Design of a Pipelined RISC Microprocessor

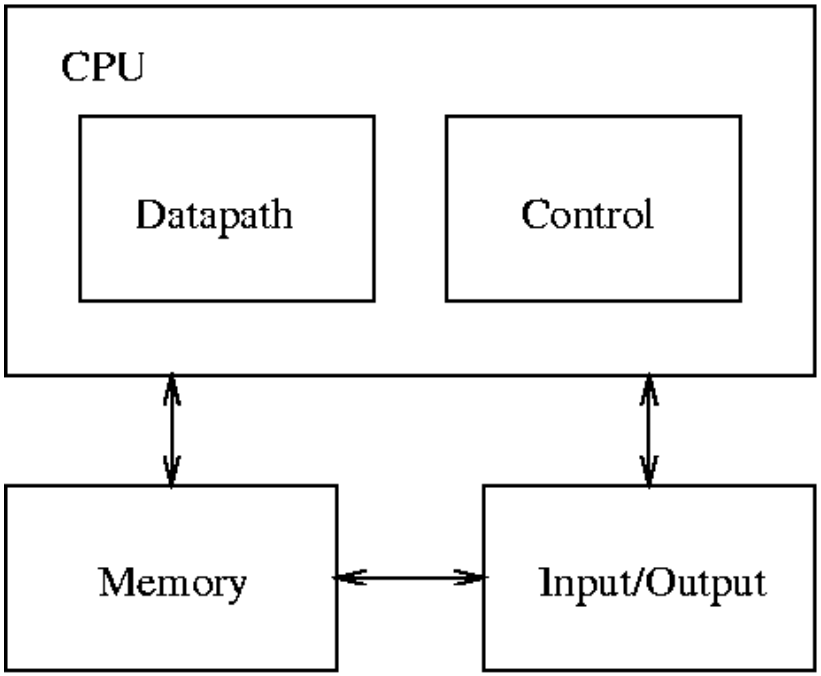
Sunggu Lee,
Advanced Digital Logic Design Using Verilog, State Machines, and Synthesis for FPGA, Chap. 9, Neilson, 2006.

Outlines

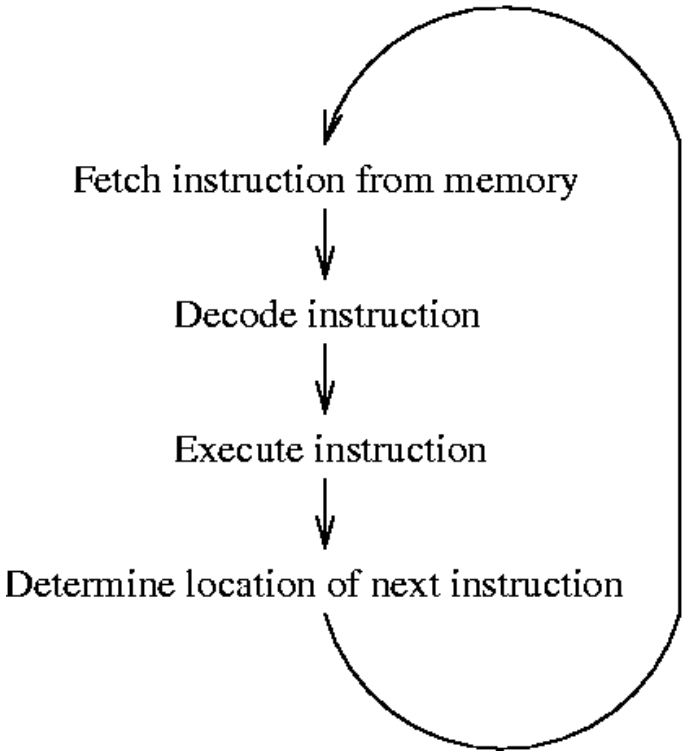
- Basic operation of a microprocessor
  - How to design a basic microprocessor circuit
- Design of an instruction pipeline for a RISC microprocessor
- Instruction set and circuit implementation of a commercial microprocessor
  - Uses THUMB microprocessor as an example
- Pipelined microprocessor design

Basic Computer System

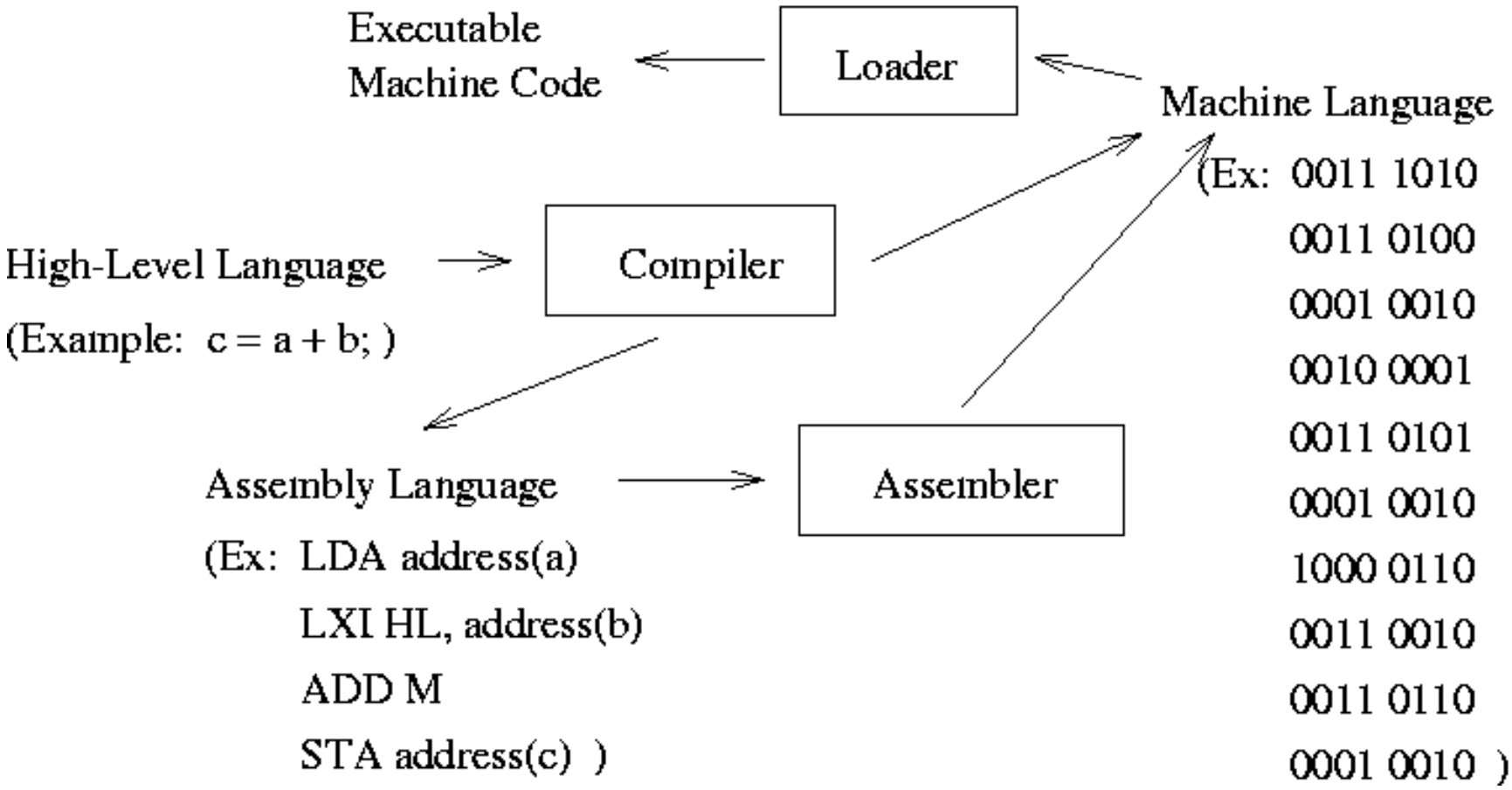
- Block Diagram



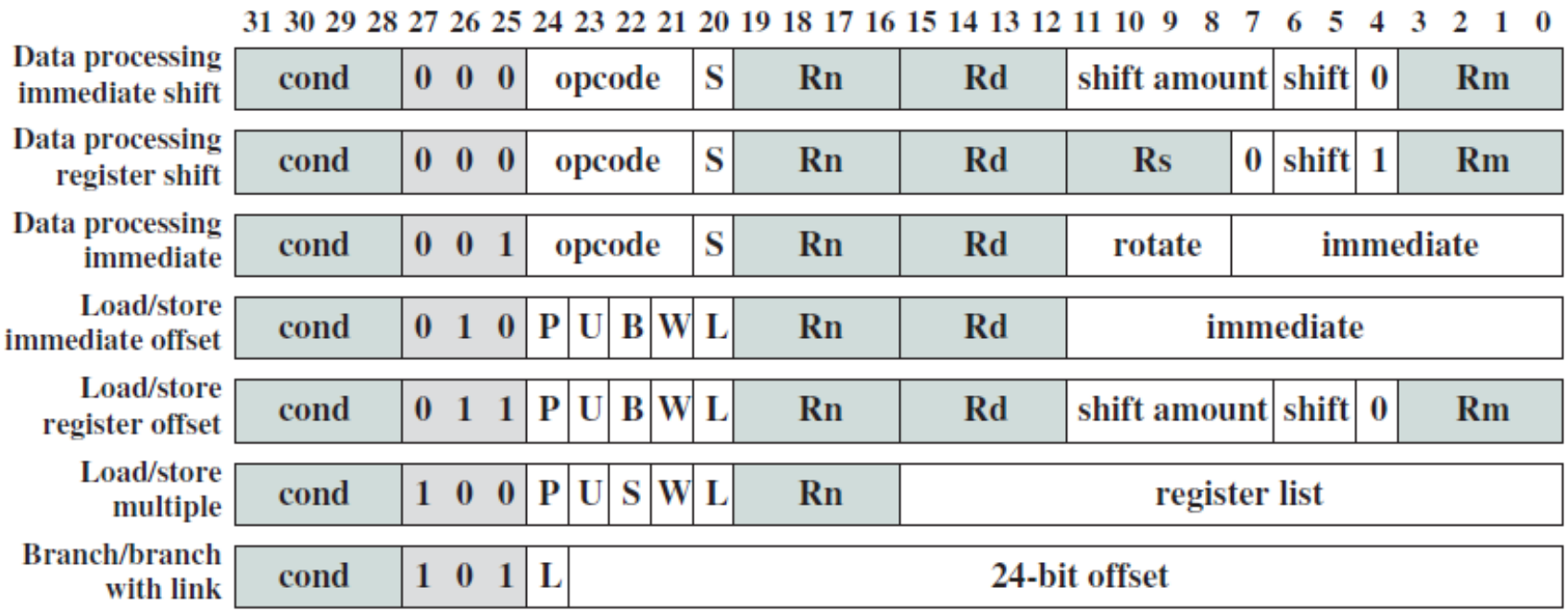
- Basic Operation



Process of Preparing a Computer Program for Execution



ARM Instruction Format



S = For data processing instructions, signifies that the instruction updates the condition codes

S = For load/store multiple instructions, signifies whether instruction execution is restricted to supervisor mode

P, U, W = bits that distinguish among different types of addressing mode

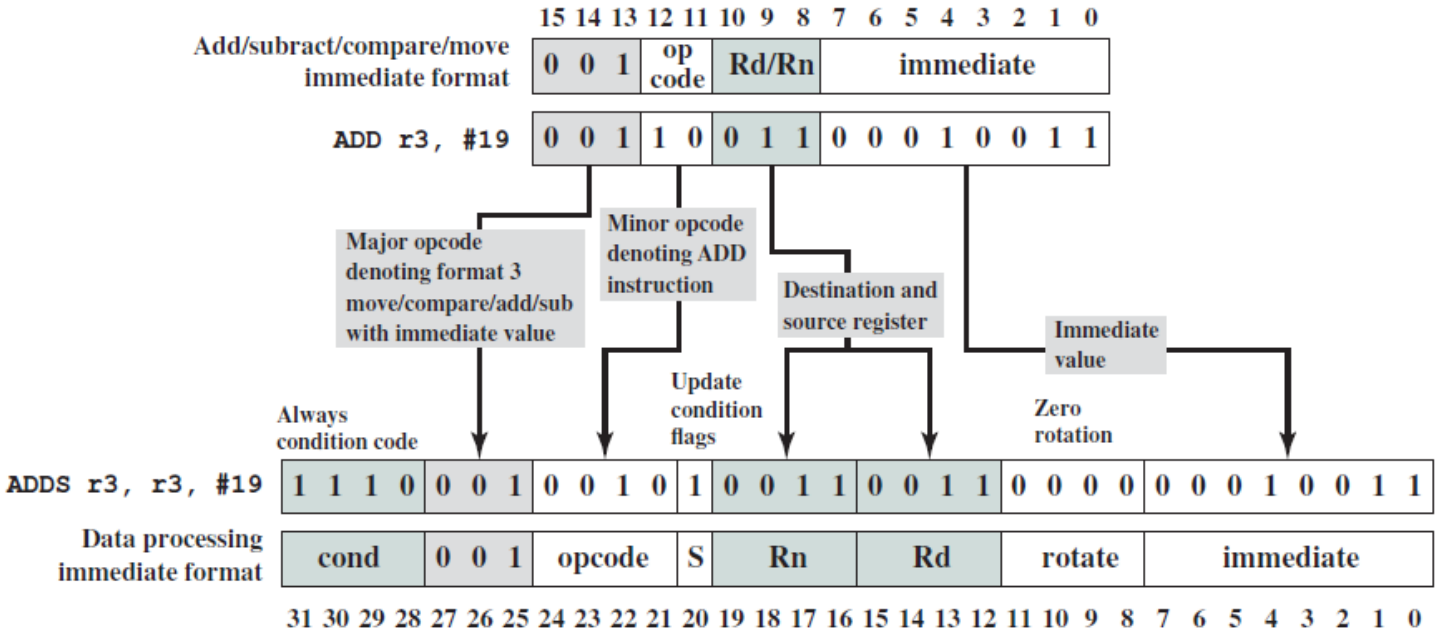
B = Distinguishes between an unsigned byte (B==1) and a word (B==0) access

L = For load/store instructions, distinguishes between a Load (L==1) and a Store (L==0)

L = For branch instructions, determines whether a return address is stored in the link register

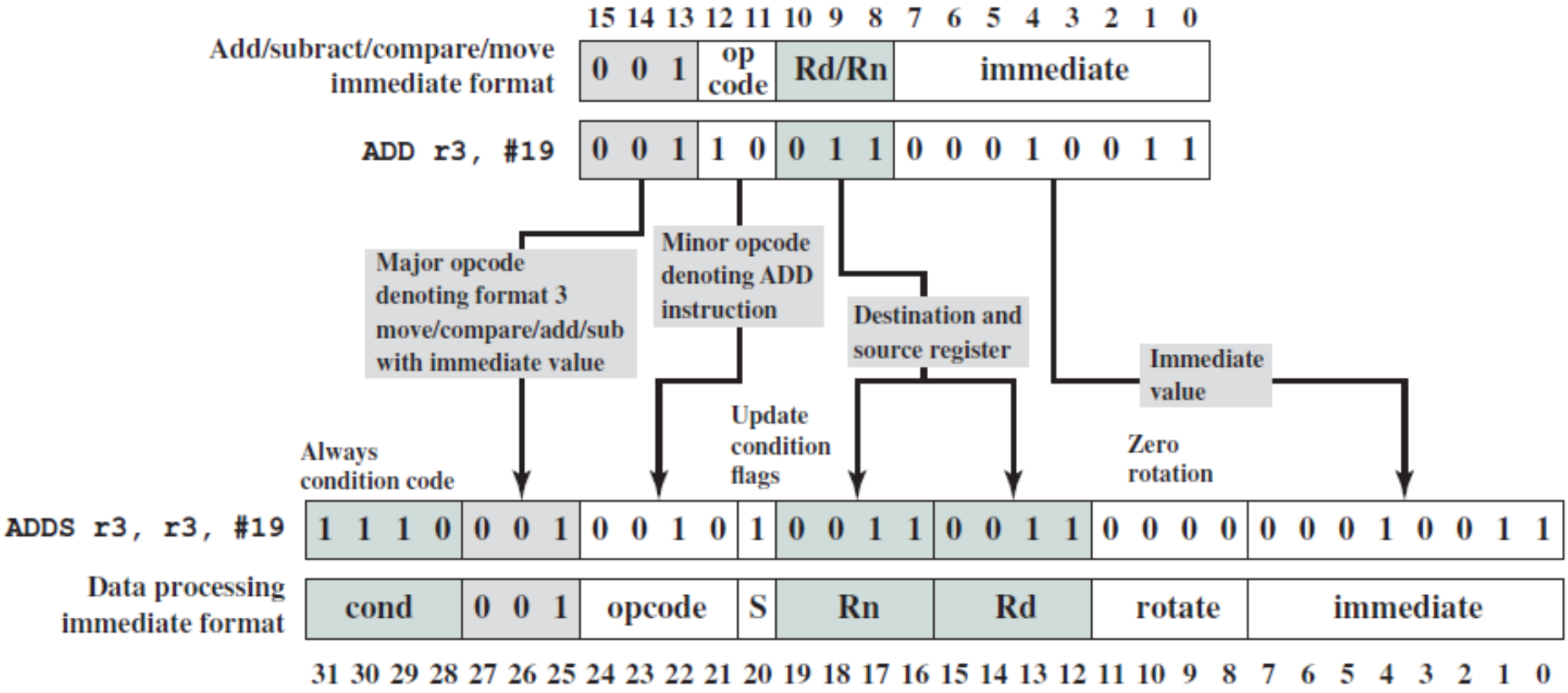
THUMB Instructions

- Unconditional
  - No condition code field (4-bit saving)
- ALU instructions always update flags
  - No update-flag bit (1-bit saving)
- Subset of ARM instructions
  - 2-bit opcode field, and 3-bit type field
  - 2-bit saving
- 8 registers (instead of 16 registers)
  - 3-bit (instead of 4-bit) register reference
- No rotate field for immediate values
  - 4-bit saving



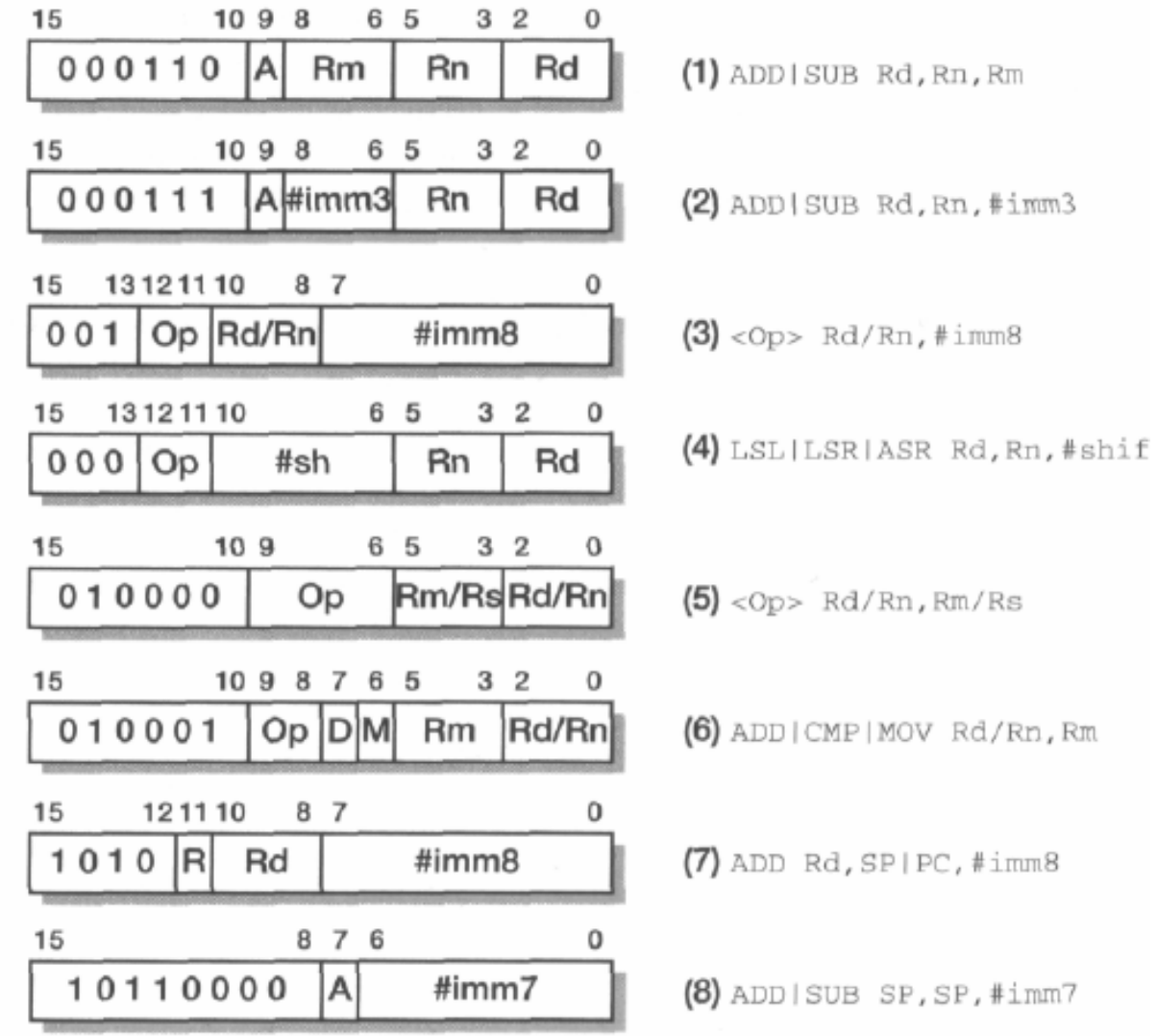
Expanding THUMB into ARM

- ARM instructions: 32-bit
- THUMB instructions: 16-bit



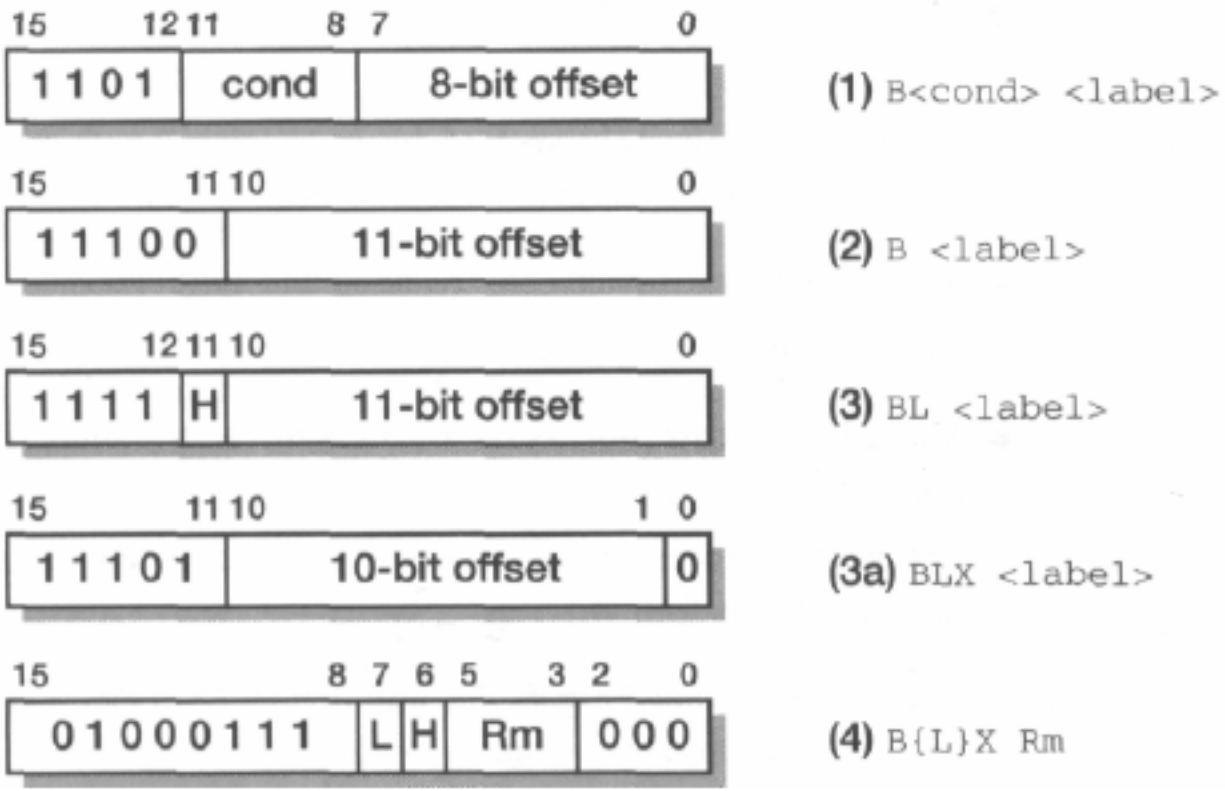
THUMB data processing instruction

- e.g. ADD, SUB, …



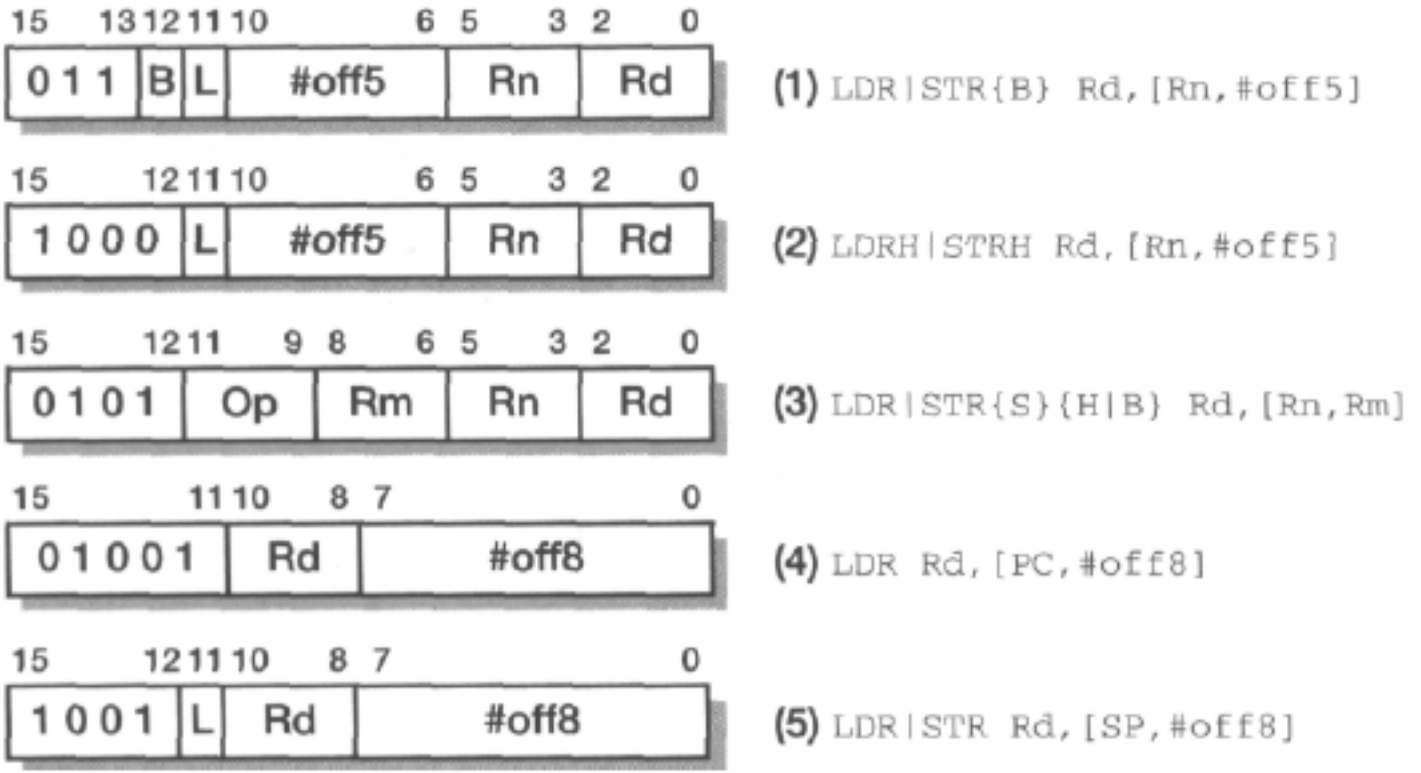
THUMB branch instruction

- e.g., Brach



THUMB register transfer instruction

- e.g. LDR



THUMB Instruction Classes (1/2)

- Refer to THUMB Set Listing

Instruction classes (indexed by *op*)

```
LSL | LSR
ASR
ADD | SUB
ADD | SUB
MOV | CMP
ADD | SUB
AND | EOR | LSL | LSR
ASR | ADC | SBC | ROR
TST | NEG | CMP | CMN
ORR | MUL | BIC | MVN
CPY Ld, Lm
ADD | MOV Ld, Hm
ADD | MOV Hd, Lm
ADD | MOV Hd, Hm
CMP
CMP
CMP
BX | BLX
LDR Ld, [pc, #immed*4]
STR | STRH | STRB | LDRSB pre
LDR | LDRH | LDRB | LDRSH pre
STR | LDR Ld, [Ln, #immed*4]
STRB | LDRB Ld, [Ln, #immed]
STRH | LDRH Ld, [Ln, #immed*2]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	*op*	*immed5*					*Lm*			*Ld*		
0	0	0	1	0	*immed5*					*Lm*			*Ld*		
0	0	0	1	1	0	*op*	*Lm*			*Ln*			*Ld*		
0	0	0	1	1	1	*op*	*immed3*			*Ln*			*Ld*		
0	0	1	0	*op*	*Ld/Ln*			*immed8*							
0	0	1	1	*op*	*Ld*			*immed8*							
0	1	0	0	0	0	0	0	*op*		*Lm/Ls*			*Ld*		
0	1	0	0	0	0	0	1	*op*		*Lm/Ls*			*Ld*		
0	1	0	0	0	0	1	0	*op*		*Lm*			*Ld/Ln*		
0	1	0	0	0	0	1	1	*op*		*Lm*			*Ld*		
0	1	0	0	0	1	1	0	0	0	*Lm*			*Ld*		
0	1	0	0	0	1	*op*	0	0	1	*Hm & 7*			*Ld*		
0	1	0	0	0	1	*op*	0	1	0	*Lm*			*Hd & 7*		
0	1	0	0	0	1	*op*	0	1	1	*Hm & 7*			*Hd & 7*		
0	1	0	0	0	1	0	1	0	1	*Hm & 7*			*Ln*		
0	1	0	0	0	1	0	1	1	0	*Lm*			*Hn & 7*		
0	1	0	0	0	1	0	1	1	1	*Hm & 7*			*Hn & 7*		
0	1	0	0	0	1	1	1	*op*	*Rm*				0	0	0
0	1	0	0	1	*Ld*			*immed8*							
0	1	0	1	0	*op*		*Lm*			*Ln*			*Ld*		
0	1	0	1	1	*op*		*Lm*			*Ln*			*Ld*		
0	1	1	0	*op*	*immed5*					*Ln*			*Ld*		
0	1	1	1	*op*	*immed5*					*Ln*			*Ld*		
1	0	0	0	*op*	*immed5*					*Ln*			*Ld*		

THUMB Instruction Classes (2/2)

Instruction classes (indexed by *op*)

`STR | LDR Ld, [sp, #immed*4]`
`ADD Ld, pc, #immed*4 |`
`ADD Ld, sp, #immed*4`
`ADD sp, #immed*4 | SUB sp,`
`#immed*4`
`SXTH | SXTB | UXTH | UXTB`
`REV | REV16 | | REVSH`
`PUSH | POP`
`SETEND LE | SETEND BE`
`CPSIE | CPSID`
`BKPT immed8`
`STMIA | LDMIA Ln!,{register-list}`
`B<cond> instruction_address+`
`4+offset*2`
Undefined and expected to remain so
`SWI immed8`
`B instruction_address+4+offset*2`
`BLX ((instruction+4+`
`(poff<<12)+offset*4) &~ 3)`
This must be preceded by a branch prefix instruction.
This is the branch prefix instruction. It must be followed by a relative `BL` or `BLX` instruction.
`BL instruction+4+ (poff<<12)+`
`offset*2` This must be preceded by a branch prefix instruction.

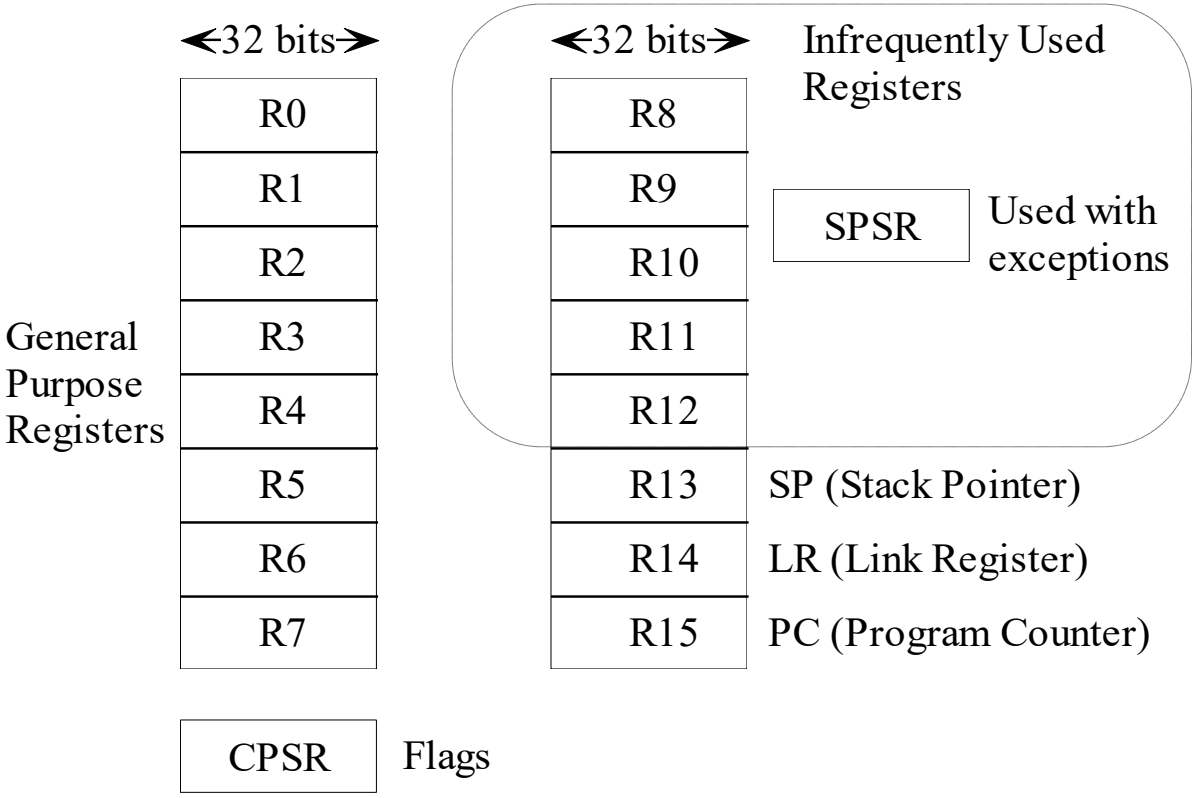
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	*op*	*Ld*			*immed8*							
1	0	1	0	*op*	*Ld*			*immed8*							
1	0	1	1	0	0	0	0	*op*	*immed7*						
1	0	1	1	0	0	1	0	*op*		*Lm*			*Ld*		
1	0	1	1	1	0	1	0	*op*		*Lm*			*Ld*		
1	0	1	1	*op*	1	0	*R*	*register_list*							
1	0	1	1	0	1	1	0	0	1	0	1	*op*	0	0	0
1	0	1	1	0	1	1	0	0	1	1	*op*	0	*a*	*i*	*f*
1	0	1	1	1	1	1	0	*immed8*							
1	1	0	0	*op*	*Ln*			*register_list*							
1	1	0	1	*cond* < 1110				signed 8-bit offset							
1	1	0	1	1	1	1	0	*x*							
1	1	0	1	1	1	1	1	*immed8*							
1	1	1	0	0	signed 11-bit *offset*										
1	1	1	0	1	unsigned 10-bit *offset*										0
1	1	1	1	0	signed 11-bit prefix offset *poff*										
1	1	1	1	1	unsigned 11-bit *offset*										

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 (Instr. length = 2 bytes)
  - 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 ← new branch address
      else PC ← 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 → “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)

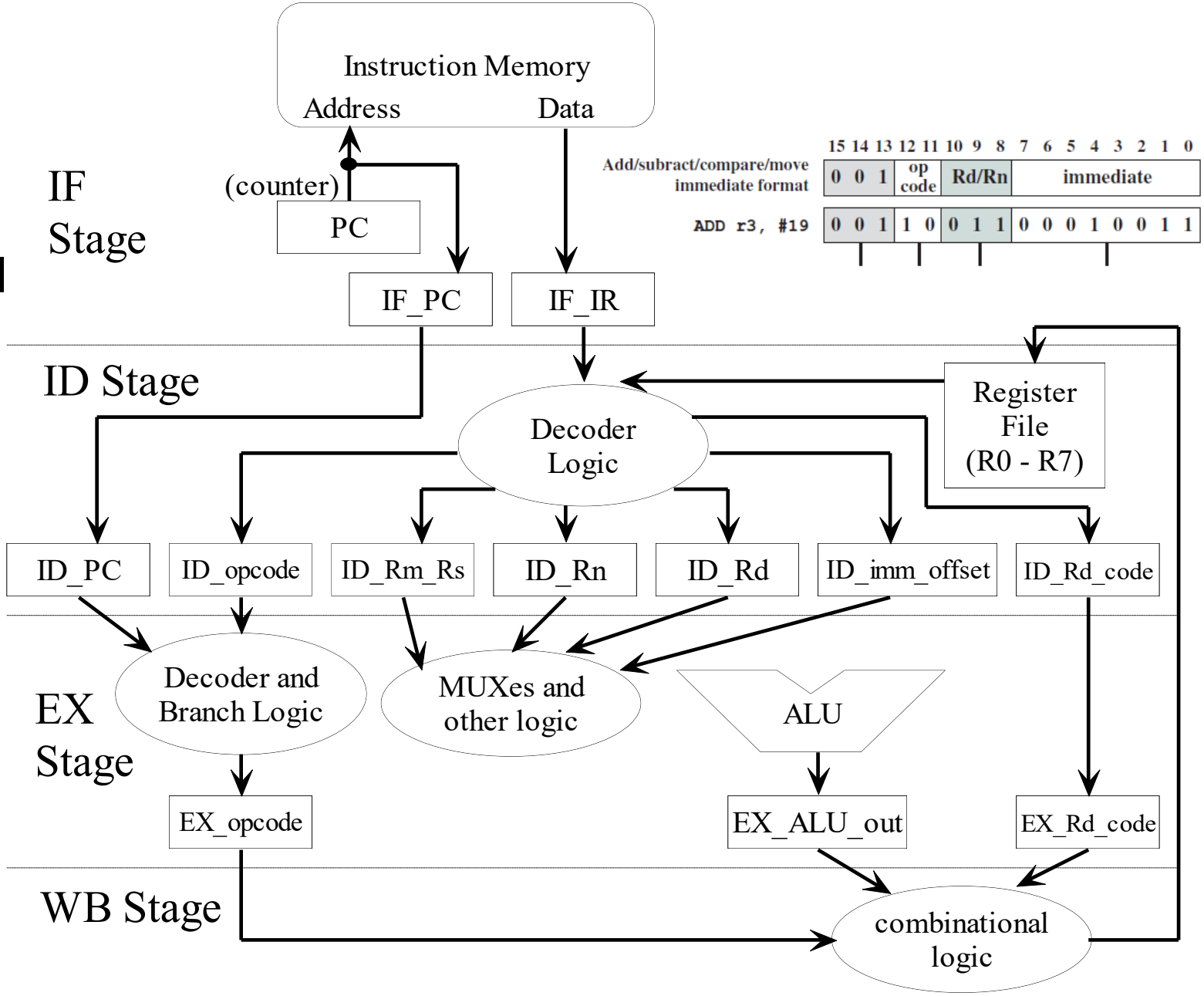
THUMB Instruction Set (4/4)

- 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

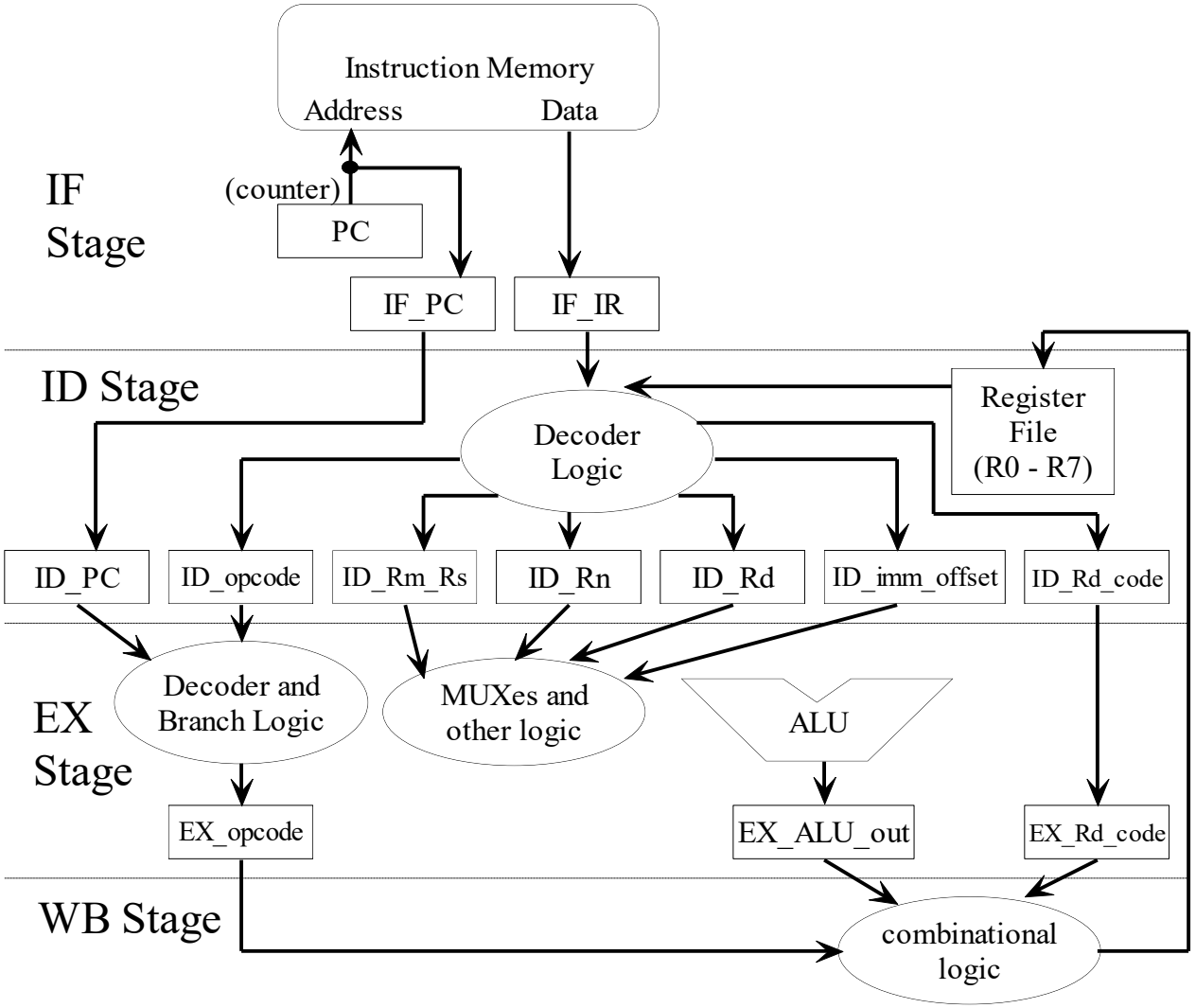


Verilog Codes for pipelined THUMB

- thumb_defs.vh
  - defines values for global constants
- thumb_syn.v
  - synthesizable circuit for THUMB
- tb_thumb.v
  - test bench for thumb.v
- 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_SIZE
// 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 Instruction Set Listings

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

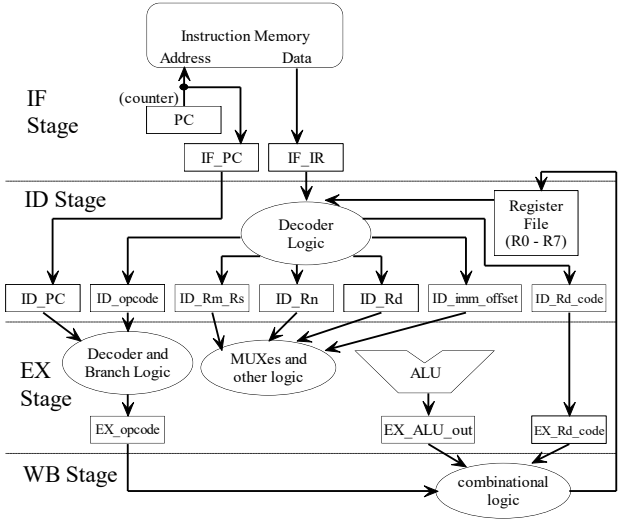
```
// ASSIGN STATEMENTS

 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

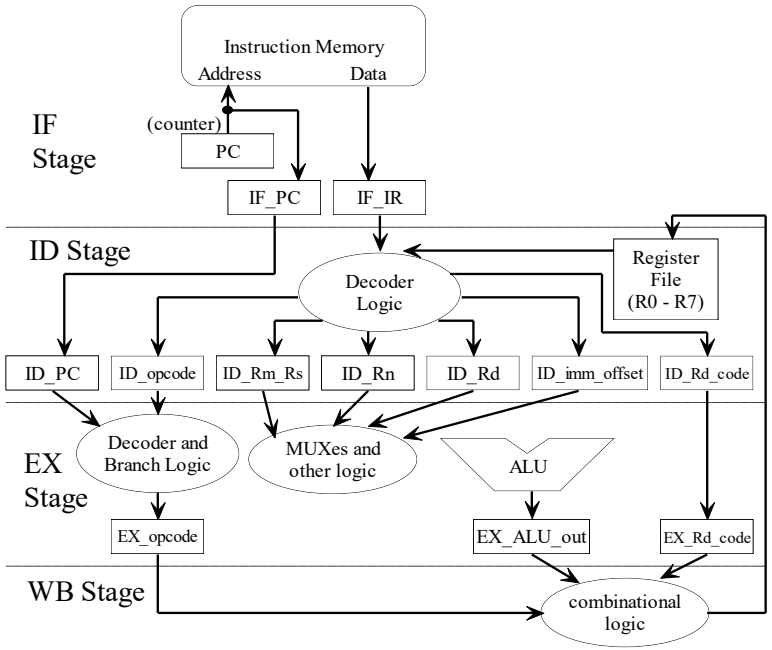
```
// Function to check condition codes for conditional branch instructions
 function condition_passed;
  input [3:0] cond_code;  // 4-bit Thumb/ARM condition code
 begin
  case (cond_code)  // codes in Table 3-1 of [ARM 2000]
    4'b0000: condition_passed = Z_Flag;  // EQ (equal)
    4'b0001: condition_passed = ~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)



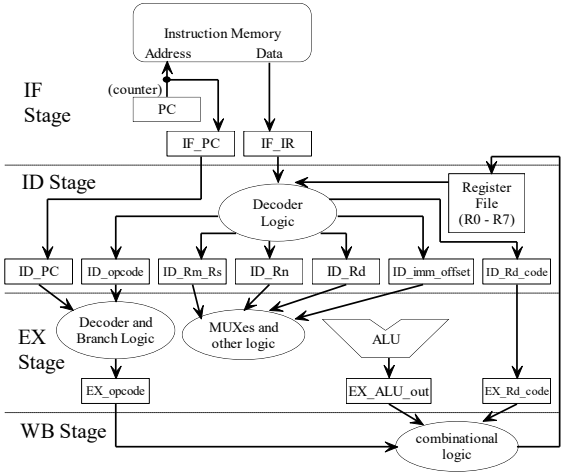
- read-enable signal (*read_instruction_n*) asserted and a valid address stored into PC
- during the next cycle, data returned from memory is latched into IF_IR pipeline register
  - when the instruction address of next instruction is being prepared (by storing the address into PC), the current instruction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

IF stage (2/2)



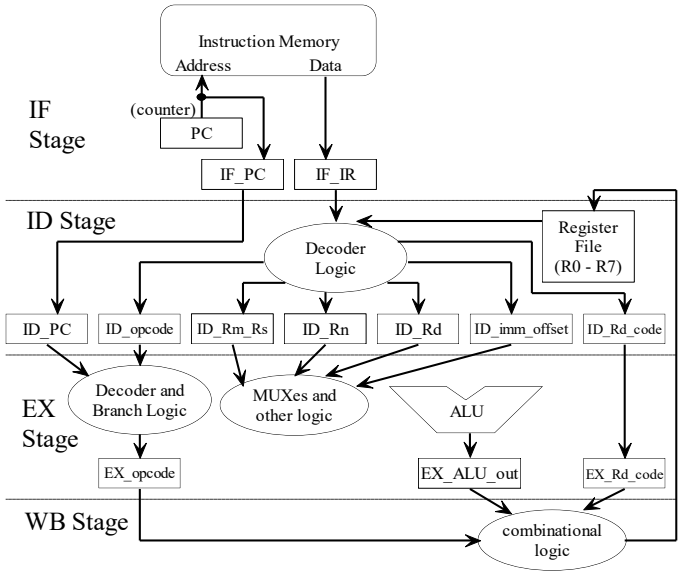
- when a branch operation (*branch_taken*) is detected
  - branch target is loaded into PC
  - the instruction fetched at this cycle (at PC+2) is incorrect
  - IR register filled with “undefined instruction”
    - ✓ pipeline stall or bubble

IF stage



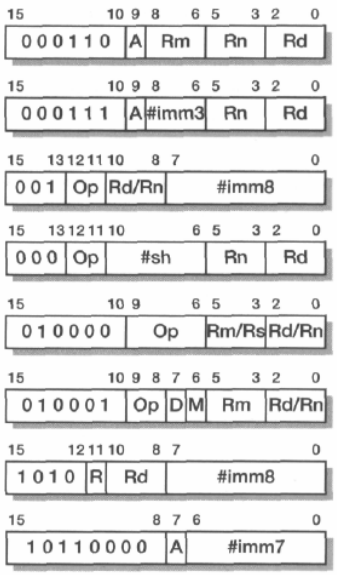

```
always @(negedge reset_n or posedge clk) begin
   if (~reset_n) begin
      PC <= 0;                          // start fetching from location 0
      read_instruction_n <= 0;       // and start memory read
   end
   else begin        // on positive clock edge,
      // execute operations and then 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 (given in testbench) and store in IR
         IF_PC <= PC + 2; end   // save next instruction address
  end
end  // of IF stage
```

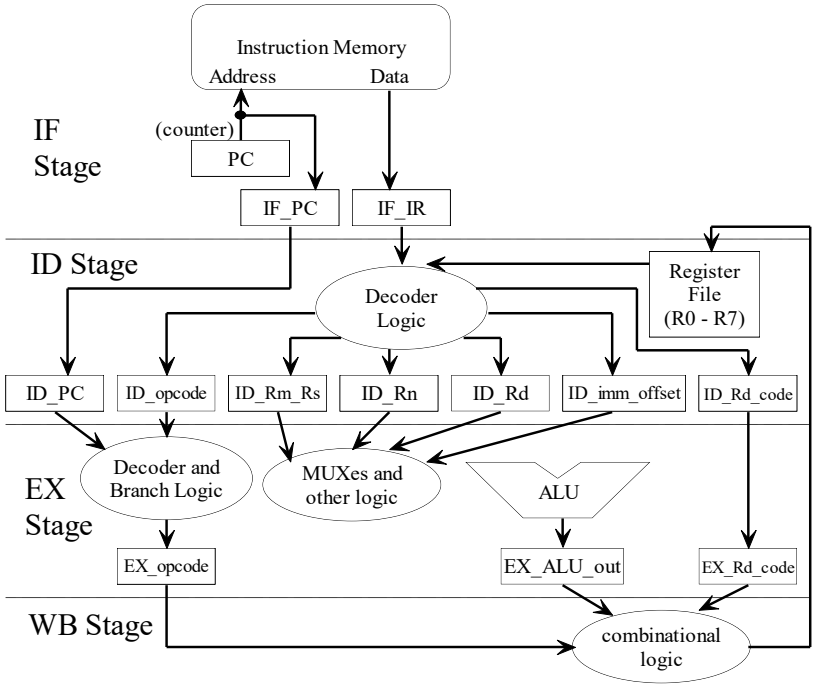
ID stage



- 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 from 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

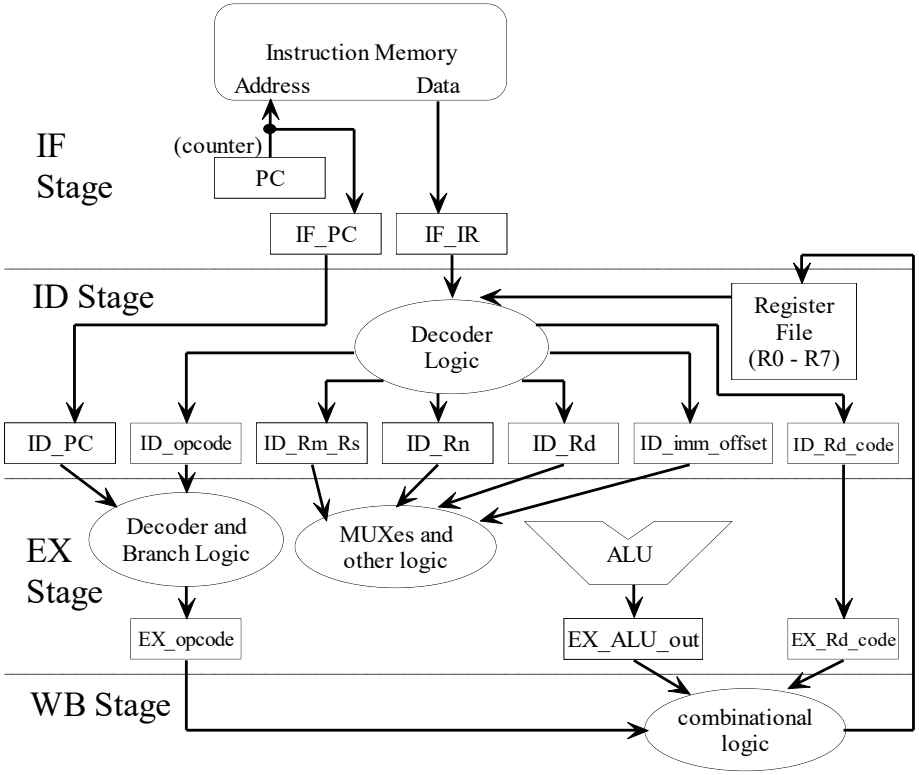





```
always @(posedge clk) begin
    case (IF_IR[`HWORD_SIZE-1:13])
    3'b000: begin // shift by immediate or add/sub
            case (IF_IR[12:11])
            2'b11: begin
                    case (IF_IR[10:9])
                    2'b10: begin // if imm = 000, same as MOV_2
                            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]];   // content of register Rd
                            ID_Rd_code <= IF_IR[2:0];  // index of register Rd
                            end
```

...

EX stage

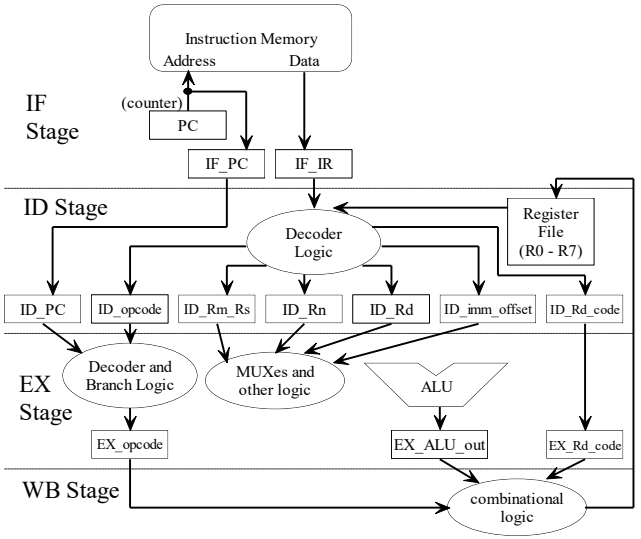


- a set of **case** blocks, simpler than ID stage
- read and write control signals for data memory are initialized at the beginning
- some instructions (such as MUL) might requires more execution time than others

EX stage

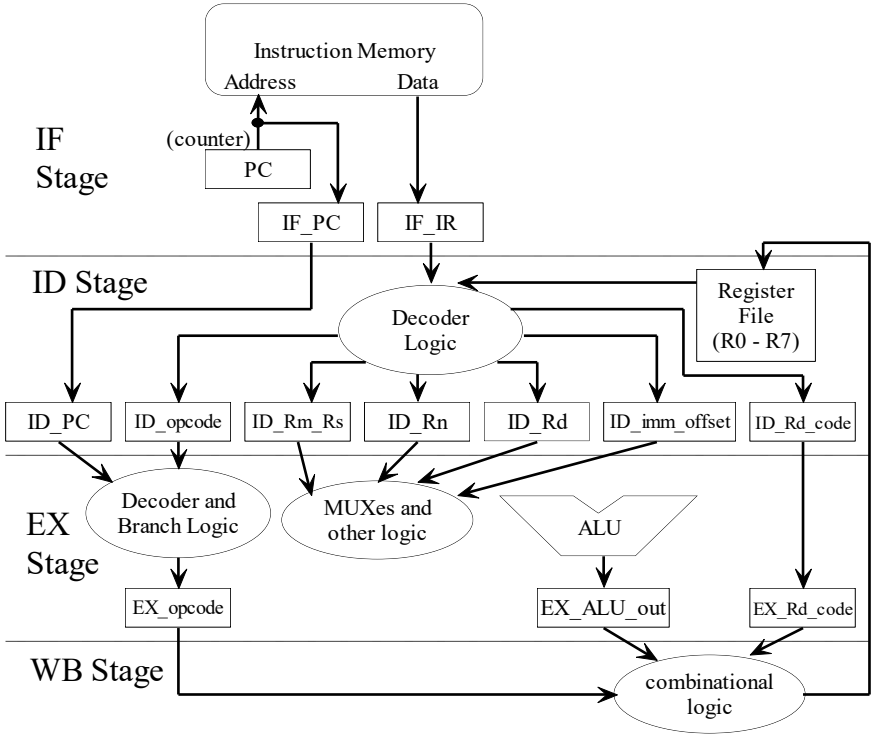
```
always @(negedge reset_n or posedge clk) begin
   if (~reset_n) begin
      branch_taken <= 1'b0;  // initialize to branch not taken
      read_data_n <= 1'b1;   // disable data memory
      write_data_n <= 1'b1;
   end
   else begin  // on positive clock edge
      read_data_n <= 1'b1;   // set 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 lst 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
      …
```



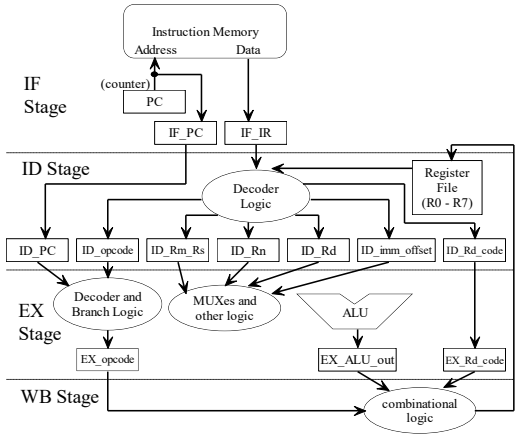
WB stage

- 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*

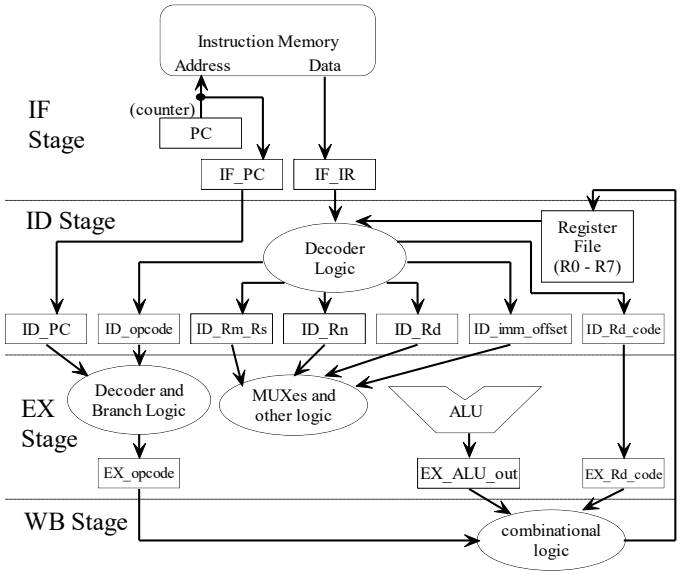


WB stage

```
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 register through output data line

test bench 1/7 (chip instantiation)

test bench

```
module tb_thumb();
…
// SIGNAL DECLARATIONS for hardware chip inputs and outputs
  wire read_instruction_n;                          // control read from instruction mem
  wire [`WORD_SIZE-1:0] instruction_address; // address of instruction, `WORD_SIZE=32
  wire [`HWORD_SIZE-1:0] instruction;           // current instruction, `HWORD_SIZE=16
  wire read_data_n;                                 // control read from data memory
  wire write_data_n;                                // control write to data memory
  wire [`WORD_SIZE-1:0] data_address;         // address of data
  wire [`WORD_SIZE-1:0] data;                       / current data
  reg reset_n;                                      // active-low RESET signal
  reg clk;                                          // clock signal

 // SIGNAL DECLARATIONS for signals being used internally
  reg [`WORD_SIZE-1:0] output_instruction; // output of instruction memory
  reg [`WORD_SIZE-1:0] output_data;          // output of data memory
  reg [`WORD_SIZE-1:0] write_data;         // data to be written

 // instantiate the unit under test
  thumb UUT (
                read_instruction_n,     // chip output to control instruction memory read
                instruction_address,  // chip output for address of instruction memory read
                instruction,               // chip input from instruction memory read
                read_data_n,             // chip output to control data memory read
                write_data_n,             // chip output to control data memory write
                data_address,           // chip output for address of data memory read/write
                data,                        // chip input from data memory read, or chip output to data memory write
                reset_n,                  // chip input to reset all storage elements in hardware
                clk);    )                // chip input for global clock
…
```

test bench 2/7 (memory declaration)

test bench

…

```
// initialize inputs
 initial begin
   clk = 0;          // set initial clock value

   reset_n = 1;      // generate a LOW pulse for reset_n as input to chip
   #(`PERIOD1/4) reset_n = 0;      // `PERIOD1=100
   #(`PERIOD1 * 2) reset_n = 1;
 end

 // generate the clock as input to chip
 always #(`PERIOD1/2) clk = ~clk; // period = `PERIOD1=100

// model the instruction and data memory devices
 reg [`HWORD_SIZE-1:0] instruction_memory [0:`MEMORY_SIZE-1];  // `MEMORY_SIZE=256
 reg [`WORD_SIZE-1:0] data_memory [0:`MEMORY_SIZE-1];
```

test bench 3/7 (instr. read)

test bench

```
…
 // model the read process for the instruction memory device

// generate instruction as chip input during instruction memory read
  assign instruction = read_instruction_n ? 'bz : output_instruction;

// generate output_instruction from instruction memory
  always
  begin
    if (read_instruction_n == 0) begin
      #`READ_DELAY;  // assume no spurious address changes, `READ_DELAY=80
      output_instruction = instruction_memory[ instruction_address[7:0] ];
      wait (  (read_instruction_n == 1) ||
                (output_instruction != instruction_memory[instruction_address[7:0]])
              );
    end
    else begin  // end of read
      #`STABLE_TIME;   // `STABLE_TIME=10
      output_instruction = `WORD_SIZE'bz;   // output_instruction is floating when read_instruction_n = 1
      wait (read_instruction_n == 0);    // wait for another instruction read
    end
  end
```

test bench 4/7 (data read)

test bench

```
…
 // model the read process for the data memory device

// generate data as chip input during data memory read
  assign data = read_data_n ? `WORD_SIZE'bz : output_data;

// generate output_data from data memory read
  always
  begin
    if (read_data_n == 0) begin
      #`READ_DELAY;  // assume no spurious address changes
      output_data = data_memory[data_address[7:0]];
      wait ((read_data_n == 1) ||
            (output_data != data_memory[data_address[7:0]]));
    end
    else begin  // end of read
      #`STABLE_TIME;
      output_data = `WORD_SIZE'bz;  // output_data is floating when read_data_n = 1
      wait (read_data_n == 0);   // wait for another data read
    end
  end
```

test bench 5/7 (data write) test bench

…

```
// model the write process for the data memory device
// write_data_n is set to 0 during the EX stage for store instructions
// data is written to data_memory at the beginning of the next cycle where write_data_n is set to 1
  always
  begin
    wait (write_data_n == 0); // write_data_n is set to 0 during the EX stage for store instructions
    write_data = data;   // chip output data is assigned to write_data  when write_data_n = 0
    wait ((write_data_n == 1) || (data != write_data));
    if (write_data_n == 1) begin  // wait for write enable = '1'
      #`WRITE_DELAY;   //  `WRITE_DELAY=80
      data_memory[data_address[7:0]] = write_data;   // written to data memory when write_data_n = 1
                                                     // write_data_n is set to 1 at the beginning of EX stage
    end
    else  // data != write_data (data has changed)
      write_data = data;   // new chip output data is assigned to write_data
  end
```

test bench 6/7 (assembly instructions)

…

```
 // store programs and data in the instruction and memories

// R[2] accumulates 0+1+2+3+…+8 = 36 = 0010_0100,
// and then store to data memory at address R[0]+4 = ‘b 1111_1100 + ‘b 0000_0100 = ‘b 1_0000_0000
  initial
  begin
    instruction_memory[0] = 16'h2100; // MOV r1, #0          // R[1] = 0
    instruction_memory[2] = 16'h2200; // MOV r2, #0          // R[2] = 0
    instruction_memory[4] = 16'h20fc; // MOV r0, #fc         // R[0] = fc
    instruction_memory[6] = 16'h2909; // CMP r1, #9          // set flags N, V, C, Z based on value of R[1]-9
    instruction_memory[8] = 16'hda04; // BGE 0x14            // if R[1] >= 9, goto 20 which store to memory
    instruction_memory[10] = 16'he001;// B   0x10            // if R[1] < 9, goto 0x10 = 16 which is R[2]=R[2]+R[1]
    instruction_memory[12] = 16'h3101;// ADD r1, #1          // R[1] = R[1] + 1
    instruction_memory[14] = 16'he7fa;// B   0x6             // goto 0x6 = 6 which compares R[1] and 9
    instruction_memory[16] = 16'h1852;// ADD r2, r2, r1      // R[2] = R[2] + R[1]
    instruction_memory[18] = 16'he7fb;// B   0xc             // goto 0xc = 12 which increment R[1]
    instruction_memory[20] = 16'h6042;// STR R2,R0+1*4(data:0x24)   // Mem[R[0]+4] = R[2] = 0010_0100=36
    instruction_memory[22] = 16'hdf00;// SWI                 // software interrupt to OS, halt the program
    // instruction_memory[22] = 16'h4770;// BX  r14          // branch exchange, return to main (goto R[14])
    // last instruction in original assembly is a return to main
  end
```

test bench 7/7 (verification)

…

```
// test output of program to verify proper operation of circuit

initial
begin
  #15700; // this is the time when the program output is avail.
  if ( (data_address === 32'h00000100) && (data === 32'h00000024) )
    $display("Data address (0x100) and data (0x24) correct.");
  else
    $display("ERROR: data address = %0x and data = %0x.", data_address, data);
  #500;  $display("Simulation completed at time %0t.", $time);
  $finish;  // terminate simulation after 16.2 microseconds
end
```

Summary of test bench

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 for instruction memory read
assign instruction = read_instruction_n ? 'bz : output_instruction;
…
// model for data memory read
assign data = read_data_n ? `WORD_SIZE'bz : output_data;   // output_data is assigned to chip input data
always begin
if (read_data_n == 0)  output_data = data_memory[data_address[7:0]);
…
// model  for data memory write
write_data = data;   // chip output data is assigned to write_data when write_data_n = 0
if (write_data_n == 1) data_memory[address[7:0]) = write_data;   // written to data memory when write_data_n=1
…
// store programs and data in the instruction and memory
initial begin
Instruction_memory[0] = 16'h2100;  // MOV R1, #0  (R[1] <- 0 )
Instruction_memory[2] = 16'h2200;  // MOV R2, #0  (R[2] <- 0 )
…
// test output of program for verification
...
endmodule
```