1110: condition_passed = 1'b1;    // AL (always)
      default: condition_passed = 1'b1;   // note: 4'b1111 invalid
    endcase
  end  // of function body
  endfunction  // of function condition_passed
```

IF stage (1/2)



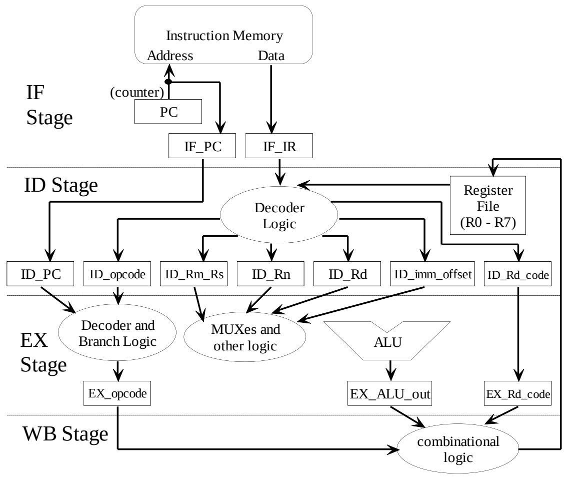
- read-enable signal (*read_instruction_n*) asserted and a valid address stored into PC
- during the next cycle, data returned from memory is latched into IF_IR pipeline register
  – when the instruction address of next instruction is being prepared (by storing the address into PC), the current instruction (whose address was output during the previous cycle) can be fetched and stored into IF_IR

IF stage (2/2)



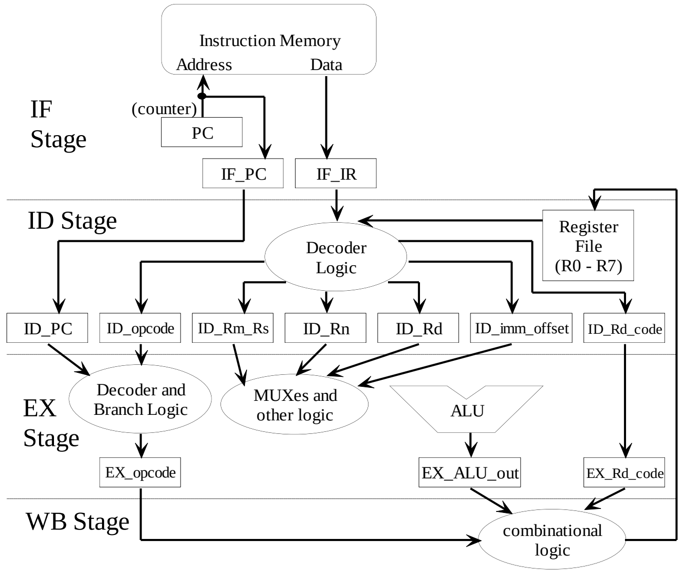
- when a branch operation (*branch_taken*) is detected
  - branch target is loaded into PC
  - the instruction fetched at this cycle (at PC+2) is incorrect
  - IR register filled with “undefined instruction”
    - pipeline stall or bubble

IF stage



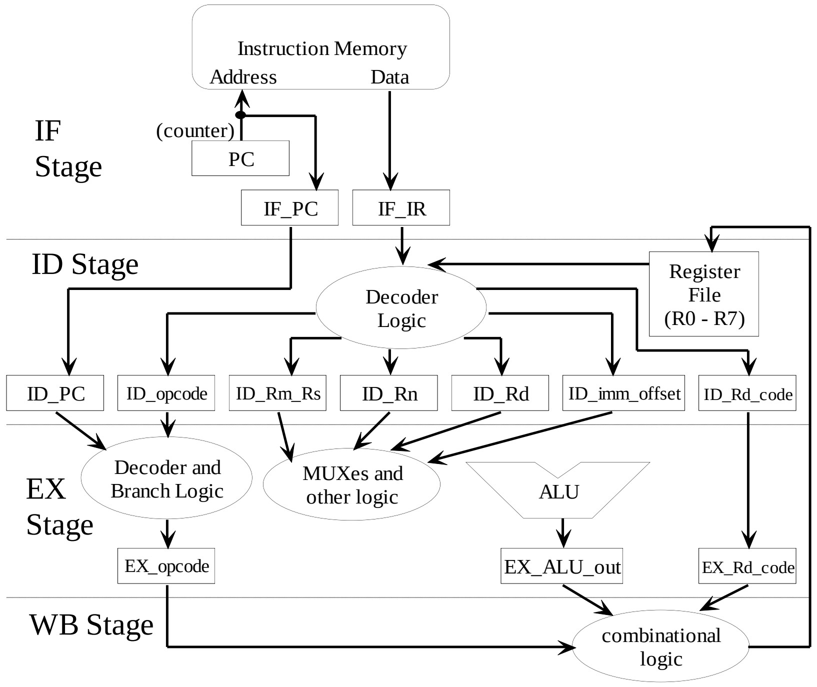
```
always @(negedge reset_n or posedge clk) begin
   if (~reset_n) begin
       PC <= 0;                              // start fetching from location 0
       read_instruction_n <= 0;      // and start memory read
   end
   else begin       // on positive clock edge,
       // execute operations and then save values in pipeline reg
       read_instruction_n <= 0; // read next instruction
       if (branch_taken) begin  // determine next instruction
          PC <= branch_target;   // operation for IF stage
          IF_IR <= `UNDEFINED_INSTRUCTION;  // to create a pipeline stall bubble
          IF_PC <= branch_target; end
       else begin
          PC <= PC + 2;        // operation for IF stage
          IF_IR <= instruction; // read instruction and store in IR
          IF_PC <= PC + 2; end   // save next instruction address
  end
end  // of IF stage
```

ID stage



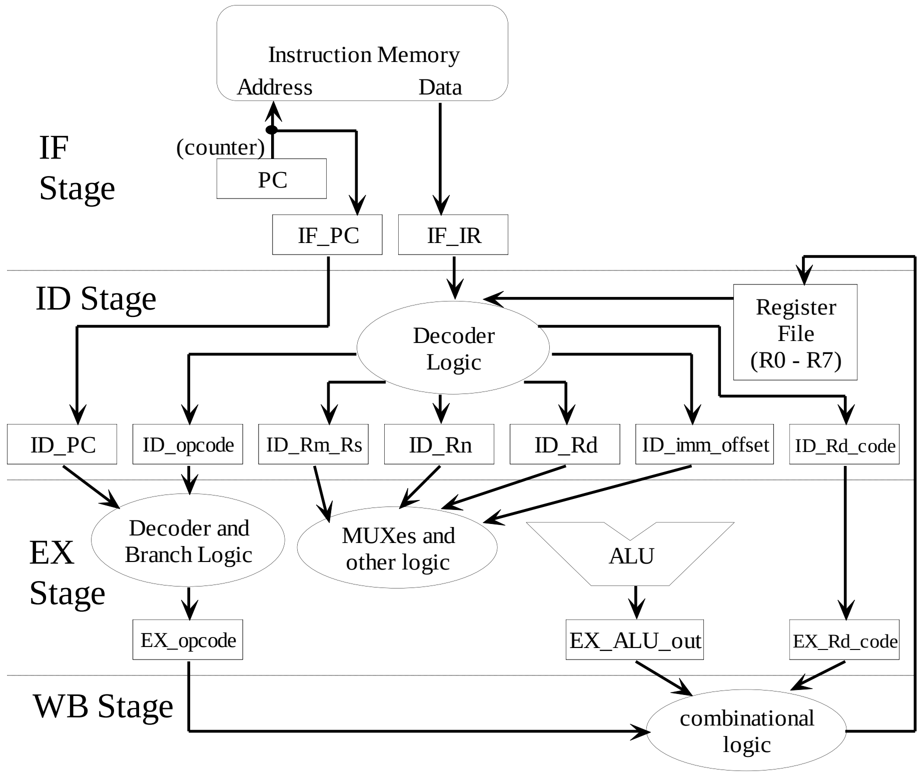
- instruction stored in IF_IR is decoded
- data to be used in the EX stage is stored into a set of common registers
  – decoder and random logic required to move data fr om register file and IF_IR is included in ID stage instead of EX stage
- This **always** block essentially consists of a set of **case** blocks

ID stage



```
always @(posedge clk) begin
    case (IF_IR[`HWORD_SIZE-1:13])
    3'b000: begin // shift by immediate or add/sub
            case (IF_IR[12:11])
            2'b11: begin
                    case (IF_IR[10:9])
                    2'b10: begin // if imm = 000, same as MOV_2
                            ID_opcode <= `ADD_1;
                            ID_imm_offset[2:0] <= IF_IR[8:6];
                            ID_Rn <= R[IF_IR[5:3]];
                            ID_Rd <= R[IF_IR[2:0]];
                            ID_Rd_code <= IF_IR[2:0];
                            end
    …
```

EX stage

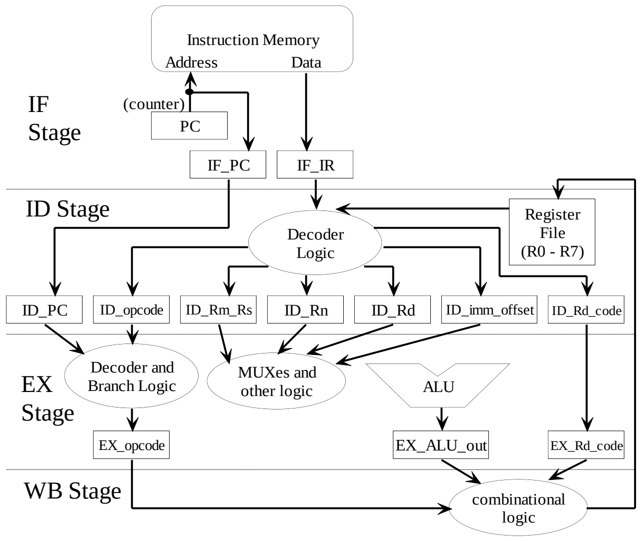


- a set of **case** blocks, simpler than ID stage
- read and write control signals for data me mory are initialized at the beginning
- some instructions (such as MUL) might requires more execution time than others

EX stage

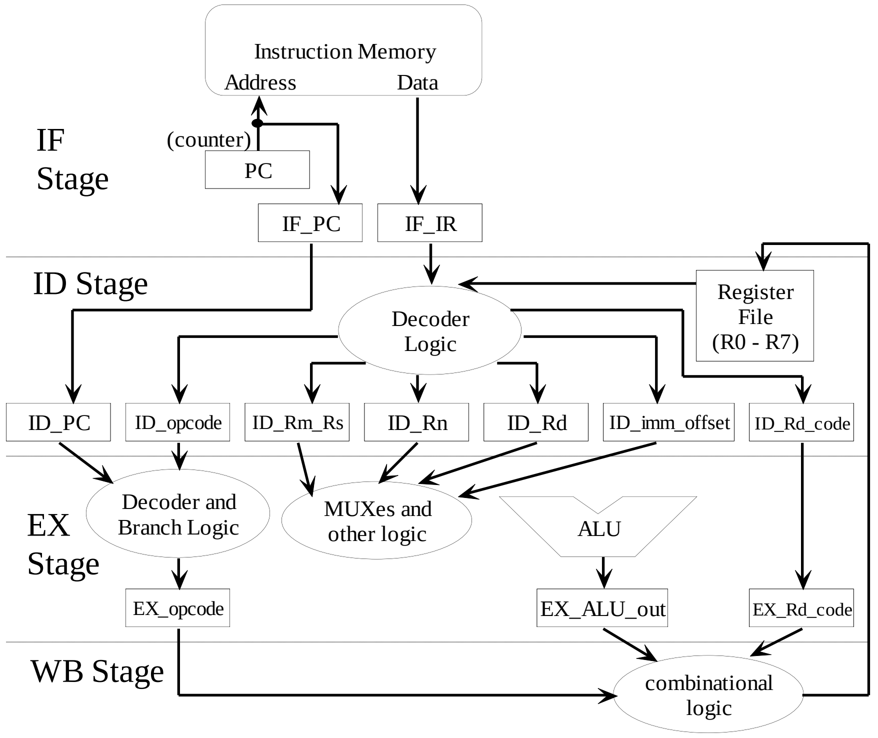
```
always @(negedge reset_n or posedge clk) begin
  if (~reset_n) begin
    branch_taken <= 1'b0;  // initialize to branch not taken
    read_data_n <= 1'b1;   // disable data memory
    write_data_n <= 1'b1;
  end
  else begin  // on positive clock edge
    read_data_n <= 1'b1;   // set default values for data mem.
    write_data_n <= 1'b1;
    if (branch_taken) begin
      EX_opcode <= `UNDEF;
      branch_taken <= 1'b0;
    end
    else begin
      case (ID_opcode)  // 1st part of EX operations
         `ADC:   ALU_out = ID_Rd + ID_Rm_Rs + C_Flag;
      `ADD_1:  ALU_out = ID_Rn + ID_imm_offset[2:0];
          …
      endcase  // end of Ist part of EX

      case (ID_opcode)  // 2nd part of EX operations
           `ADD_1: begin    // note: (imm == 000) implies MOV_2
                       EX_ALU_out[`WORD_SIZE-1:0] <=ALU_out[`WORD_SIZE-1:0];
                       …
      endcase // end of2nd part of EX
   …
```



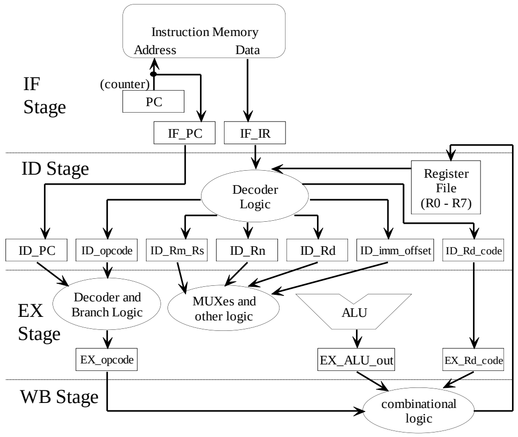
WB stage

- execution result (typically the *EX_ALU_out* register value), possibly padded with 0's, is stored into register file at the address indicated by *EX_Rd_code*

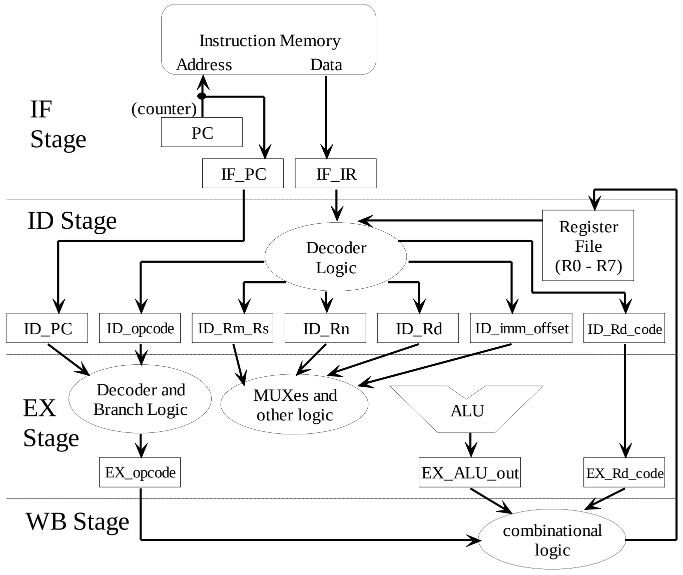


WB stage

```
always @(negedge reset_n or posedge clk) begin
  if (~reset_n)
    for (wb_i = 0;  wb_i < `REG_FILE_SIZE;  wb_i = wb_i + 1)
      R[wb_i] <= 'hffffffff;  // initialize general registers
  else
      case (EX_opcode)
      `ADC, `ADD_1, `ADD_2, `ADD_3, `ADD_5, `ADD_6, `AND,
      `ASR_1, `ASR_2, `BIC, `EOR, `LSL_1, `LSL_2, `LSR_1,
      `LSR_2, `MOV_1, `ROR, `SBC, `SUB_1, `SUB_2, `SUB_3, `SUB_4:
                    R[EX_Rd_code] <= EX_ALU_out;  // write back to Rd
          …
      `POP_R0: begin
                   found_wb_i = 0;
                   for (wb_i = 0;  wb_i < `REG_FILE_SIZE;  wb_i = wb_i + 1)
                     if (EX_imm_offset[wb_i])    // at least 1 bit must = 1
                        found_wb_i = wb_i;
                   R[found_wb_i] <= data;
                  end // of case `POP_R0;
     `UNDEF: found_wb_i = 32'bx;
      default: found_wb_i = 32'bx;
     endcase
 end // of WB stage
```



thumb_tb (test bench)



- need to model instruction and data memory
- an **always** block to generate the test vectors
  - simply create reset and clock signals
  - actual test is controlled by content of instruction memory
- another **always** block to verify the results
  - wait for an appropriate time and check whether the expected outputs appear on the address and data buses
  - including “store” instruction to observe values of specific regi ster through output data line

test bench

```
module tb_thumb();
…
// instantiate the unit under test
thumb UUT (read_instruction_n,  instruction_address,  instruction,
                  read_data_n, write_data_n, data_address, data,
                  reset_n,  clk);
…
// model read process for instruction memory
assign instruction = read_instruction_n ? ‘bz : output_instruction;
…
// model read process for data memory
assign data = read_data_n ? `WORD_SIZE’bz : output_data;
…
// model write process for data memory
…
// store programs and data in the instruction and memory
…
// test output of program for verification
...
endmodule
```