Intro to Processor Architecture

Y-86 Instruction Set Architecture
Project report

In this project, I designed y-86 sequential architecture and y-86 pipeline architecture. The Y-86 sequential architecture will work for all the Y-86 instructions.

Y-86 Instructions

The Y-86-64 instruction set architecture (ISA) is a subset of the x86-64 instruction set. This instruction set architecture will process 8-byte integer operations. A total of 12 instructions are included in this Y86-64 ISA. It includes arithmetic operations like addition, subtraction, and bitwise operations like AND, XOR. Other than that it also includes move instructions to move data to registers and memory, it also includes nop, halt, call, ret, push, pop, conditional and unconditional jump, move instructions.

The length of a Y-86 instruction ranges from 1 to 10 bytes depending on the instruction code(icode). The first byte of the instruction consists of an icode, ifun each of four-bit length. This icode determines whether the instruction needs the register file address(rA, rB) or value(V or D or Dest) in the instruction memory to perform the instruction. The ifun determines the type of arithmetic operation or condition type for jump, conditional move instructions. While designing the Y86 processor the next program counter(PC) will increase depending on the length of the current instruction and also on the jump, call and return instructions. The instruction set of Y86-64ISA are written in page 11.

Sequential Y86-64 processor

For processing an instruction several operations are involved like reading the instruction from the instruction memory, obtaining the values from registers, doing operations on the values, and writing in memory or register file. All these operations can be divided into 5 stages and connecting these 5 modules in an organized manner will give us a complete processor. The five stages are Fetch, decode, execute, memory, writeback. The output one

module is connected as the input to the other module and finally when the writeback stage(module) occurs the pc will point to the next instruction that needs to be executed and this be given as input to our fetch stage.

In a sequential processor, all these 5 stages are executed one after another in a single clock cycle.

Stages of Y86-64 processor

Each pipeline stage is written as a separate module and a main module is written to combine all these modules. The main module processes each instruction written in the instruction memory file sequentially. The main module executes one instruction per single clock cycle.

Fetch

The block diagram used for implementing the fetch module is shown in fig-1. The read colors line is the output from the fetch module. The instruction memory is a block where the instructions are stored. The input to the fetch module comes from the program counter(PC).

During the execution of the fetch stage, initially, it reads the first byte of instruction by accessing the address of instruction memory using the pc. Then the first four bits are assigned to *icode* and the next 4 bits are assigned to *ifun* if the pc points to a valid address in the instruction memory otherwise *imem_error* becomes 1 and the processor stops by showing an error in the stat block. If *icode (>11)* is not a valid Y86 instruction then *instr_valid* becomes 1 and the processor stops execution. If *imem_error* = 0, *instr_valid* =1 then *icode* and *ifun* are fetched properly. Now depending on the *icode* the decision block need valC need regIDs reads *rA, rB* and *valC* from the *instruction memory.* Depending on *icode* and *needvalC, needregIDs* the pc increment will increment the pc value so that it can point to the start of next instruction and stores it in the v*alP.*

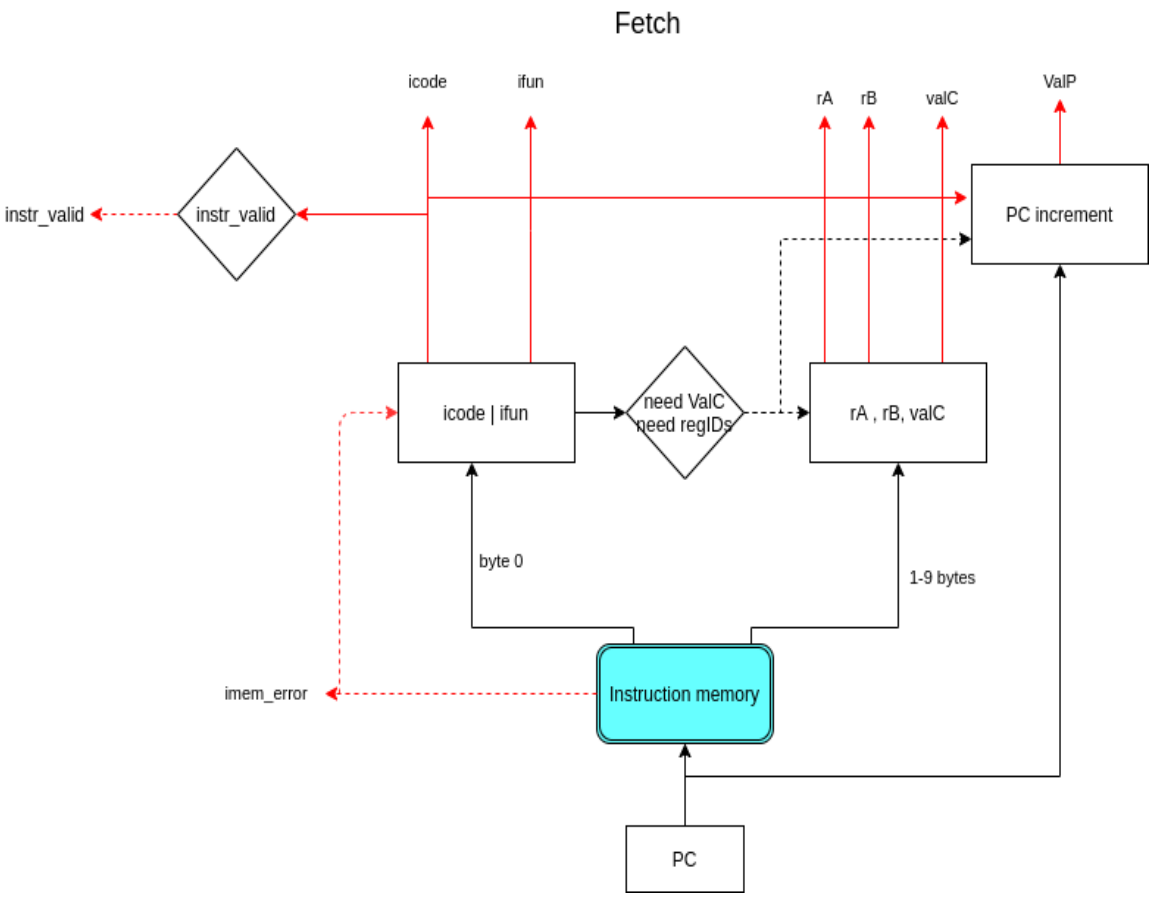


Fig - 1 Fetch module , dashed lines are 1 bit wires, icode,ifun,rA,rB are 4 bit wires. valC is 64bit and valP is 10 bit wires.

Rhombus blocks are logical blocks, rectangular blocks are assigning blocks.

Ex: IRMOVQ Let's say *pc = 2* and instruction memory[2] is 30(in hex) and splits it into *icode, ifun* each of four bits, then our *needregIDs and needvalC* (because icode = 3) both sets 1. So, it reads the next 1 byte from the PC and splits it into *rA, rB*. Then it reads the next 8bytes and sets it into *valC*. The PC increment will increment the pc by the length of the instruction it reads from PC. Here the length of the instruction is 10 bytes, so *valP* = 10+pc(=2) = 12. Assuming instruction memory is of enough size and pc = 2 is a valid address then our *imem_error = 0* and as the instruction is valid, *instr_valid = 1.*

Decode and write back

The block diagram used for implementing decode and write-back module is as shown in Fig -2. The register file in the fig-2 is a 2D array registers having 16 rows each row is of length 64 to store the 64 bit number. The 16 rows acts as 16 registers(r0 - r15) where each register can store 64 bits.

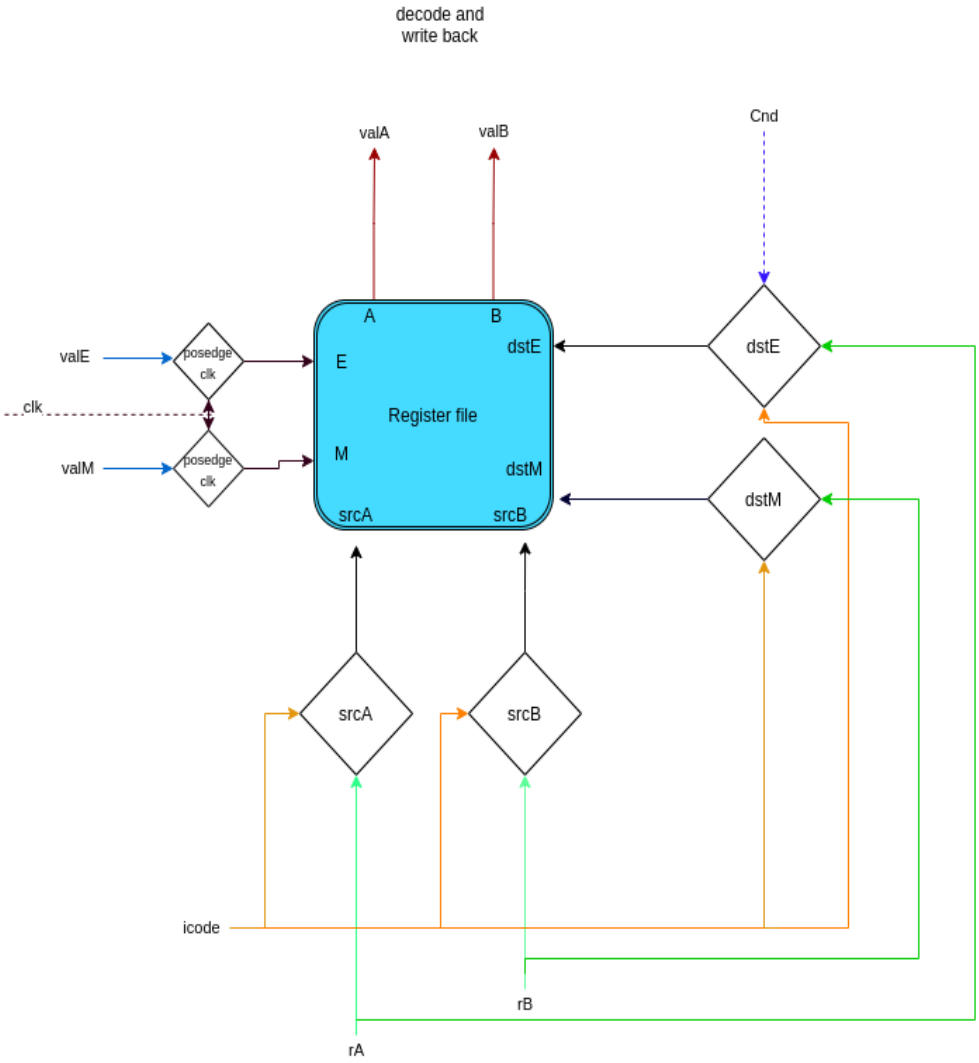


Fig - 2 Decode - writeback module. srcA, srcB,dstE,dstM are logical blocks that send data depending on icode,cnd. valA, valB, valE, valM are 64 bit wires. clk, cnd are 1 bit wires. icode, rA,rB are 4bit wires. valE, valM writes at posedge clk.

In the decode, we take *icode,rA.rB* as inputs and then depending on the value of icode we *rA* and assign it to *valA,* also read *rB*, and assign it to *valB.* The srcA and srcB blocks will determine whether we need to write *valA, valB* depending on icode.

In the write back, *valE* and *valM* needs to write into the *register file* depending on the *icode.* The destination registers for writing *valE* and *valM* are settled by the logical block dstE and dstM based on the *icode.* These values are written only when the positive edge of the clock cycle is triggered. This is because by the time all the *valE*, *valM* are settled correctly and we can write them to the register file memory.

Ex:

Let's take the instruction having *icode = 6, ifun = 0.* Let's say *rA* = 2, *rB = 3.* This instruction performs addition and then writes the result into *rB.*

In the decode module we assign *valA = reg_file[rA] , valB = reg_file[rB]* where the reg_file is 2d array consisting of 16 registers. In the execute stage we will get the sum(*valE)* of *valA, valB* and finally at the writeback our *dstM = f (*no value from memory needs to be written). *dstE = rB* and at the posedge *clk* we write *valE* into *dstE.*

Execute

The block diagram used to implement the execute module is shown in fig.3 . The ALU block performs the operations add, sub, AND, XOR depending on the ALU func.

In the execute module we perform the arithmetic and bitwise operations. This module also performs the conditional operations and sets the condition bit based on the condition instructions. This stage computes the effective memory address, increments or decrements of the stack pointer. For conditional move instruction , the condition codes will be evaluated in this module. These modules have inputs *valC* ,*valA*,*valB*,*icode* and *ifun*. Based on the inputs the module outputs *valE* and sets the *cnd* bit. ALU A , ALU B blocks send the input to ALU based on the icode. The ALU func. output is decided by the *icode*, *ifun*. The condition block sets the cnd based on the *ifun* and using the condition code register.

EX:

Let's say our *icode* = 6, *ifun* = 0, *valA* = 3, *valB* = 6. As icode is 6 ALUA outputs *valA*= 3 and it is the input of ALU and ALUB outputs *valB* = 6. As *icode* =6 , *ifun* = 0 the ALU performs addition operation and outputs the sum as *valE*(= 9) . It also sets the condition code register(CC) which contains signed, overflow, zero flag.

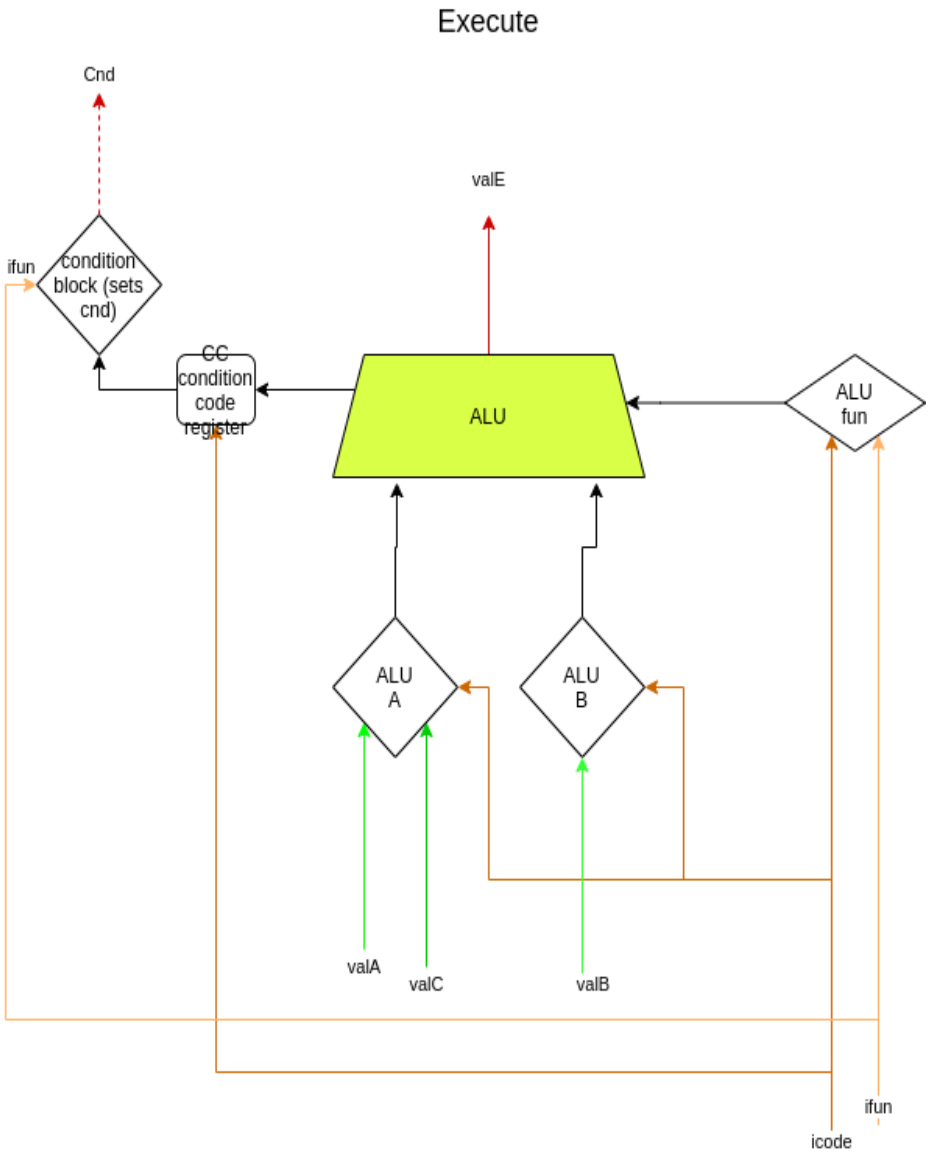


Fig - 3 Execute module. Icode, ifun are 4 bit wires. valX are 64 bit wires. Cnd is a 1 bit wire.

MEMORY

The block diagram for the implemented memory module is as shown in fig4. The inputs for the memory stage are *icode, valA, valE, valP* and *icode*. The output is *valM* and also displays the current instruction status of the processor.

Memory stage can either read data from the memory or write data into the memory. Reading and writing operations are done based on the icode. Mem_address blocks output the *address* of the memory that needs to be read or written. The mem_Data block outputs the *data* if anything needs to be written into the memory. Mem_read and mem_write sets the *read* or *write* flags to 1 based on the instruction (*icode*).

Memory stage contains a stat block. This stat block indicates the status of the processor by taking *imem_error, dmem_error, instr_valid, icode* as inputs.

Stat block description

If (*dmem_error* || *imem_error*) stat = SADR(address error)

Else if(!*instr_valid*) stat = SINS (instruction not valid)

Else if (*icode* == 0) stat = SHLT (halt operation encountered. Processor stops execution)

Else Stat = SAOK (normal operation)

Ex: Let's say our *icode* = 4 (rmmovq). This instruction writes the data in *rA* to *valE* = *valC* + reg_file[rB] in the memory. Let's say our *valA*(reg_file[*rA*]) = 2, valE = 4.

In the memory stage *valE* is taken as *address*, *valA* is considered as *data*. The icode sets the write flag to 1. So, now memory[address(*valE*)] writes to *valA* => memory[4] = 2. If all the errors are zero and *instr_valid* is 1 means the instruction is performed normally. Therefore, stat block displays SAOK.

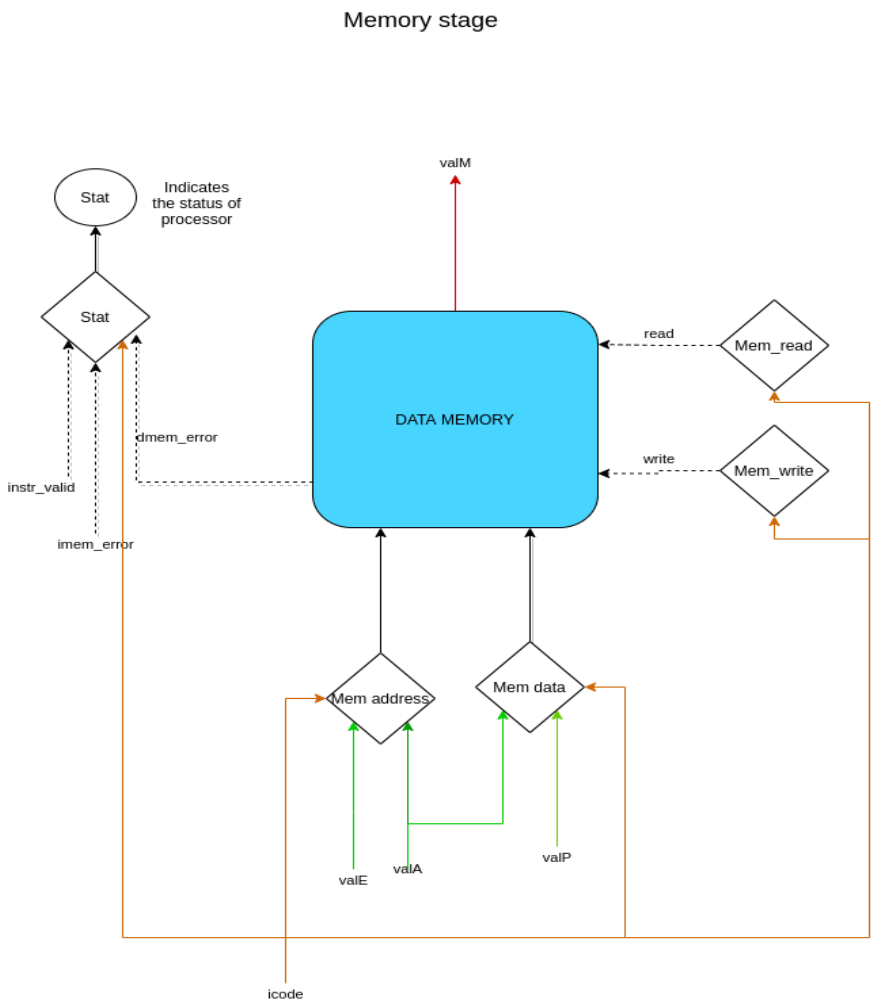


Fig -4 Memory stage. Icode is 4bit wire, valX are 64 bit wire and dashed line are single bit wire(signals).

PC update

PC update is a small module updating the PC value to the next instruction it needs to be pointed. The *PC* value can change to the next instruction or jumps to a destination pc value if the condition is satisfied or it can also update based on call, ret instruction. All this is done based on the instruction opcode(icode).

Fig -5 shows the block diagram for the PC update. The output of the pcupdate block is the next pc value. The pc update is only updated when all the previous modules are completed. For this I took a *clk* and I am given enough cycle time so that all the modules will complete their execution. At every rising edge of the clk I will update the *pc* value.

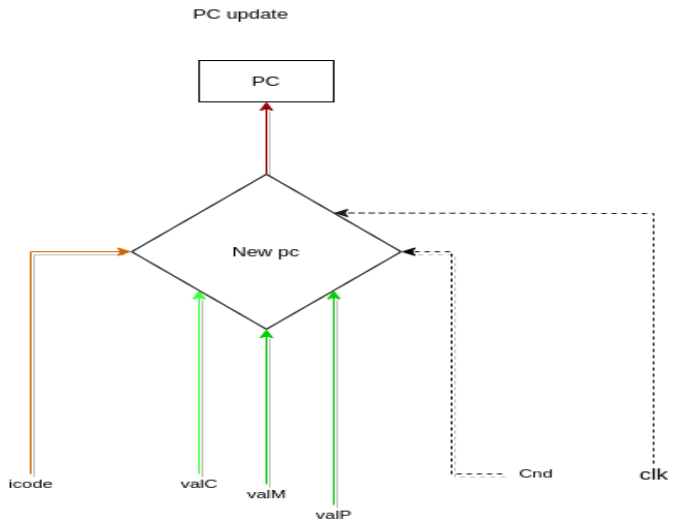


Fig - 5 PC update.

Integrating all modules

The final block diagram for the Y86-64 sequential processor is as shown in FIg-6. All the modules of fetch, decode,execute,memory, writeback and pcupdate are combined in a single module as shown in Fig - 6. The blocks of fetch,decode-writeback, execute, memory and pcupdate are designed as shown in fig 1-6.

Y86 processor wires description

valA, valE, valM,valC,valE,valP - 64 bit wires used to the values to different stages.

Icode, ifun - 4 bit wires, used to send the instruction type and its function to different stages.

rA,rB - 4 bit wires, sends the addresses read from the instruction memory to decode-writeback module.

Imem_error, dmem_error, instr_valid - single bit wire used to update and status of the processor.

Cnd - single bit wire, used to send signal when jump or condition move condition satisfied.

Clk - 1 bit wire, used as clk with time period 10ns and to update pc, write into the register file.

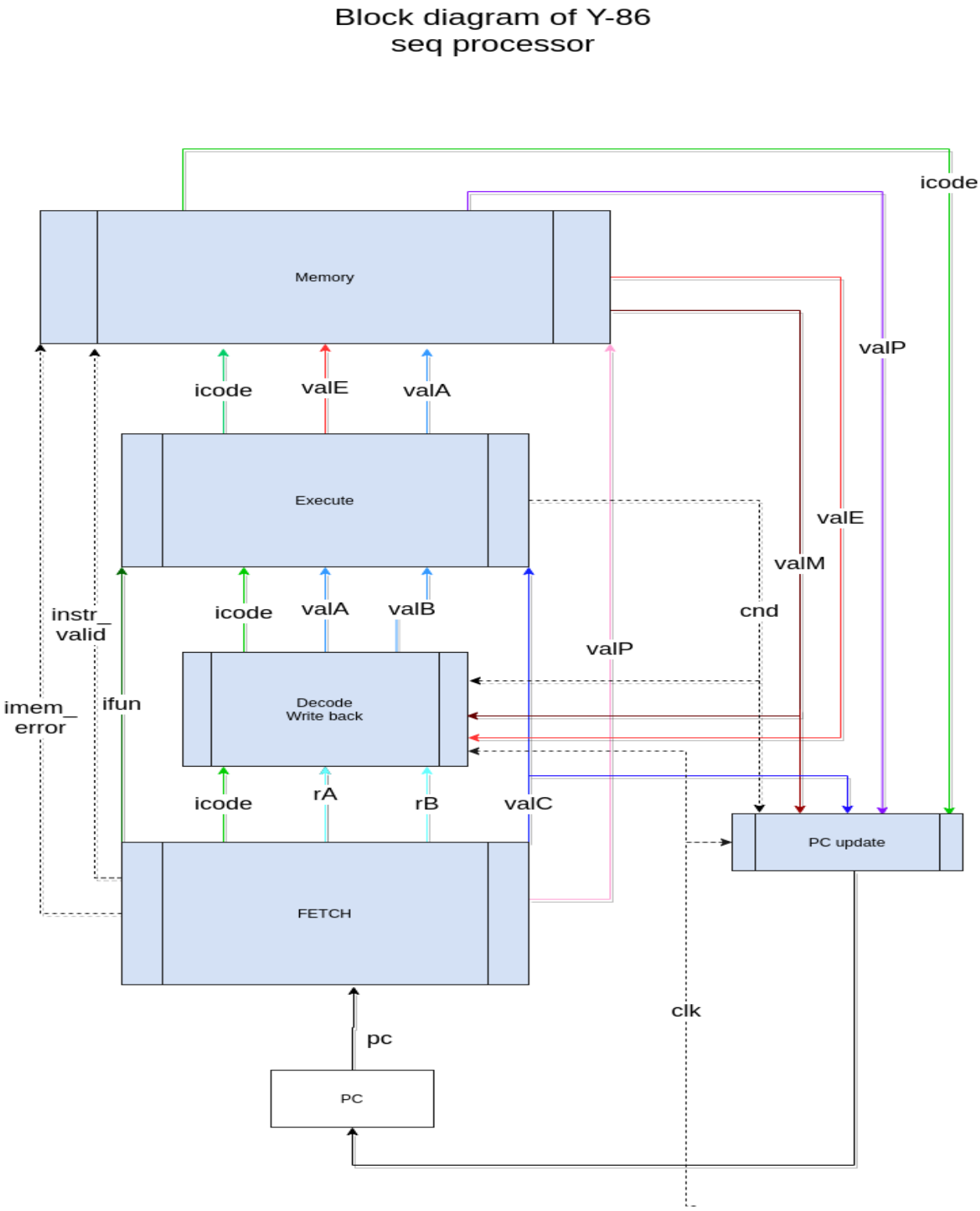


Fig - 6 Block diagram for sequential processor.

Instructions supported by Y86-64 sequential processor

The designed Y86-64 sequential processor supports all the 12 y86-64 instruction set instructions. The instruction encoding and length of each instruction for all the 12 instructions of the Y86-64 instruction set are as follows.

Instruction	icode	ifun	need regs		val
halt	0	0			
nop	1	0			
cmovq rA,rB	2	fn(0-6)	rA	rB	
Irmovq V,rB	3	0	F	rB	V(8 bytes)
Rmmovq rA, D(rB)	4	0	rA	rB	D(8 bytes)
Mrmovq D(rB), rA	5	0	rA	rB	D(8ytes)
OPq rA, rB	6	fn(0-6)	rA	rB	
jXX Dest	7	fn(0-6)			Dest(8 bytes)
Call Dest	8	0			Dest(8 bytes)
Ret	9	0			
Pushq rA	A	0	rA	F	
Popq rA	B	0	rA	F	

Operations, jump and move instruction encodings for different ifun is as follows.

Operations		icode	ifun	Branches		icode	ifun	Moves	fun	icode
addq	-	6	0	jmp	-	7	0	rrmovq	2	0
subq	-	6	1	jle	-	7	1	cmovle	2	1
Andq	-	6	2	jl	-	7	2	cmovl	2	2
Xorq	-	6	3	je	-	7	3	cmove	2	3
				Jne	-	7	4	cmovne	2	4
				Jge	-	7	5	cmovge	2	5
				Jg	-	7	6	cmovg	2	6

Each Y86-64 instruction is of the form - **icode(4bits)+ifun(4bits)+rArB(0 or 1 byte)+ valC(0 or 8bytes).** '+' means concatenate here.

Encoding a Y86 instruction.

Let's say we want to move a value to a particular register.

The Y86 instruction is irmovq rB val => let's say irmovq %rsp 5

The Y86 encoding for this instruction can be done by looking above instruction encoding table - **30f40500000000000000**

Note - while giving V or D or Dest value in the instruction encoding the every byte should be written in reverse order to the actual form.

This will give instruction encoding from the assembly language of Y86 to byte representation for all the Y86-64 instructions.

Note - A code for finding the hcf of two two numbers is written in C and the c file, assembly code, encoding instruction file is in the github repository.

GTK Wave outputs

The GTK wave output for finding the HCF using Y86 instruction encoding is as follows.

The inputs(two numbers) are stored in the registers RA, RB and the output/ HCF of the two numbers is stored in the R0 register.

The only registers used for finding the HCF of two numbers are Ra,Rb are inputs and r1,r2 as intermediate registers and also R0 for storing the HCF and as well intermediate registers used for swapping.

For testing the HCF Y86 instruction encoding file I took a = 4, b = 6 and kept a in rA, b in Rb. Then I run the **seq_processore.v** file to find the output.

The first two cycles will move the immediate values(*valC)* to index 1, index 2 into the register file 2D array.

```
 Fetch -- icode - 3, ifun -  0, rA -f, rB - b, valc - 0000000000000006, valP - 014, pc - 00a
Decode -- icode - 3, valA - 0000000000000000, valB- 0000000000000000
Execute -- icode - 3 ,ifun - 0, valA -                    0, valB -                    0, valC -
         6, valE -                    6, over,neg,zero - 000
stat - SAOK , icode - 3

Memory -- icode - 3, address - 0000000000000000, data - 0000000000000000, read - 0 , write - 0
write back -- Time -- 3000 ps,icode - 3,cnd - x, valE - 0000000000000006, valM - xxxxxxxxxxxxxxxx
REGISTER FILE
R0 -                    0     ----     R1 -                    0
R2 -                    0     ----     R3 -                    0
R4 -                  128     ----     R5 -                    0
R6 -                    0     ----     R7 -                    0
R8 -                    0     ----     R9 -                    0
R10 -                   4     ----     R11 -                   6
R12 -                   0     ----     R13 -                   0
R14 -                   0     ----     R15 -                   0
```

Fig - 6

The register file contents at the end of the execution is as follows. We can observe that R0 stores the GCD of the numbers in R10 and R11.

```
write back -- Time -- 57000 ps,icode - 2,cnd - 1, valE - 0000000000000002, valM - xxxxxxxxxxxxxxx
REGISTER FILE
R0 -                    2      ----      R1 -                    0
R2 -                    2      ----      R3 -                    0
R4 -                  128      ----      R5 -                    0
R6 -                    0      ----      R7 -                    0
R8 -                    0      ----      R9 -                    0
R10 -                   4       ----      R11 -                   6
R12 -                   0       ----      R13 -                   0
R14 -                   0       ----      R15 -                   0



Stat -SHLT
```

Fig -7

Gtk wave outputs

Initial time

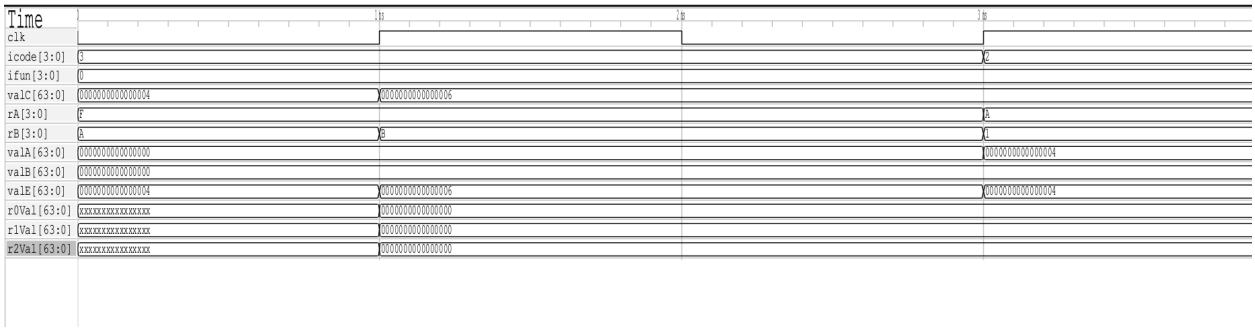


Fig - 8

Final time

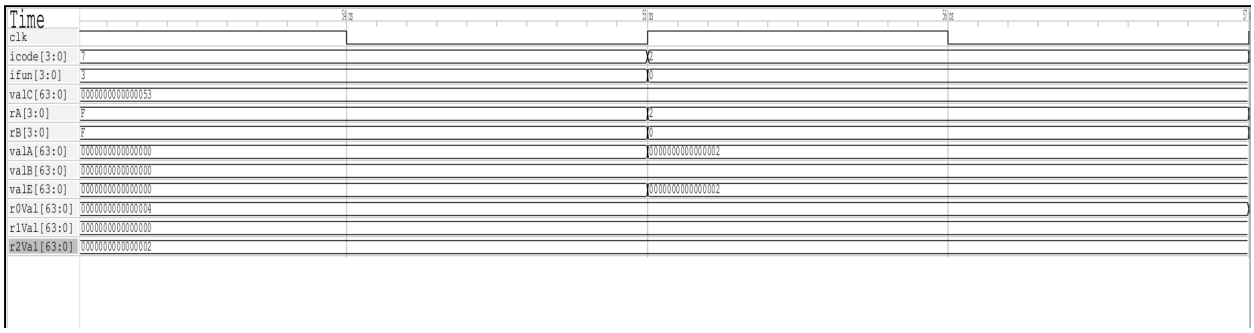


Fig - 9

Hero in the GTK waves above r0Val is my output, r1Val, r2Val are my intermediate registers used. R10 and R11 are inputs which are constant throughout my execution.

The execution for finding HCF of two numbers will take several clock cycles depending on how big the numbers are. So, in order to see the output for gtk wave let's look at the execution of adding 2 numbers in two different registers.

Instructions -

30 f2 05 00 00 00 00 00 00 00

30 f1 06 00 00 00 00 00 00 00

60 12

00

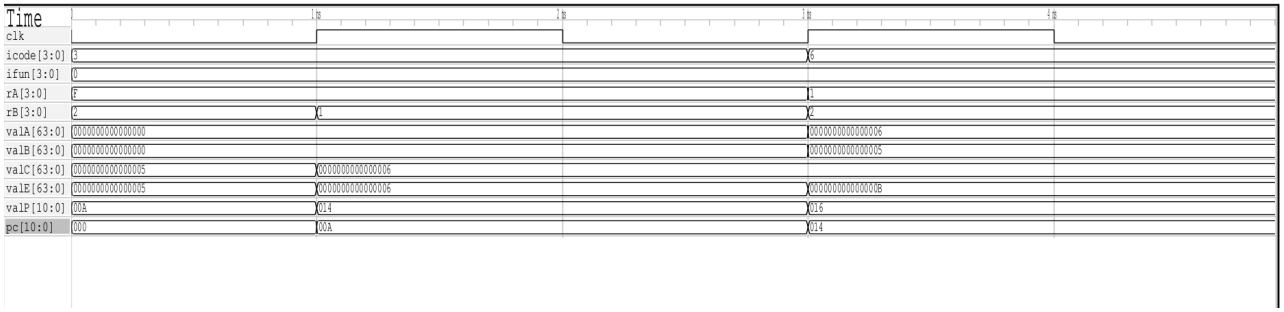
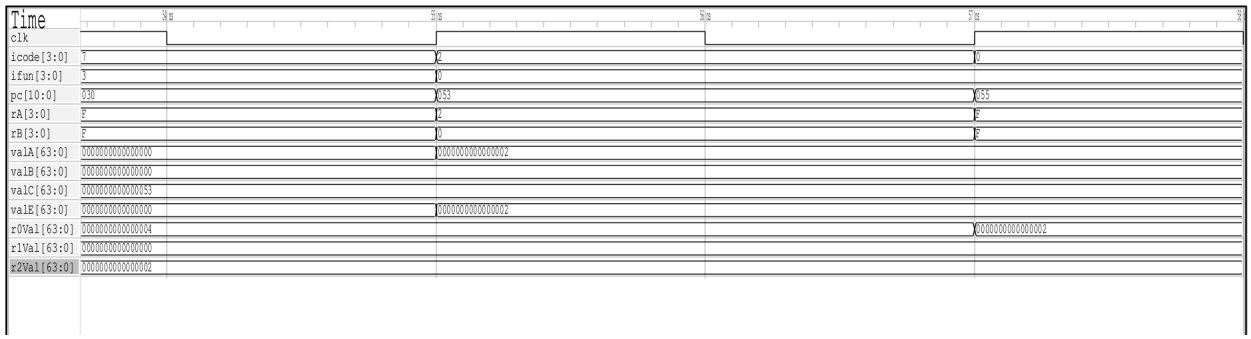


Fig - 10

The final contents in the register file are -



Flg - 11

Processor Features -

Every Y86-64 instruction processor instruction will complete its execution in the range of picoseconds. In the Y86-64 sequential design I choose my clk with 2nano seconds clock cycle. By this time all stages complete their execution and get settled to their final states.

Processor Frequency - 1/clock cycle time = 0.5*10^9 Hz = 0.5GHz

Instructions memory size - 1000 bytes

Register file sze - 16 reg with each 8 bytes - 128bytes

Data memory size - 1024 rows with each row of 8 bytes - 8KB

We can increase all the instruction, register, memory files by increasing the size of the 2-D array in the verilog file.

Instruction to run the processor

Sequential Y86 processor

To run the sequential Y86 processor, the first step is to write the instructions in the instructions memory.

For writing the instructions, open the *instr_mem.txt* in the *Sequential* folder and start writing the Y86 instructions that need to be executed by separating each byte by a space or new line. Some of the instruction files that I run are in the *instruction_file* folder, by copying and pasting these instructions in the *instr_mem.txt* we can run those runs.

Make sure that while entering 8 byte value, byte reversed order(as described on page - 12). Then run the *seq_processor.v* file using the iverilog command.

We can observe the outputs for each instruction at each stage and reg_file contents after every instruction.

For observing the outputs in gtkstage run the *seq.vcd* using gtkwave.

For testing each module(stage). Go to each folder and run the *stage_tb.v* file.

For testing each module the values to each module(stag) needs to be changed in the test_bench file of that corresponding module.

For running the hcf program instruction copy the instructions of the *hcf_instr.txt* to *instr_mem.txt* and then change the values of *a* and *b* then run the seq_processor.v file.

Pipeline processor

In the sequential processor, we can observe that if any module finishes its execution it will remain at rest until the next instruction is fetched or its previous blocks complete their execution. We can see that we are wasting the resources by running them only for some amount of time in a clock cycle. In a pipelined architecture every block will start their execution at every clock cycle(which is less time period compared to sequential). In the pipelined architecture we move our pc increment to the fetch stage. Once fetch outputs its values its start reading the next pc instruction at the next clock stage. Every other block will work in a similar way, they send their outputs to the output register and start reading the values from its input register at the rising edge of the clk.

Fetch (in pipelined)

The fetch module block diagram is as shown in figure - 12.

The PC update will now be written in the fetch block so that it does not need all stages complete their execution.

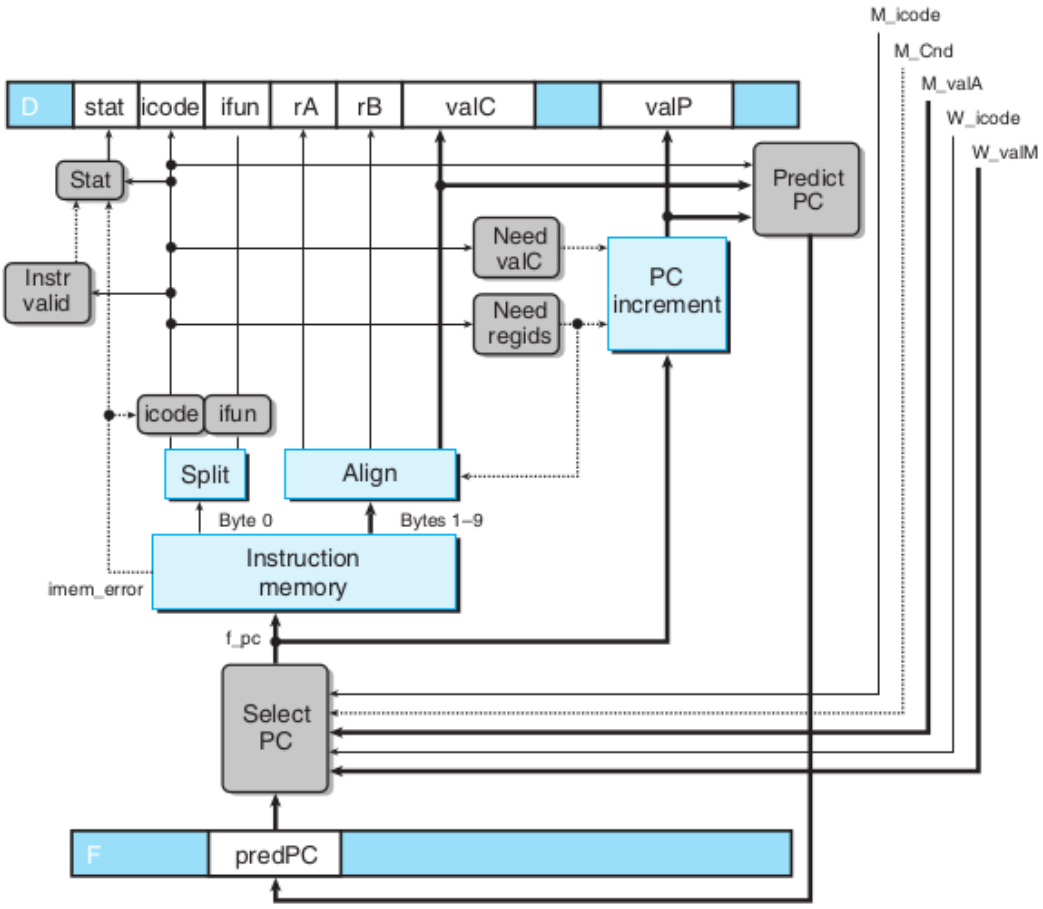


Fig - 12 Fetch module block diagram.(for pipelined)

Decode - writeback

The Decode -writeback module is implemented based on the block diagram written in Fig - 13. All the data forwarding connections are now taken that are coming from execution and memory and write_back stage. The forwarding logic for A and B are chosen based on the priority order they need to be taken.

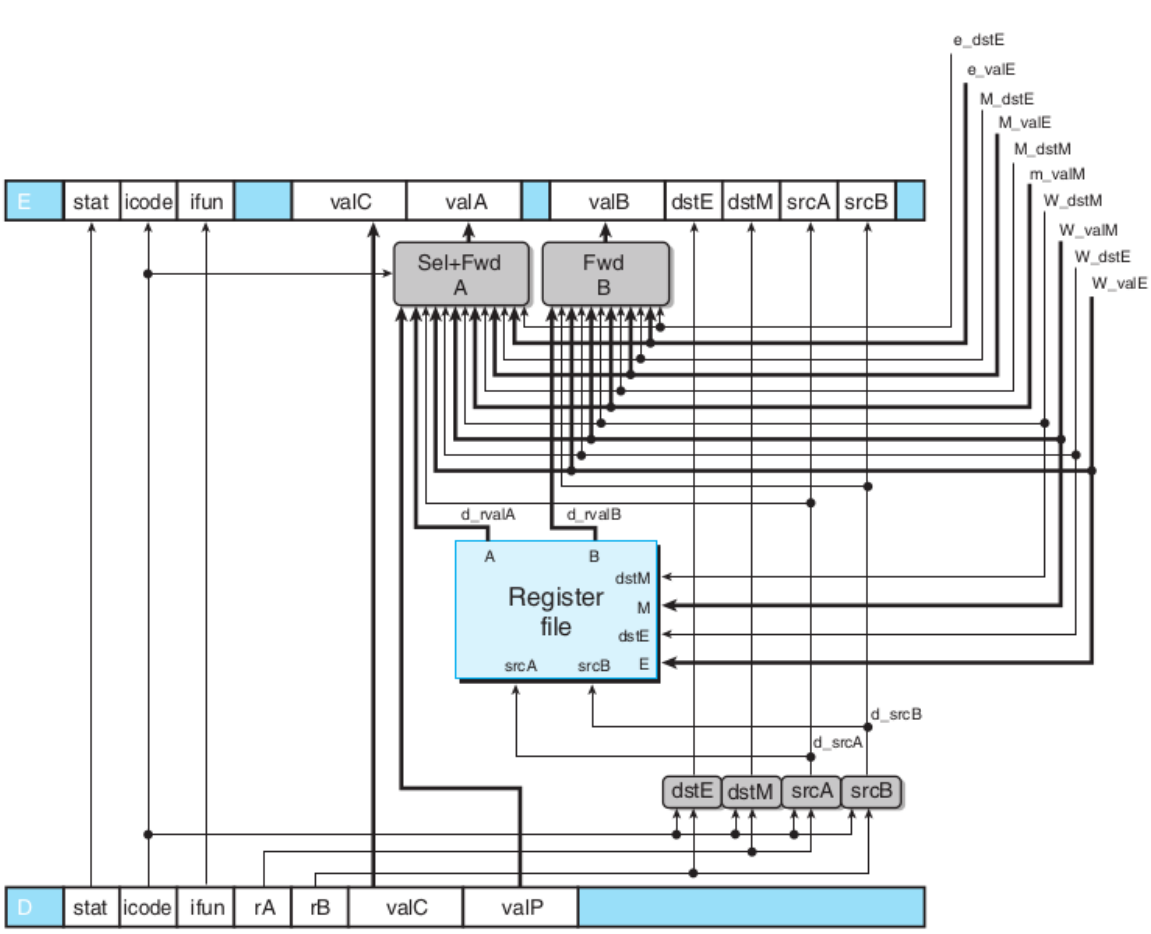


Fig - 13 Decode - write back module for pipelined architecture.

Execute

The block diagram for execute stage is as shown in FIg- 14. For the execute module the inputs are taken from the *M* register at rising of the clk signal and write the output to the W register. Here we also forward the output of the execution stage to the decode module. These values are used in the decode module.

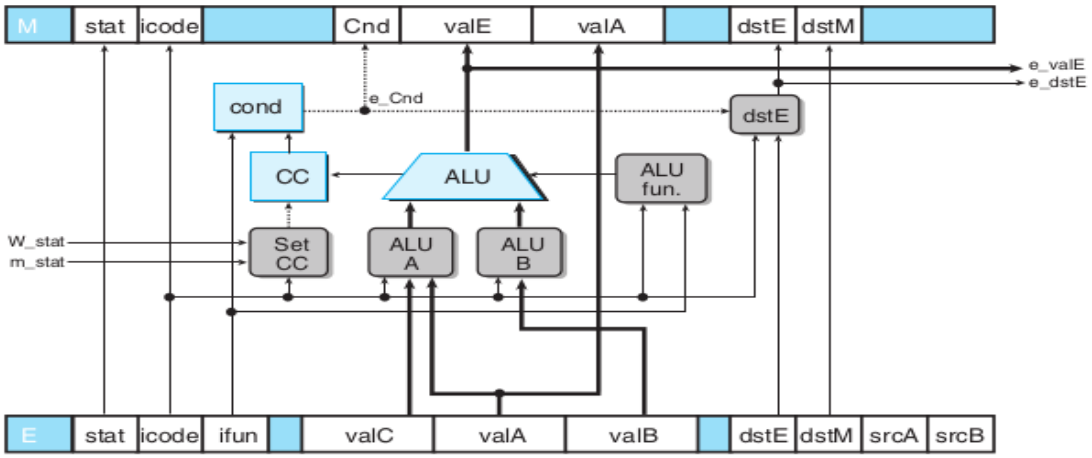
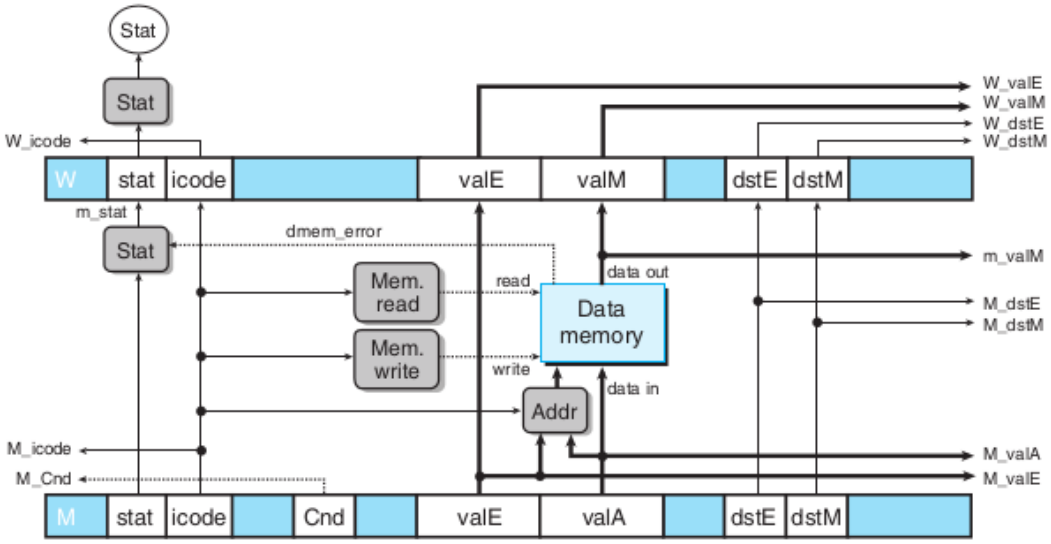


Fig - 14 Execute stage

The fetch, decode, execute stages are tested and integrated . These blocks can be executed using *execute_tb.v* in the execute block of the pipeline folder. The memory module is written.

Memory stage -

The block diagram for the memory stage is as shown in Fig - 15. In this module, we take inputs from the E register and wrti


Stat
Stat
W_icode
W
stat
icode
valE
valM
dstE
dstM
W_valE
W_valM
W_dstE
W_dstM
m_stat
Stat
dmem_error
Mem. read
read
data out
Data memory
m_valM
M_dstE
M_dstM
Mem. write
write
data in
Addr
M_icode
M_Cnd
M_valA
M_valE
M
stat
icode
Cnd
valE
valA
dstE
dstM