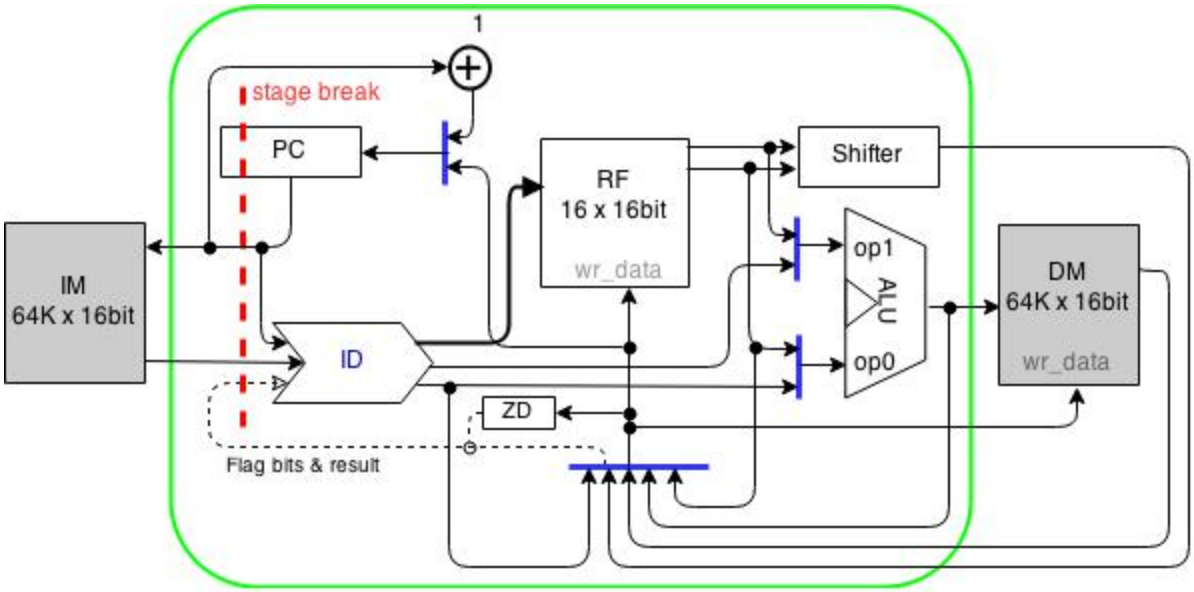
EE477 Project: 2-stage Pipelined CPU

1. Core Organization



By the end of this course, you will build a complete simplified 16-bit ARM based CPU. The 2-stage pipelined CPU will breakdown the stages between the instruction fetch and instruction decode as shown with the red dotted line in the diagram.

To help you understand the core organization, below is the detail of each component:

1.1. MUX

Each blue colored bar line in the diagram represents a multi-input selector (MUX), in which each selector operates based on the select control signals determined by the Instruction Decoder.

1.2. Program Counter (PC)

This is a 16-bit program counter that keeps track of where the position of current instruction is at. It updates its PC value at the rising edge of each clock cycle. The PC can either increment by 1 or branch to a different PC value based on the MUX output.

- Inputs: next PC value
- Outputs: current PC value
- Size: 16 bits

1.3. Instruction Memory (IM)

This memory stores all binary instruction sets. It provides 64K address lines, each address line is 16 bits. The IM receives the current program count (corresponding to the instruction address position) from the PC, then outputs a 16-bit instruction. This operation is called "instruction fetch," and this is the end of the 1st stage in the pipelining.

** The IM will be modeled in Verilog to be integrated into the whole design.

- ➢ Inputs: 16-bit address value
- ➢ Outputs: 16-bit binary instruction
- ➢ Size: 64Kx16bit

1.4. Instruction Decoder (ID)

This is the key control module in the CPU, which interprets the binary instruction, determines all control signals required in the datapath, and outputs appropriate operands for further data processing. This operation is called "instruction decode," and this is the beginning of the 2nd stage in the pipelining.

- ➢ Inputs: 16-bit instruction, 16-bit PC value, 4-bit flags, 16-bit Link Register value, 16-bit Stack Pointer value
- ➢ Outputs:
  - ○ regfile: read_addr0, read_addr1, write_en, wr_addr
  - ○ shifter: operation control bits
  - ○ ALU: operand1, operand0, operation control bits
  - ○ data mem: read_en, write_en
  - ○ all MUXs control bits
- ➢ Additional functions: Zero extend immediate value to 16-bit, sign extend immediate value to 16-bit

1.5. Register File (RF)

Register file contains sixteen 16-bit registers (r0 ~ r15). r0 ~ r12 are general purpose registers, r13 ~ r15 are reserved registers representing Link Register, Stack Pointer Register and Program Status Register. To utilize your mini project design, since the access to r13~r15 for read/write is not available from the regfile, please use two stand alone 16-bit registers for LR and SP for this CPU project.

- ➢ Inputs: read_addr0, read_addr1, write_en, write_addr, write_data
- ➢ Outputs: read_data0, read_data1

1.6. Shifter

The shifter unit supports bit manipulation of logical shift in either direction, arithmetic right shift and right rotate.

- ➢ Inputs: operation control such as shift, right and arith, 16-bit data, 5-bit shift amount
- ➢ Outputs: 16-bit result, carry_flag, negative_flag

1.7. ALU

The arithmetic logic unit supports add, subtract, invert, and, or, xor operations with the two 16-bit inputs. The two inputs are either received from the RF or from the ID.

- ➢ Inputs: 16-bit operand1, 16-bit operand0, operation control bits
- ➢ Outputs: 16-bit result, carry_flag, negative_flag, overflow_flag

1.8. Zero Detector (ZD)

This component simply detects whether or not the final result is zero. It output a "1" if the data is zero and outputs an "0" if the data is not zero.

- ➢ Inputs: 16-bit result
- ➢ Outputs: zero_flag

1.9. Data Memory (DM)

This is the main memory for your processor which offers a data storage capacity of 16-bit entries and each entry is 2 bytes long. If read or write request is received, the data memory will accept the 16-bit address value from the ALU.

** The DM will be modeled in Verilog to be integrated into the whole design.

- ➢ Inputs: input_addr (read or write address input shares the same port), read_en, write_en, write_data
- ➢ Outputs: read_data
- ➢ Size: 64K x 16bit

1.10. Stage Registers (not shown in the diagram)

Additional registers are needed in order to safely store necessary data from the 1st stage and pass it onto the 2nd-stage. These important stage registers are listed below:

- ❖ Instruction Register (IR)

Stores the 16-bit instruction fetched from the Instruction Memory

- Flag Register (for N, Z, C, V flags)

The program status register is simplified down to a 4-bit register that uses 1 bit per each flags. *Please follow the bit index arrangement in the Flags section in your design.*

- Stack Pointer (SP, alias r13)

Stack Pointer acts as a pointer to the active stack in DM. It starts from the bottom of the memory which is address 16'hffff, then moves up one by one as more subroutine calls are made. The SP register will always store: the latest data memory address the pointer is at - 1. The DM address[SPreg + 1] will stores the latest return link PC value if any.

- Link Register (LR, alias r14)

Link Register stores the Return Link (return PC value). This is a value that relates to the return address from a subroutine that is entered using a Branch with Link instruction.

2. Control Logic

Control logic is implemented within the Instruction Decoder. This module takes in a fresh instruction every cycle and generates signals to control the operation of the datapath. Prominent examples are:

- RegFile: write_en
- ALU: select operand0 and operand1 inputs from either the ID or from the RF
- Shifter: operation control bits such as shift, right, arith
- DataMem: read_en and write_en
- PC: the MUX control bit which determines whether or not to branch
- Flags: update flags from the flag outputs of shifter, ALU or previous flags from ID
- Select final results from RF, ALU, DM, shifter or ID

3. Supported Instructions[1]

As mentioned earlier, this 2-stage pipelined CPU is implemented based on the ARM ISA. This section will provide the supported instructions in detail with reference to the arm v6m manual.

3.1. Flags

[1] Source: DDI0419C_arm_architecture_v6m_reference_manual

The required four flags in the CPU architecture are N, Z, C, V. Please see the table below for detail.

Flag Register bit index	Flag Symbol	Flag Name	Detail (from ARM Manual)
3	N	Negative	Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then N is set to 1 if the result is negative and set to 0 if it is positive or zero.
2	Z	Zero	Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.
1	C	Carry	Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.
0	V	Overflow	Overflow condition code flag.Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

3.2. Additional Branch Conditions

B<cc>	Condition bits	Flag Condition	Detail
EQ	0000	z==1	equal
NE	0001	z==0	not equal
CS	0010	c==1	carry set
CC	0011	c==0	carry clear
MI	0100	n==1	minus, negative
PL	0101	n==0	plus, positive or zero
VS	0110	v==1	overflow
VC	0111	v==0	no overflow
HI	1000	c==1 and z==0	unsigned higher
LS	1001	c==0 or z==1	unsigned lower or same
GE	1010	n==v	signed greater than or equal
LT	1011	n!=v	signed less than
GT	1100	z==0 and n==v	signed greater than
LE	1101	z==1 or n!=v	signed less than or equal
AL	1110	Any	Always (unconditional)

3.3. Instructions

Attached in the next page is a table of all assembly instructions expected to support in the CPU design.

16-bit Instruction																Assembler	Operation	S updates	Action
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	0	1	0	0	Rd			imm8								MOVS Rd, #<imm>	Move	N Z	Rd := imm8, ZeroExt(<imm8>, 16)
0	1	0	0	0	1	1	0	0	Rm				Rd			MOV Rd, Rm			Rd = Rm
0	0	0	1	1	1	0	imm3		Rn				Rd			ADDS Rd, Rn, #<imm3>	Add	N Z C V	Rd := Rn + ZeroExt(<imm3>, 16)
0	0	0	1	1	0	0	Rm		Rn				Rd			ADDS Rd, Rn, Rm		N Z C V	Rd := Rn + Rm
1	0	1	1	0	0	0	0	0	imm7							ADD SP, SP, #<imm7>			SP := SP + ZeroExt(<imm7>, 16)
0	0	0	1	1	0	1	Rm		Rn				Rd			SUBS Rd, Rn, Rm	Subtract	N Z C V	Rd := Rn + ~(Rm) + 1
0	0	0	1	1	1	1	imm3		Rn				Rd			SUBS Rd, Rn, #<imm3>		N Z C V	Rd := Rn + ~(imm3) + 1
1	0	1	1	0	0	0	0	1	imm7							SUB SP, SP, #<imm7>			SP := SP + ~(ZeroExt(<imm7>, 16)) + 1
0	1	0	0	0	0	1	0	1	0	Rm			Rn			CMP Rn, Rm	Compare	N Z C V	Result := Rn + ~(Rm) + 1; Result is not used
0	1	0	0	0	0	0	0	0	0	Rm			Rdn			ANDS Rdn, Rm	Logical	N Z	Rd := Rn AND Rm
0	1	0	0	0	0	0	0	0	1	Rm			Rdn			EORS Rdn, Rm		N Z	Rd := Rd XOR Rm
0	1	0	0	0	0	1	1	0	0	Rm			Rdn			ORRS Rdn, Rm		N Z	Rd := Rd OR Rm
0	1	0	0	0	0	1	1	1	1	Rm			Rdn			MVNS Rdn, Rm		N Z	Rd := ~(Rm)
0	1	0	0	0	0	0	0	1	0	Rm			Rdn			LSLS Rd, Rd, Rm	Shift	N Z C	Rd := Rd << Rm
0	1	0	0	0	0	0	0	1	1	Rm			Rdn			LSRS Rd, Rd, Rm		N Z C	Rd := Rd >> Rm
0	1	0	0	0	0	0	1	0	0	Rm			Rdn			ASRS Rd, Rd, Rm		N Z C	Rd := Rd >>> Rm
0	1	0	0	0	0	0	1	1	1	Rm			Rdn			RORS Rd, Rd, Rm	Rotate	N Z C	Right shift Rm amount of bits, the shifted out bits are inserted into the vacated bits on the left
0	1	1	0	0	imm5					Rn			Rd			STR Rd, [Rn, #<imm5>]	Store		Mem[Rn + ZeroExtend(imm5, 16), 2] = Rd
0	1	1	0	1	imm5					Rn			Rd			LDR Rd, [Rn, #<imm5>]	Load		Rd = Mem[Rn + ZeroExtend(imm5, 16), 2]
1	1	0	1	cond				imm8								B<cc> <label>	Branch		If the condition is satisfied, PC = PC + SignExt(imm8, 16)
1	1	1	0	0	imm11											B <label>			PC = PC + SignExt(imm11, 16)
0	1	0	0	0	1	0	1	0	0	imm6						BL <label>			LR = PC+1; PC = PC + SignExt(imm6, 16)
0	1	0	0	0	1	1	1	0	Rm				0	0	0	BX Rm			PC = Rm
1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	NOOP	No Operation		Execution stalls for one cycles