Lecture 3

Single-Cycle Implementation

We study how to implement a hardware for Y86 instruction set architecture[BO10]. We first study a simple implementation technique i.e. *single-cycle implementation*. In this implementation, *All types of instructions are executed in one cycle.*

3.1 Datapath

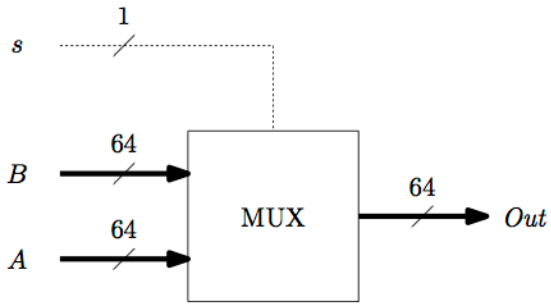
Datapath is the processor components performing arithmetic or logical operations. Executing an instruction is equivalent to controlling the flow of data from one place to another place through various components.

For example, `add %rcx, %rax` is performed by obtaining the values of `%rax` and `%rcx`, feeding both values to the ALU, conducting an addition, and writing the result to `%rax`.

3.2 Basic Hardware Components

Multiplexer or *MUX* is a device for choosing a value from a set of values. It is used to select an input based on a control signal.

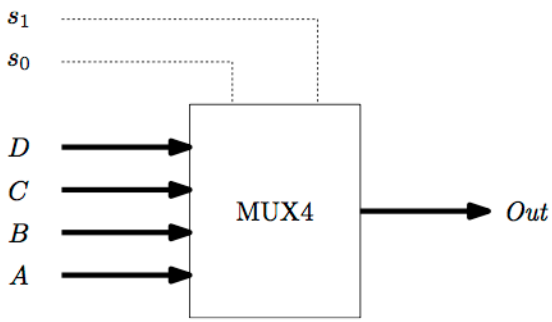
2-to-1 Multiplexer



where A and B are values or data signals to be selected, s is one-bit control signal.

$$out = \begin{cases} A & \text{if } s = 0 \\ B & \text{if } s = 1 \end{cases}$$

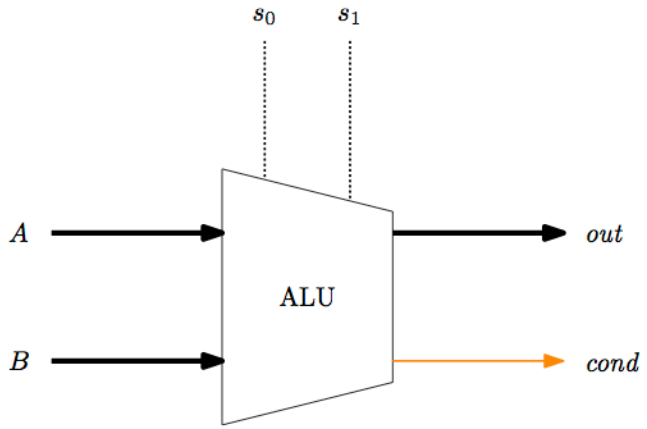
4-to-1 Multiplexer



where A, B, C and D are values or data signals to be selected, and s_0, s_1 are control signals.

$$out = \begin{cases} A & \text{if } s_1 = 0, s_0 = 0 \\ B & \text{if } s_1 = 0, s_0 = 1 \\ C & \text{if } s_1 = 1, s_0 = 0 \\ D & \text{if } s_1 = 1, s_0 = 1 \end{cases}$$

ALU is a digital circuit for conducting arithmetic/logical operations.



where A and B are data signals, s_0 and s_1 are control signal to select an operation (**add**, **sub**, **and**, **xor**).

$$out = \begin{cases} A + B & \text{if } s_1 = 0, s_0 = 0 \\ A - B & \text{if } s_1 = 0, s_0 = 1 \\ A \,\&\, B & \text{if } s_1 = 1, s_0 = 0 \\ A \hat{} B & \text{if } s_1 = 1, s_0 = 1 \end{cases}$$

cond is a 3-bit data signal composing of *CF* (carry flag), *ZF* (zero flag), and *SF* (sign flag, or negative flag). The values of these flags depend on the result of the operation, or the value of *out*.

```
cmp   $10, %rax     # Calculate %rax-10, set cond (ZF:SF:CF)
je    L1            # Jump to L1 if ZF==1
```

$$ZF = \begin{cases} 1 & \text{if } out = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$SF = \begin{cases} 1 & \text{if } out < 0 \\ 0 & \text{otherwise} \end{cases}$$

Note: when the most significant bit of *out* is 1, we have $out < 0$ according to the two's complement technique.

CF (carry flag) indicates whether a *carry* is generated out of the most significant bit (the leftmost bit). For example, `0xffffffff` + `0x00010001` yields '1' out of the 32-bit storage. This is the value of the carry flag. Therefore, $CF = 1$.

```
0xffffffff =  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0x00010001 =  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 +
             ---------------------------------------------------------------
            1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The carry flag is set after conducting an addition or a subtraction.

Exercise 3.1 Write the values of *out* and *cond* for the following operations.

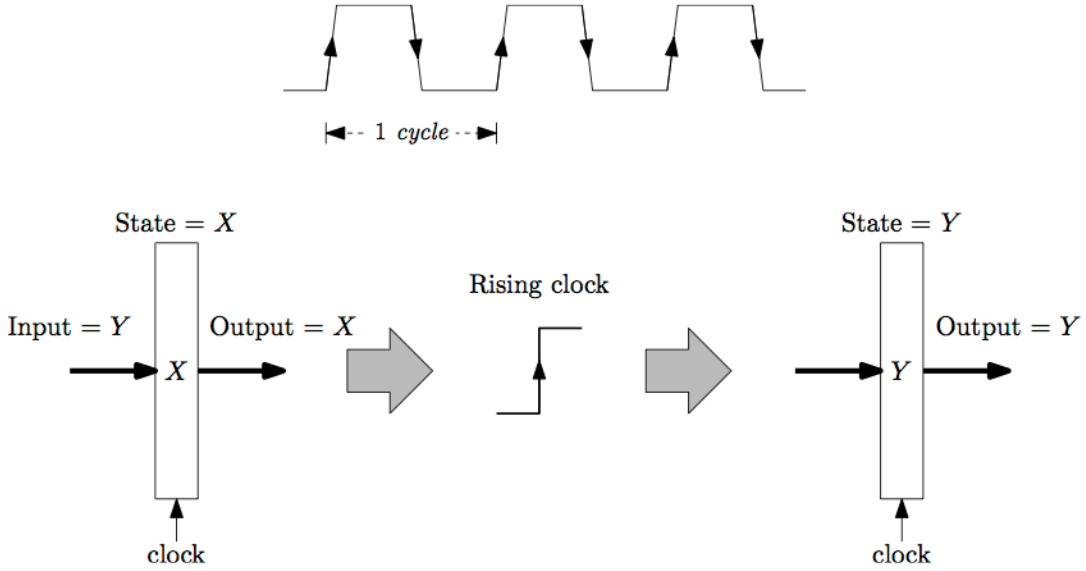
1. `0x70010001 + 0x20`

2. `0xfffffffe + 0x2`

3. `0x10101010 & 0xffff0000`

4. `0x10101010 ^ 0xffff0000`

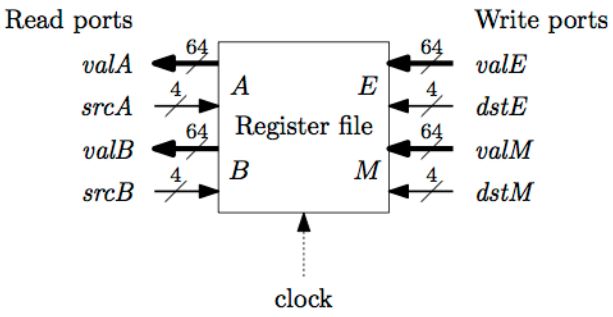
Register and Memory are hardware components capable for storing values. They are *synchronous circuits* with a global clock signal.



Register file stores a number of values accessible by 4-bit integers. Due to the requirement of the Y86 ISA, the register file needs to have 2 channels for reading values, and 2 channels for writing values.

When a register number (a 4-bit integer) is fed to *srcA*, the value of the register can be obtained from *valA* after a short period of time. This is the same for *srcB* and *valB*.

To write a value to a register, the register number and the value to be written need to be provided. However, the state of the register will be changed after being triggered by a rising clock signal. There are two channels to write values i.e. *dstE* – *valE*, and *dstM* – *valM*.



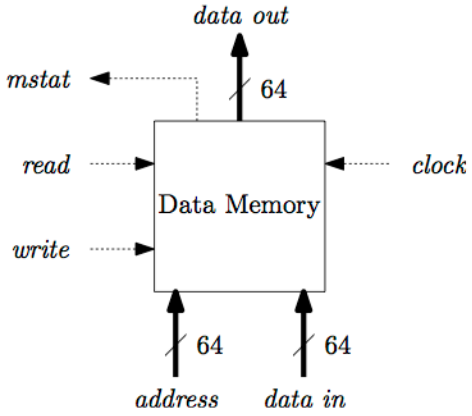
Name	Code	Name	Code
`%rax`	`0`	`%r8`	`8`
`%rcx`	`1`	`%r9`	`9`
`%rdx`	`2`	`%r10`	`A`
`%rbx`	`3`	`%r11`	`B`
`%rsp`	`4`	`%r12`	`C`
`%rbp`	`5`	`%r13`	`D`
`%rsi`	`6`	`%r14`	`E`
`%rdi`	`7`	No register	`F`

Memory is similar to the registers. It can store larger amount of data. There are two control signals for the memory unit i.e. *read* and *write*.

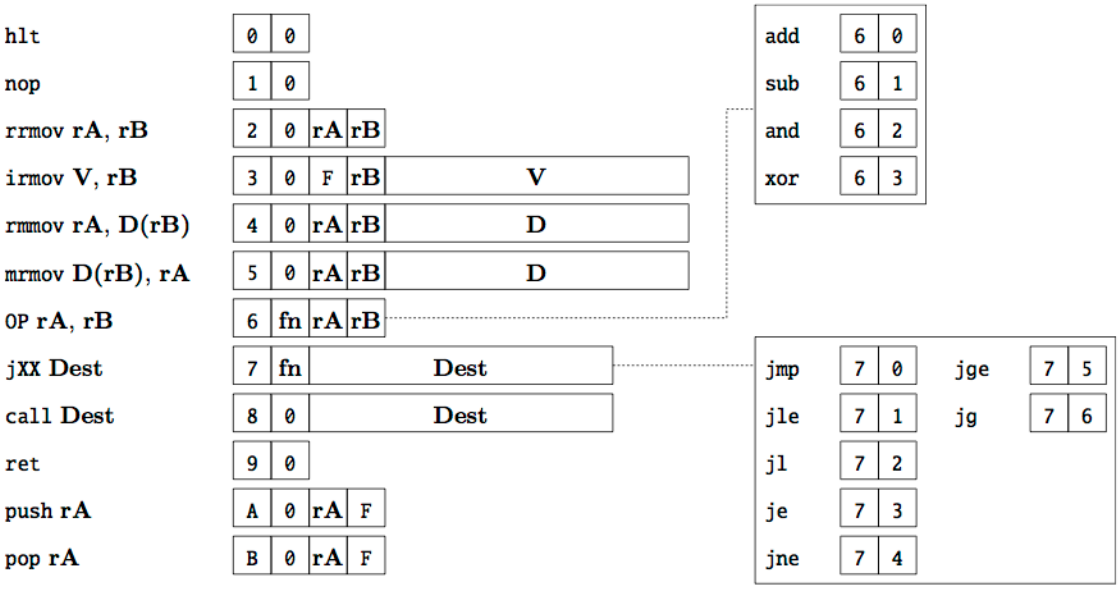
When $read = 1, write = 0$, the *data out* will become the value of the data at the *address*.

When $read = 0, write = 1$, the *data in* will written at the *address* after being triggered by a rising clock signal.

The basic unit of data for the *data memory* is set to 4 bytes.



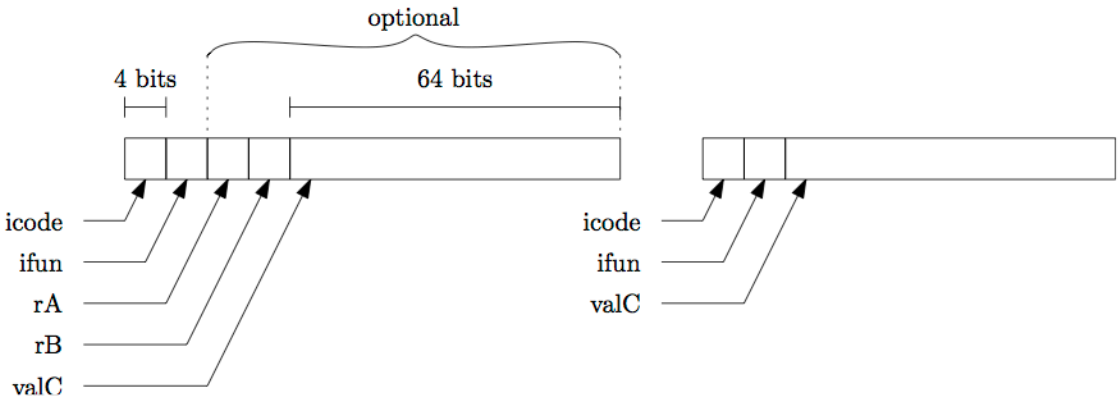
3.3 Instructions of Y86



Registers

0	1	2	3	4	5	6	7
%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
8	9	A	B	C	D	E	F
%r8	%r9	%r10	%r11	%r12	%r13	%r14	No

3.4 Y86 Instruction Format



3.5 Single-Cycle Implementation of Y86

3.5.1 Stages

1. **Fetch**

   Read the bytes of instruction from the instruction memory at the address stored in the program counter (PC). The read bytes are split into fields i.e. *icode*, *ifun*, *rA*, *rB*, and *valC*.

2. **Decode**

   Read up to two operands from the registers by feeding *rA* and/or *rB* to *srcA* and/or *srcB*. This yields *valA* and *valB*.

3. **Execute**

   Perform the operation by the ALU. The resulting value is referred as *valE*. The flags from the calculation is stored in *cond*.

4. **Memory**

   Write the resulting value to the memory, or read data from the memory and refer to it as *valM*.

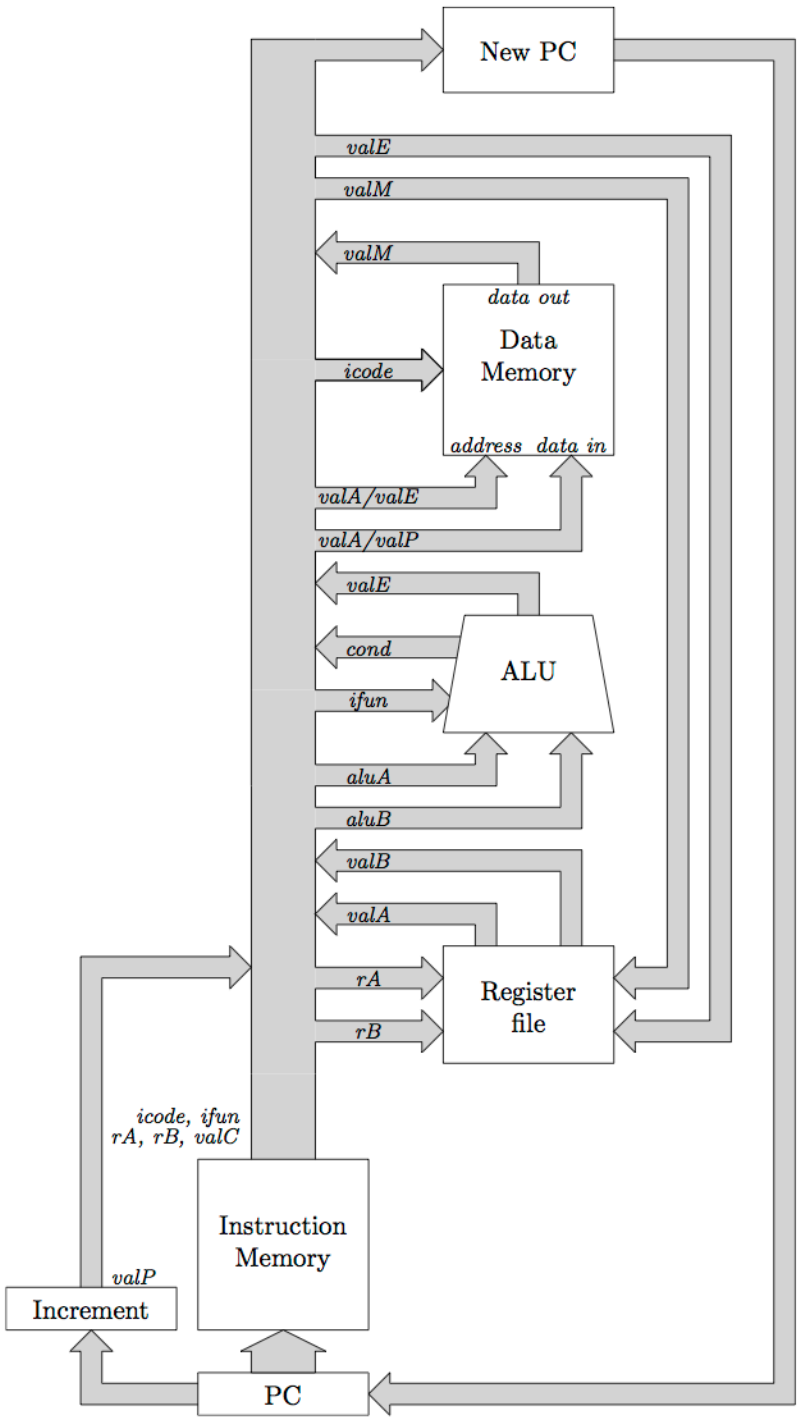
5. **Write back**

   Write the resulting value to the register file. Two values can be written to two different registers by specifying *dstE* and *dstM*.

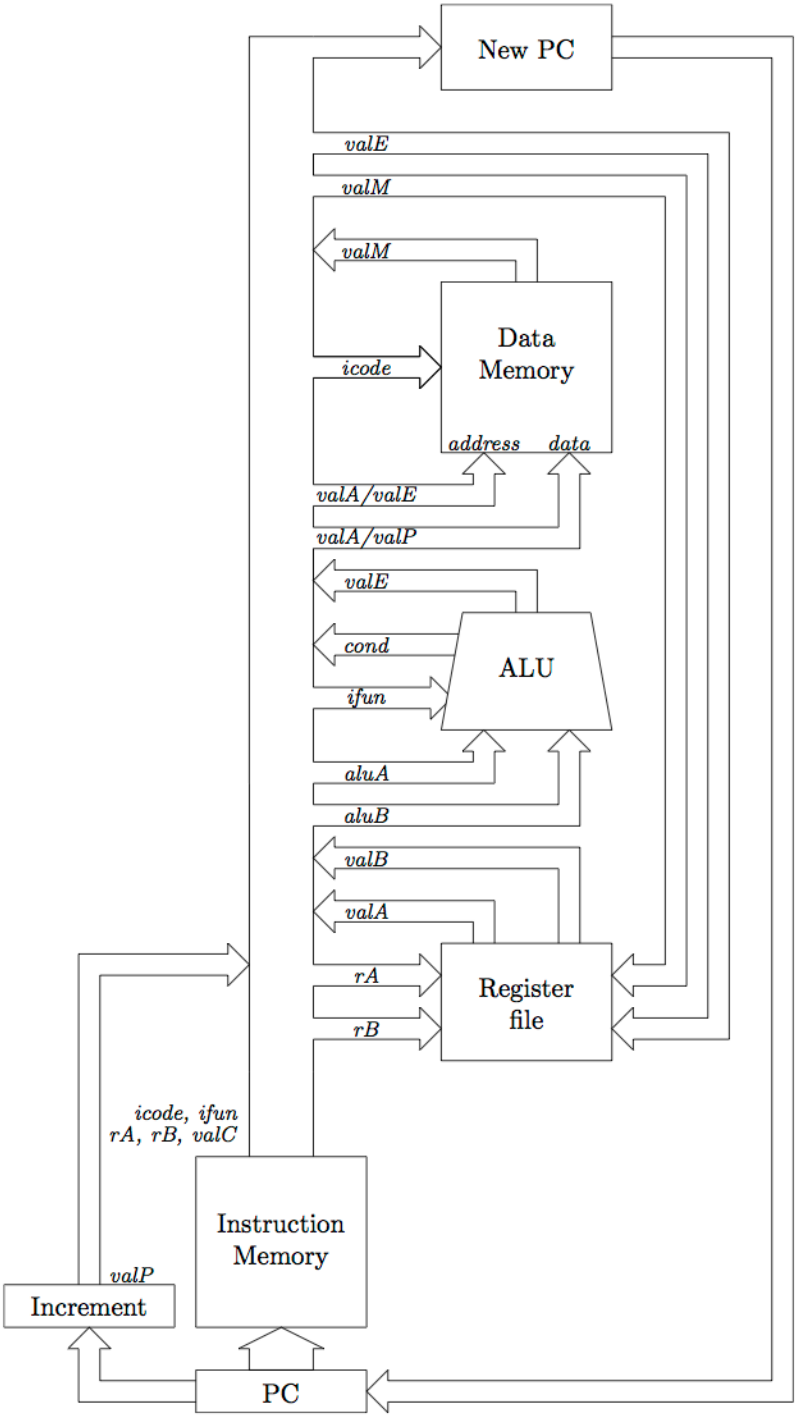
6. **PC Update**

   Make the PC point to the next instruction.

3.5.2 Datapath for Y86



Example 3.1 Show how to execute an instruction **`add %rax, %rcx`** (machine code = **`0x6001`**) located at address **`0x0000000000000000`**. Here, let **`%rax=10`**, **`%rcx=-9`**, and **`PC=0x0000000000000000`**.



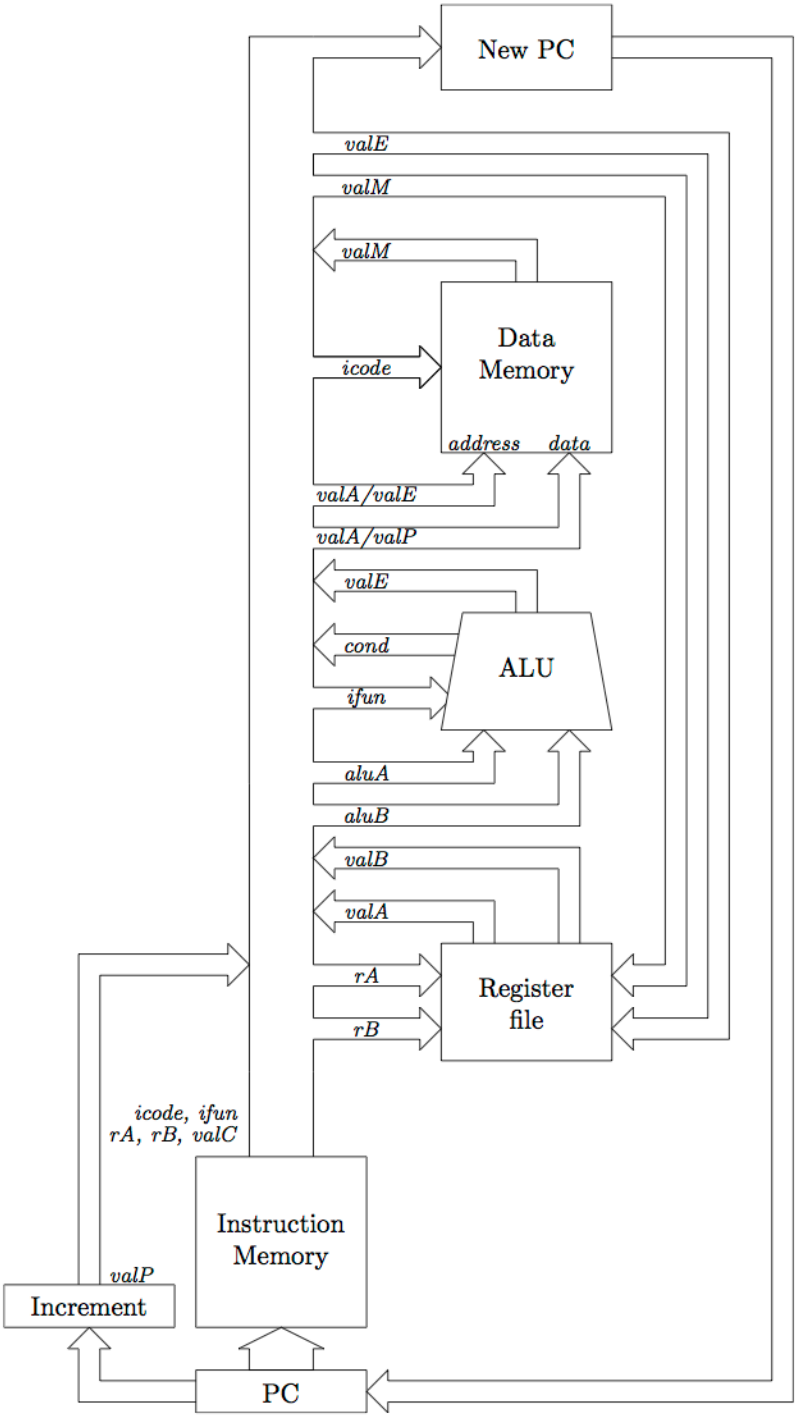
Stage	`add %rax, %rcx`
Fetch	$icode{:}ifun \leftarrow M_1[PC] = 6{:}0$ $rA{:}rB \leftarrow M_1[PC+1] = 0{:}1$ $valP \leftarrow PC + 2 = 0 + 2$
Decode	$valA \leftarrow R[0] = 10$ $valB \leftarrow R[1] = -9$
Execute	$valE \leftarrow valA + valB = 10 + (-9) = 1$ $cond \leftarrow ZF{:}SF{:}CF = 0{:}0{:}1$
Memory	
Write back	$R[1] \leftarrow valE$
PC update	$PC \leftarrow 2$

Exercise 3.2 Show how to execute the following program composing of two instructions as shown below. Here, let `%rax=4`, `%rcx=0x4000`, and `PC=0x0`.

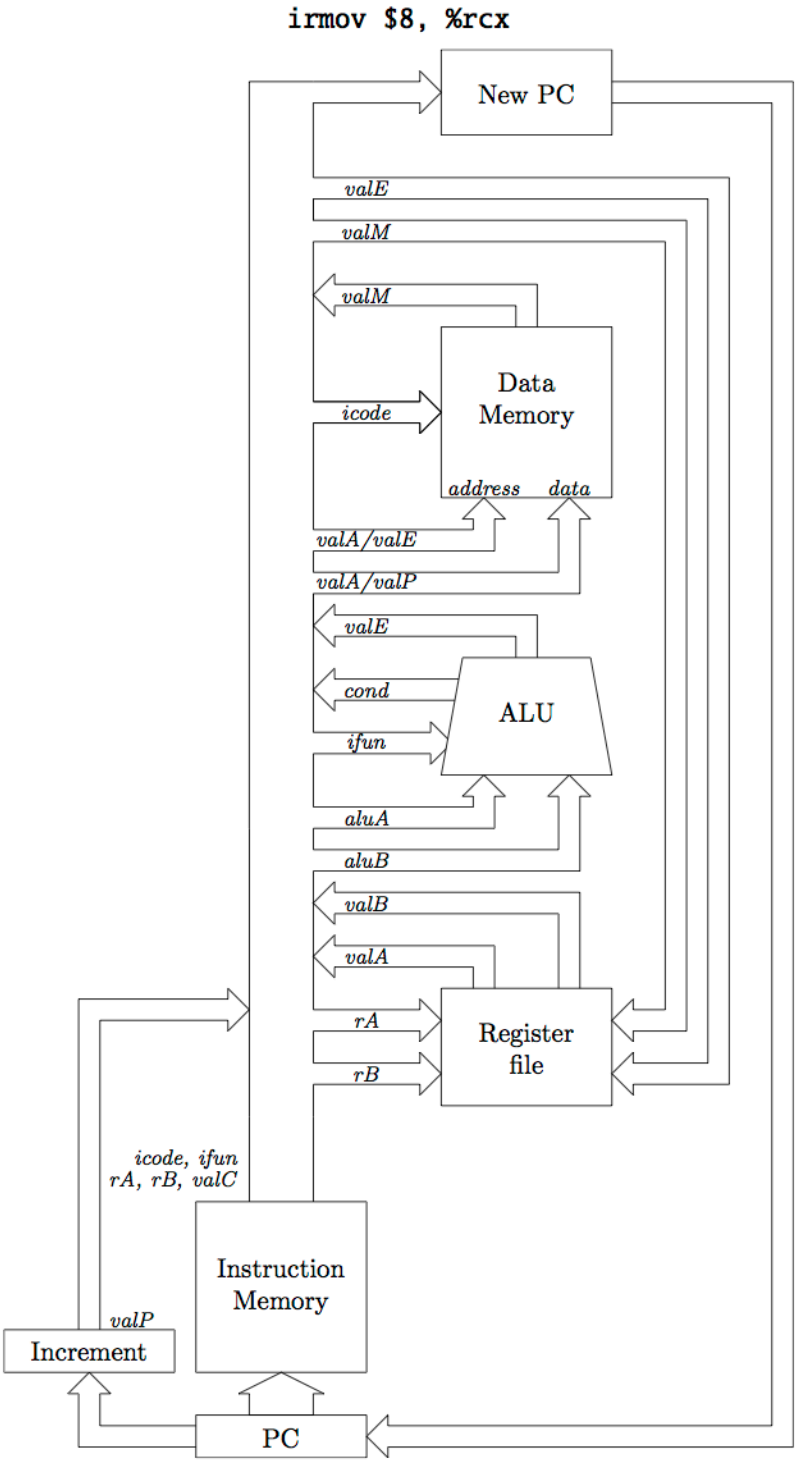
Address	Assembly	Machine Code
`0`	`rmmov %rax, 4(%rcx)`	`4001000000000000004`
`10`	`irmov $8, %rcx`	`30F10000000000000008`

Stage	`rmmov %rax, 4(%rcx)`
Fetch	
Decode	
Execute	
Memory	
Write back	
PC Update	

rmmov %rax, 4(%rcx)



Stage	`irmov $8, %rcx`
Fetch	
Decode	
Execute	
Memory	
Write back	
PC Update	


irmov $8, %rcx
New PC
valE
valM
valM
Data Memory
icode
address
data
valA/valE
valA/valP
valE
cond
ALU
ifun
aluA
aluB
valB
valA
rA
rB
Register file
icode, ifun
rA, rB, valC
Instruction Memory
valP
Increment
PC

3.5.3 Computation Steps

Halt (hlt)

Stop executing instruction.

Stage	`hlt = 0|0`	
Fetch	*icode*, *ifun*	*icode*:*ifun* $\leftarrow M_1[PC]$
	rA, *rB*	
	valC	
	valP	$valP \leftarrow PC + 0$
Decode	*valA*, *srcA*	
	valB, *srcB*	
Execute	*valE*	
	cond	
Memory	*valM*	
Write back	*dstE*	
	dstM	
PC update	*PC*	$PC \leftarrow valP$

No operation (nop)

Do nothing, then move to the next instruction.

Stage	`nop = 1|0`	
Fetch	*icode*, *ifun*	*icode*:*ifun* $\leftarrow M_1[PC]$
	rA, *rB*	
	valC	
	valP	$valP \leftarrow PC + 1$
Decode	*valA*, *srcA*	
	valB, *srcB*	
Execute	*valE*	
	cond	
Memory	*valM*	
Write back	*dstE*	
	dstM	
PC update	*PC*	$PC \leftarrow valP$

rrmov

Stage	`rrmov rA,rB = 2|0|rA|rB`	
Fetch	*icode*, *ifun*	$icode{:}ifun \leftarrow M_1[PC]$
	rA, *rB*	$rA{:}rB \leftarrow M_1[PC+1]$
	valC	
	valP	$valP \leftarrow PC+2$
Decode	*valA*, *srcA*	$valA \leftarrow R[rA]$
	valB, *srcB*	
Execute	*valE*	$valE \leftarrow valA+0$
	cond	
Memory	*valM*	
Write back	*dstE*	$R[rB] \leftarrow valE$
	dstM	
PC update	*PC*	$PC \leftarrow valP$

irmov

Stage	`irmov V,rB = 3|0|F|rB|V`	
Fetch	*icode*, *ifun*	$icode{:}ifun \leftarrow M_1[PC]$
	rA, *rB*	$rA{:}rB \leftarrow M_1[PC+1]$
	valC	$valC \leftarrow M_4[PC+2]$
	valP	$valP \leftarrow PC+6$
Decode	*valA*, *srcA*	
	valB, *srcB*	
Execute	*valE*	$valE \leftarrow valC+0$
	cond	
Memory	*valM*	
Write back	*dstE*	$R[rB] \leftarrow valE$
	dstM	
PC update	*PC*	$PC \leftarrow valP$

rmmov

Stage	`rmmov rA,D(rB) = 4|0|rA|rB|D`	
Fetch	$icode, ifun$	$icode{:}ifun \leftarrow M_1[PC]$
	rA, rB	$rA{:}rB \leftarrow M_1[PC+1]$
	$valC$	$valC \leftarrow M_4[PC+2]$
	$valP$	$valP \leftarrow PC+6$
Decode	$valA, srcA$	$valA \leftarrow R[rA]$
	$valB, srcB$	$valB \leftarrow R[rB]$
Execute	$valE$	$valE \leftarrow valC + valB$
	$cond$	
Memory	$valM$	$M_4[valE] \leftarrow valA$
Write back	$dstE$	
	$dstM$	
PC update	PC	$PC \leftarrow valP$

mrmov

Stage	`mrmov D(rB), rA = 5|0|rA|rB|D`	
Fetch	$icode, ifun$	$icode{:}ifun \leftarrow M_1[PC]$
	rA, rB	$rA{:}rB \leftarrow M_1[PC+1]$
	$valC$	$valC \leftarrow M_4[PC+2]$
	$valP$	$valP \leftarrow PC+6$
Decode	$valA, srcA$	
	$valB, srcB$	$valB \leftarrow R[rB]$
Execute	$valE$	$valE \leftarrow valC + valB$
	$cond$	
Memory	$valM$	$valM \leftarrow M_4[valE]$
Write back	$dstE$	
	$dstM$	$R[rA] \leftarrow valM$
PC update	PC	$PC \leftarrow valP$

ALU Instructions

Stage	`OP rA,rB = 6|X|rA|rB`	
Fetch	$icode, ifun$	$icode{:}ifun \leftarrow M_1[PC]$
	rA, rB	$rA{:}rB \leftarrow M_1[PC+1]$
	$valC$	
	$valP$	$valP \leftarrow PC+2$
Decode	$valA, srcA$	$valA \leftarrow R[rA]$
	$valB, srcB$	$valB \leftarrow R[rB]$
Execute	$valE$	$valE \leftarrow valA\ op\ valB$
	$cond$	$cond \leftarrow ZF{:}SF{:}CF$
Memory	$valM$	
Write back	$dstE$	$R[rB] \leftarrow valE$
	$dstM$	
PC update	PC	$PC \leftarrow valP$

Jump instructions

Stage	`jXX = 7|X|D`	
Fetch	$icode, ifun$	$icode{:}ifun \leftarrow M_1[PC]$
	rA, rB	
	$valC$	$valC \leftarrow M_4[PC+1]$
	$valP$	$valP \leftarrow PC+5$
Decode	$valA, srcA$	
	$valB, srcB$	
Execute	$valE$	
	$cond$	$branch \leftarrow Check(cond, ifun)$
Memory	$valM$	
Write back	$dstE$	
	$dstM$	
PC update	PC	$PC \leftarrow valC$ if $(branch == 1)$ else $valP$

Push

Stage	`push rA = A|0|rA|F`	
Fetch	*icode*, *ifun*	$icode{:}ifun \leftarrow M_1[PC]$
	rA, *rB*	$rA{:}rB \leftarrow M_1[PC+1]$
	valC	
	valP	$valP \leftarrow PC+2$
Decode	*valA*, *srcA*	$valA \leftarrow R[rA]$
	valB, *srcB*	$valB \leftarrow R[4]$ (`%rsp`)
Execute	*valE*	$valE \leftarrow (-4)+valB$
	cond	
Memory	*valM*	$M_4[valE] \leftarrow valA$
Write back	*dstE*	$R[4] \leftarrow valE$
	dstM	
PC update	*PC*	$PC \leftarrow valP$

Pop

Stage	`pop rA = B|0|rA|F`	
Fetch	*icode*, *ifun*	$icode{:}ifun \leftarrow M_1[PC]$
	rA, *rB*	$rA{:}rB \leftarrow M_1[PC+1]$
	valC	
	valP	$valP \leftarrow PC+2$
Decode	*valA*, *srcA*	$valA \leftarrow R[4]$
	valB, *srcB*	$valB \leftarrow R[4]$
Execute	*valE*	$valE \leftarrow (+4)+valB$
	cond	
Memory	*valM*	$valM \leftarrow M_4[valA]$
Write back	*dstE*	$R[4] \leftarrow valE$
	dstM	$R[rA] \leftarrow valM$
PC update	*PC*	$PC \leftarrow valP$

Call

Push the address of the next instruction, and jump to D

Stage	`call D = 8|0|D`	
Fetch	*icode*, *ifun*	*icode:ifun* $\leftarrow M_1[PC]$
	rA, *rB*	
	valC	$valC \leftarrow M_4[PC+1]$
	valP	$valP \leftarrow PC+5$
Decode	*valA*, *srcA*	
	valB, *srcB*	$valB \leftarrow R[4]$
Execute	*valE*	$valE \leftarrow (-4)+valB$
	cond	
Memory	*valM*	$M_4[valE] \leftarrow valP$
Write back	*dstE*	$R[4] \leftarrow valE$
	dstM	
PC update	*PC*	$PC \leftarrow valC$

Return

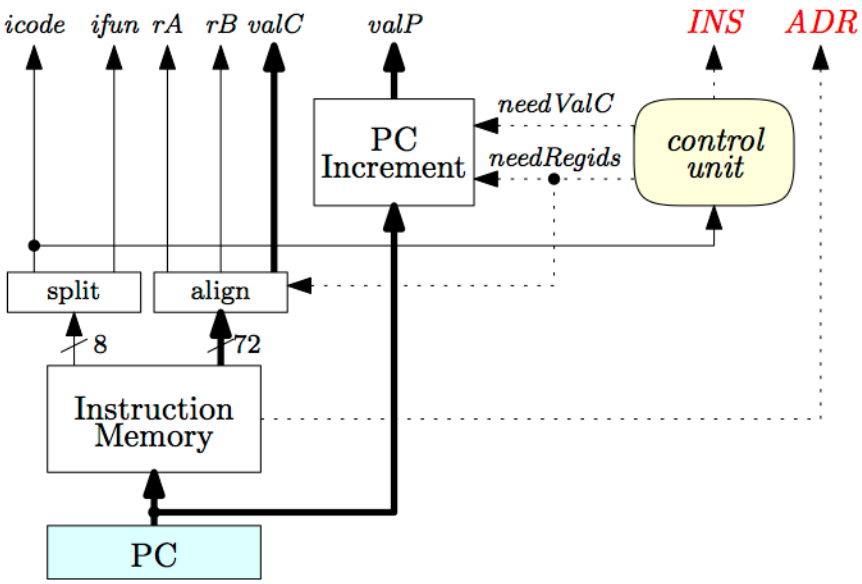
Pop an address from the stack, and jump to that address

Stage	`ret = 9|0`	
Fetch	*icode*, *ifun*	*icode:ifun* $\leftarrow M_1[PC]$
	rA, *rB*	
	valC	
	valP	
Decode	*valA*, *srcA*	$valA \leftarrow R[4]$
	valB, *srcB*	$valB \leftarrow R[4]$
Execute	*valE*	$valE \leftarrow (+4)+valB$
	cond	
Memory	*valM*	$valM \leftarrow M_4[valA]$
Write back	*dstE*	$R[4] \leftarrow valE$
	dstM	
PC update	*PC*	$PC \leftarrow valM$

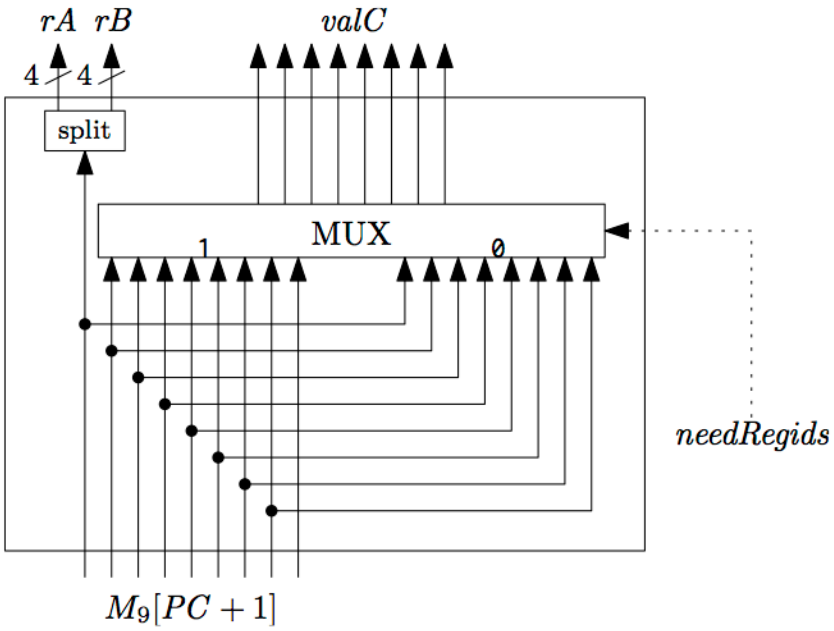
3.5.4 Control Unit

Up to now, we have studied how to execute an instruction by controlling the flow of values in the datapath. Practically, the flow can be only controlled by signals (in form of binary digits) from the control unit. In this section, we will study how to design the control unit.

Fetch Stage



The *align* component accepts 9 bytes from the address $PC + 1$ and selects the appropriate value of *valC* according to *icode*.



There are 3 control signals for separating the instruction into 5 fields i.e. *icode*, *ifun*, *rA*, *rB*, and *valC*. The values of the control signals can be shown in the following truth table. Note that *INS* is a signal indicating an *instruction error* when it is set to 1.

ADR indicates an *address error* when the specified addres is not in the memory.

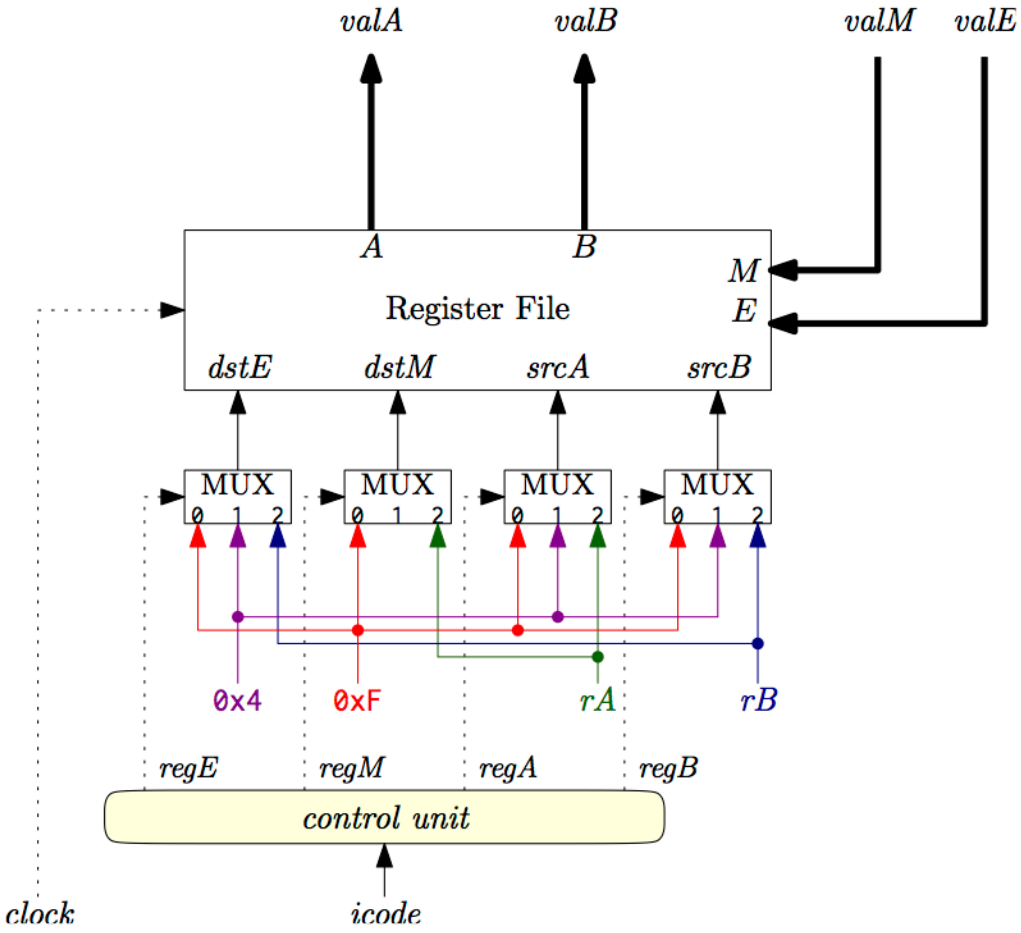
instruction	*icode*	*needRegids*	*needValC*	*INS*
`hlt`	`0000`	`0`	`0`	`0`
`nop`	`0001`	`0`	`0`	`0`
`rrmov`	`0010`	`1`	`0`	`0`
`irmov`	`0011`	`1`	`1`	`0`
`rmmov`	`0100`	`1`	`1`	`0`
`mrmov`	`0101`	`1`	`1`	`0`
`OP`	`0110`	`1`	`0`	`0`
`jXX`	`0111`	`0`	`1`	`0`
`call`	`1000`	`0`	`1`	`0`
`ret`	`1001`	`0`	`0`	`0`
`push`	`1010`	`1`	`0`	`0`
`pop`	`1011`	`1`	`0`	`0`
	`1100`	`0`	`0`	`1`
	`1101`	`0`	`0`	`1`
	`1110`	`0`	`0`	`1`
	`1111`	`0`	`0`	`1`

From the *needRegids* and *needValC* signals, we can compute the next instruction address as

$$valP = PC + 1 + (needRegids) + 8 \times (needValC)$$

Decode and Write-back Stages

We need to set appropriate values for the registers to be read in the decode stage and the registers to be written in the write-back stage.


valA
valB
valM
valE
A
B
M
E
Register File
dstE
dstM
srcA
srcB
MUX
MUX
MUX
MUX
0 1 2
0x4
0xF
rA
rB
regE
regM
regA
regB
control unit
clock
icode

Example 3.2 Write the truth table for the *regE* control signal .

instruction	*icode*	*regE*
`hlt`	`0000`	`00` = `0xF`
`nop`	`0001`	`00` = `0xF`
`rrmov`	`0010`	`10` = rB
`irmov`	`0011`	`10` = rB
`rmmov`	`0100`	`00` = `0xF`
`mrmov`	`0101`	`00` = `0xF`
`OP`	`0110`	`10` = rB
`jXX`	`0111`	`00` = `0xF`
`call`	`1000`	`01` = `%rsp`
`ret`	`1001`	`01` = `%rsp`
`push`	`1010`	`01` = `%rsp`
`pop`	`1011`	`01` = `%rsp`
	`1100`	`00` = `0xF`
	`1101`	`00` = `0xF`
	`1110`	`00` = `0xF`
	`1111`	`00` = `0xF`

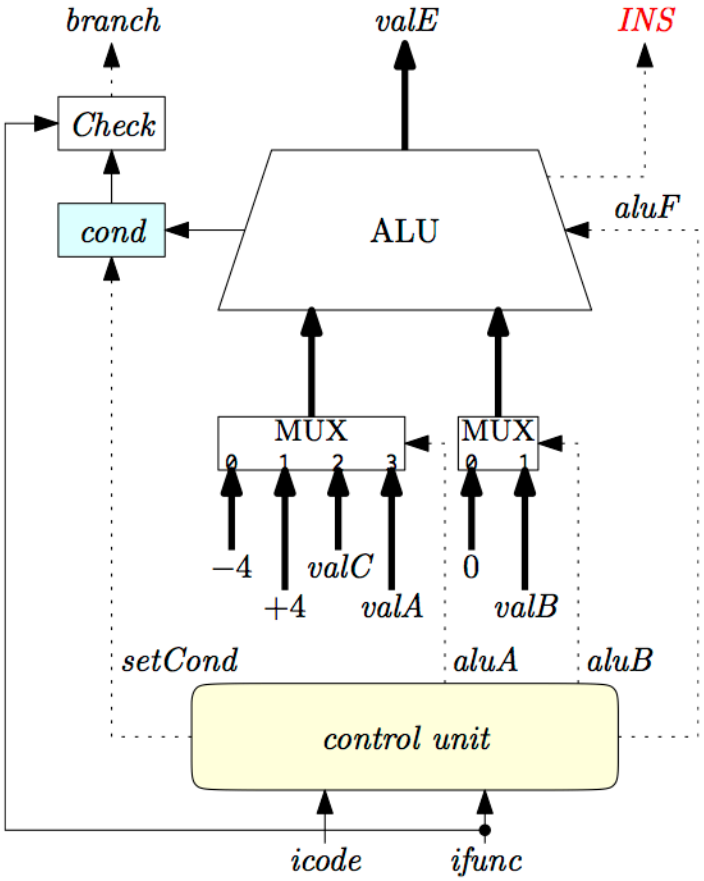
Exercise 3.3 Complete the following truth table for the *regA* signal.

instruction	*icode*	*regA*
`hlt`	`0000`	
`nop`	`0001`	
`rrmov`	`0010`	
`irmov`	`0011`	
`rmmov`	`0100`	
`mrmov`	`0101`	
`OP`	`0110`	
`jXX`	`0111`	
`call`	`1000`	
`ret`	`1001`	
`push`	`1010`	
`pop`	`1011`	
	`1100`	
	`1101`	
	`1110`	
	`1111`	

Execute Stage

In this stage, two values are fed to the ALU for calculation. The *aluF*, a 2-bit control signal, is used to select the operation (`00`=add, `01`=subtract, `10`=bitwise and, `11`=bitwise xor).

The ALU also outputs *cond* composing of *ZF*, *SF*, and *CF*. This value is compared with *ifun* to determine if a conditional jump will be taken.



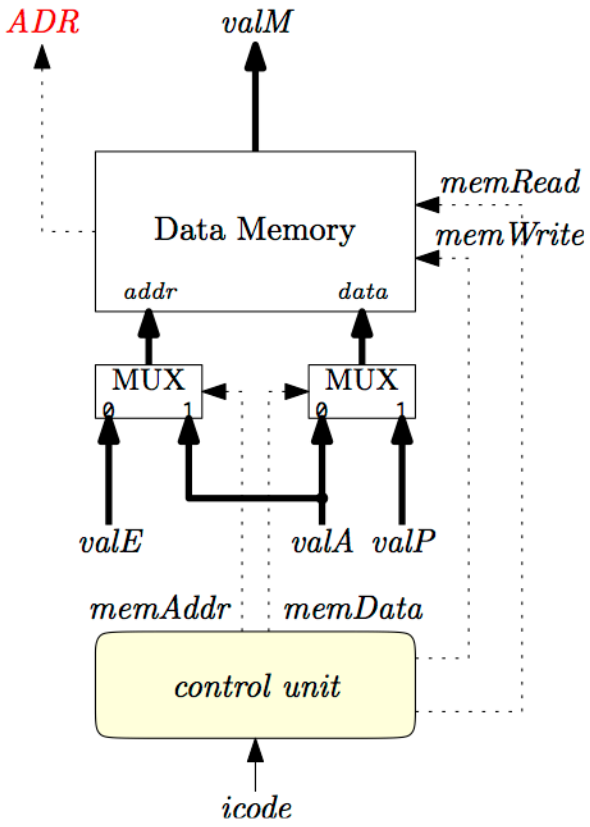
Exercise 3.4 Complete the following truth table for the *aluA*, *aluB*, and *aluF* signals.

instruction	*icode*	*aluA*	*aluB*	*aluF*
`hlt`	`0000`			
`nop`	`0001`			
`rrmov`	`0010`			
`irmov`	`0011`			
`rmmov`	`0100`			
`mrmov`	`0101`			
`OP`	`0110`			
`jXX`	`0111`			
`call`	`1000`			
`ret`	`1001`			
`push`	`1010`			
`pop`	`1011`			
	`1100`			
	`1101`			
	`1110`			
	`1111`			

Memory Stage

To read a piece of data from the *data memory*, we need to specify the *address* to be read, set '`1`' to *memRead*, and set '`0`' to *memWrite*.

To write a piece of data to the *data memory*, we need to specify the *data*, the *address*, set '`0`' to *memRead*, and set '`1`' to *memWrite*.



Exercise 3.5 Complete the following truth table for the *memAddr*, *memData*, *memRead* and *memWrite* signals.

instruction	*icode*	*memAddr*	*memData*	*memRead*	*memWrite*
`hlt`	`0000`				
`nop`	`0001`				
`rrmov`	`0010`				
`irmov`	`0011`				
`rmmov`	`0100`				
`mrmov`	`0101`				
`OP`	`0110`				
`jXX`	`0111`				
`call`	`1000`				
`ret`	`1001`				
`push`	`1010`				
`pop`	`1011`				
	`1100`				
	`1101`				
	`1110`				
	`1111`				

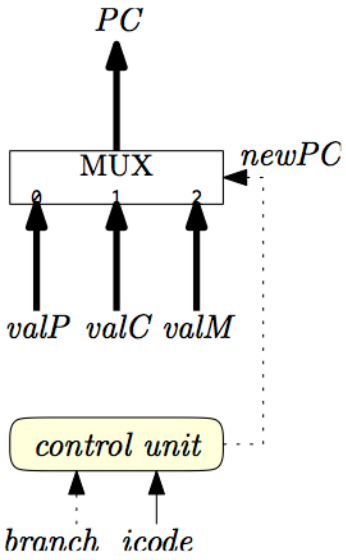
PC Update Stage

The final stage of the datapath sets the new value of *PC*. This is the address of the next instruction to be executed.

The new value is set to *valP* which is the address of the next instruction in the instruction memory.

It is set to *valC* for `jmp` and `call` instructions, as well as all the conditional jump instructions when *branch* signal is '`1`'.

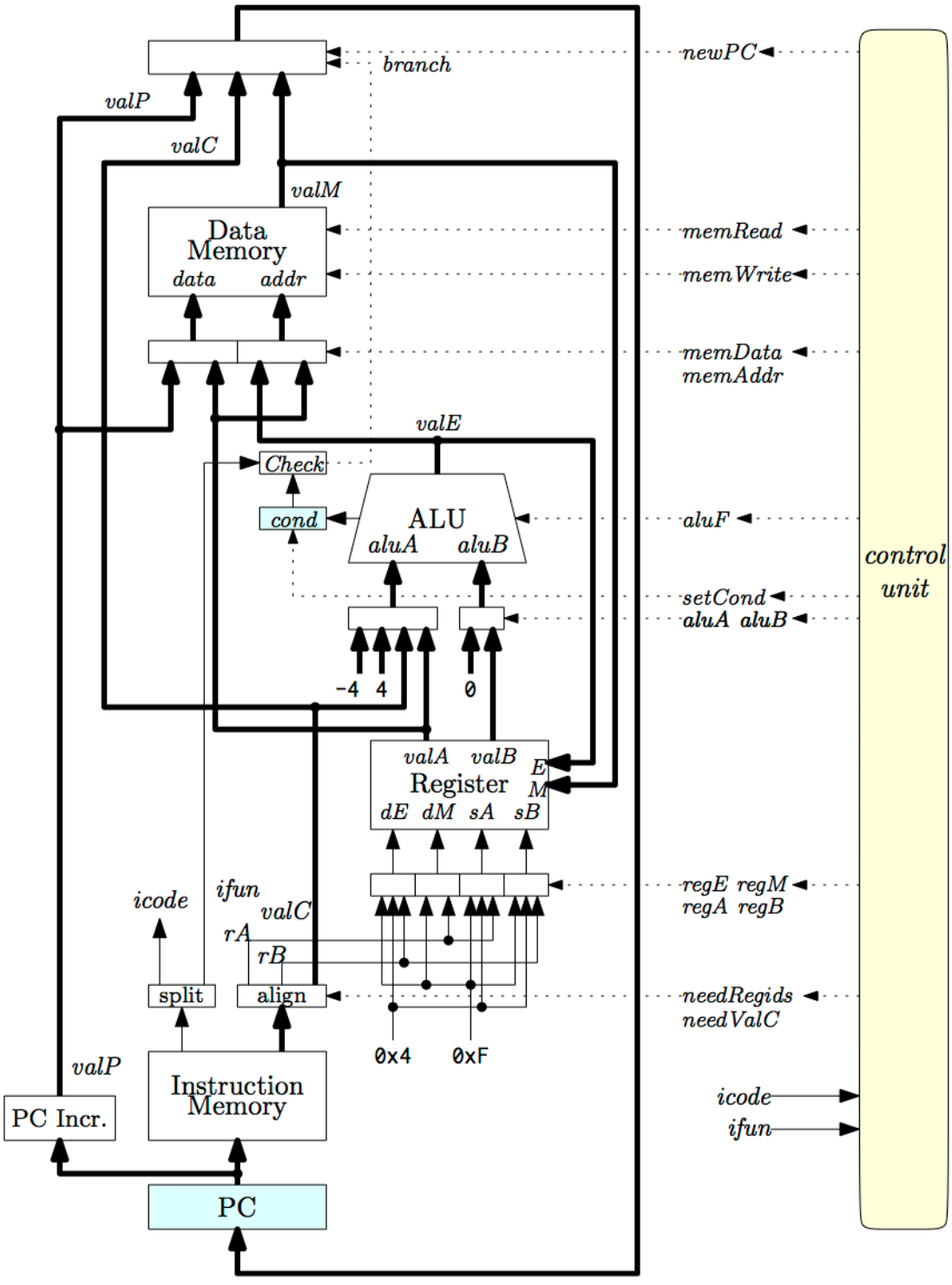
It is set to *valM* for `ret` instruction.



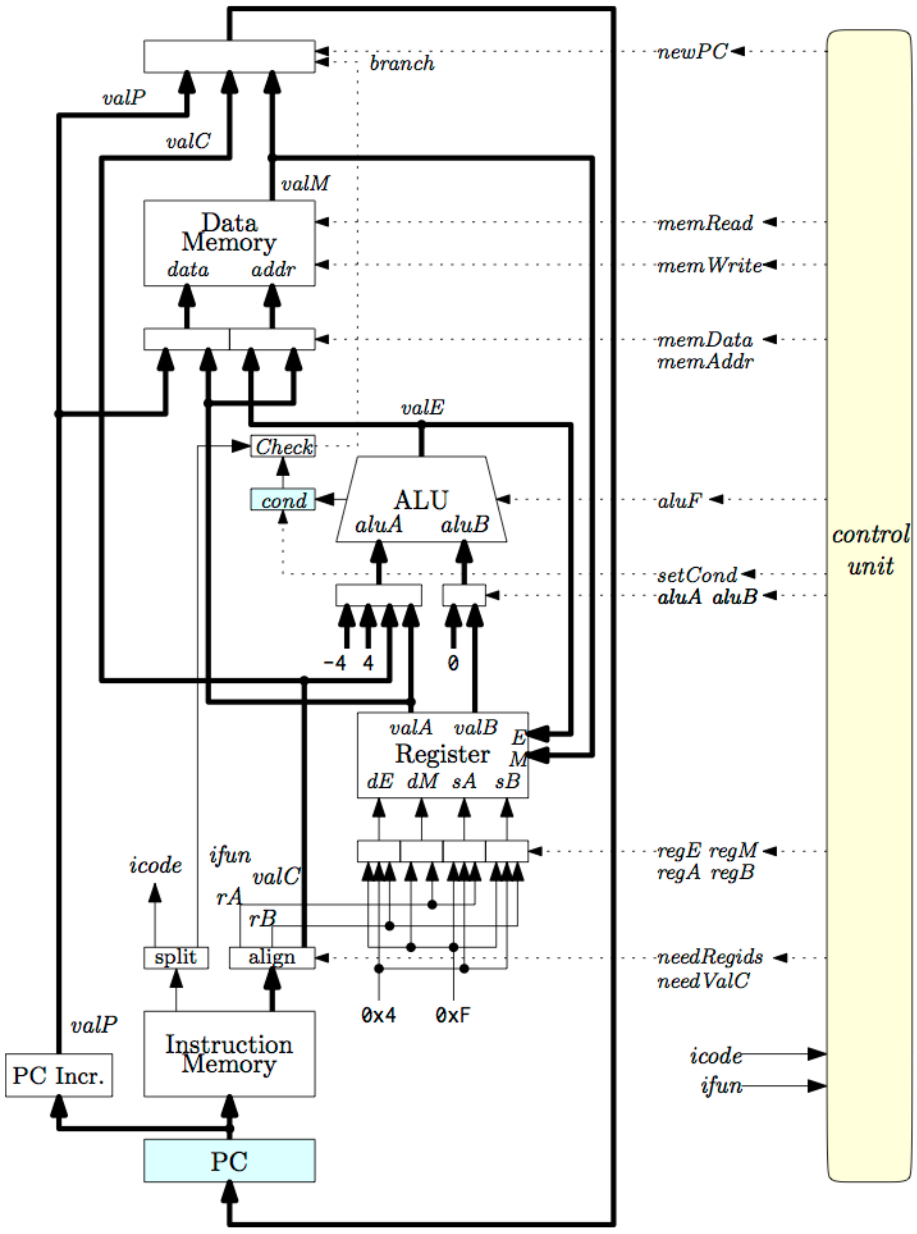
The truth table of the *newPC* signal is left as a self-practicing exercise.

Execution Status

3.5.5 Combining All Stages



Exercise 3.6 Highlight the path used to execute an instruction `rmmov %rax, 8(%rbp)`, and specify the values of all the control signals.



Exercise 3.7 What is the CPI for this implementation of Y86?

3.6 Determining Clock Period

The *clock period* needs to be long enough to complete all the stages in the entire datapath. Otherwise, we cannot guarantee the correct results.

$$P = P_f + P_d + P_e + P_m + P_{wb} + P_{pc} \tag{3.1}$$

References

[BO10] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective (2nd edition)*. Addison-Wesley, 2010.