Team Name: Disconnected

Designing a Y86-64 processor

Pranav Manu (2020112019)
Rohan Reddy Bodgam (2020102039)

To compile:

Sequential:

```
iverilog -o <name> testbench.v -I ALU_Logic -I Data_mem/ -I Fetch_Logic/
-I PC_update/ -I Pipelined/ -I RegisterFile/
```

Pipelined:

```
iverilog -o <name> testbenchPipe.v -I ALU_Logic -I Data_mem/ -I
Fetch_Logic/ -I PC_update/ -I Pipelined/ -I RegisterFile/
```

Sequential Hardware Implementation:

The sequential hardware implementation consists of 6 stages.

- Fetch
- Decode
- Execute
- Memory
- Write-Back
- PC update

Fetch:

- The fetch block is the first stage in sequential implementation which reads the given instruction and sends it to the next stage in the required input form.
- Each instruction given will be in the form of 10 bytes from the instruction memory according to the value of the PC in which the first byte is always read as he instruction code and instruction function each read from 4 MSB (bits at position 7 to 4) add 4 LSB (bits at position 3 to 0 where the byte is read from 7 to 0) of the first byte respectively and it is determined to be valid or not depending on the value of the instruction code given. Using the icode, the required length of the instruction is known.

- The second byte is valC or addresses rA and rB depending on icode, and the value of need_regids is set. • There will also be a 64-bit value PC which gets incremented according to the value of need_regids (if rA and rB are required to carry out the instruction) and need_valC(if valC is required to carry out the instruction) represented in mathematical terms as valP = PC + need_regids + 8*need_valC.

Decode:

- The decode stage is done in the register file (RegFile module) which takes the inputs icode, rA and rB.
- The register file stores fifteen registers and the output of the decode stage will be valA and valB whose values will be the values stored in register rA and register rB respectively in most of the instructions given but in some cases it may be equal to the register with stack pointer which is 4. Combinational circuit decides the value of srcA, srcB, dstM and dstE.
- The valA and valB will be given as outputs according to the values of icode, rA and rB and the values stored in the 15 registers present in the register file

Execute:

- The execute stage is the place where the ADD, SUB, AND, XOR operations take place in the ALU according to the values of icode, valA, valB and valC and for some values of icode the condition code is calculated which some of the next stages depend on.
- The output of the execute block is valE and condition code, which is set as a side effect of arithmetic or logical operation.
- ALU_fun wrapper module is used to initialize the cnd depending on the icode value. The condition code registers are updated only at the positive edge of the clock.

Memory:

- In the memory block, the values of valA, valP, valE and valB are taken as inputs and they are written to the memory or read from the memory at certain indexes to get valM as the output.
- Each value in the memory is stored in terms of 8 bits which means the 64- bit values are stored in 8 successive cells of 8 bits each when being written to the memory and likewise when the memory is being read to get valM, it reads 8 successive cells of 8 bits each and transforms it to a 64-bit value.
- Data is written onto the memory only at the positive edge of the clock cycle.

- A data memory wrapper module is used to decide whether reading is required or writing, and from which address. The data in memory is stored in little endian format Write-back: In write-back stage, the values of valM or valE will be written/updated in registers rA or rB or 4(stack-pointer) at positive edge of the clock, depending on the instruction code given to execute. The writeback happens in the RegFile module.

PC update:

After every instruction is completed, the PC gets updated in the PC update stage which will be valP, valC or valM depending on the instruction code given and the condition code acquired in the execute stage for some instructions. A wrap module is used to initialize all the other modules and provide each module with the correct input signals

Instructions working:

- halt
- nop
- cmovXX
- irmovq
- rmmovq
- mrmovq
- opq
- jXX
- call
- ret
- pushq
- popq

Instructions given and their outputs:

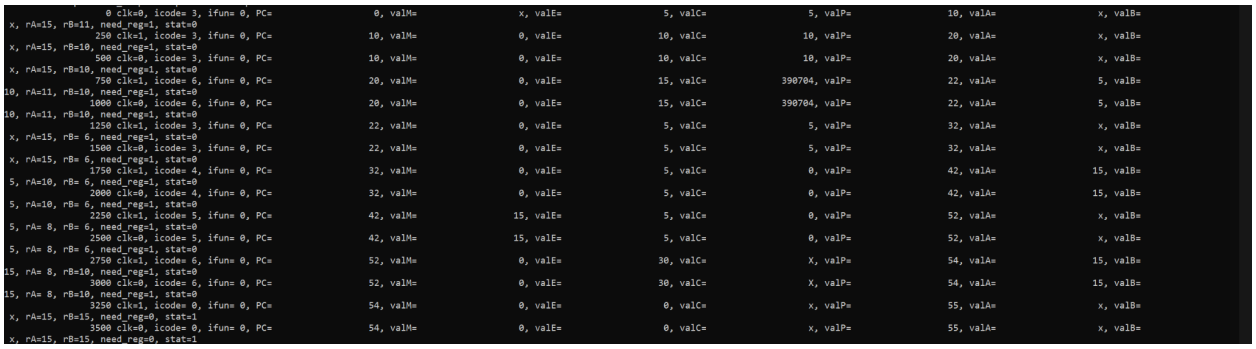
Instruction set 1: (rmmovq and mrmovq)

Code:

```
irmovq $5,  %r11
```

```
irmovq $10,  %r10

addq %r11, %r10

irmovq $5, %rsi

rmmovq %r10, 0(%rsi)

mrmovq 0(%rsi), %r8

addq %r8, %r10

halt
```

Output:



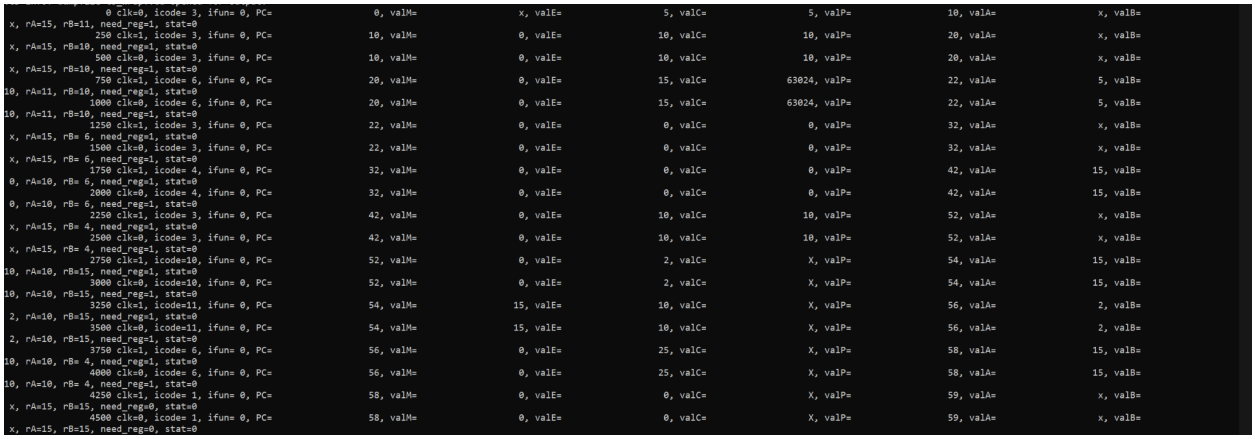
Instruction set 2: (push and pop)

Code:

```
irmovq $5, %r11

irmovq $10, %r10

addq %r11, %r10

 irmovq $0, %rsi

rmmovq %r10, 0(%rsi)

irmovq %10, %rsp

pushq %r10

popq %r10
```

```
addq %r10, %rsp

nop

halt
```

Output:

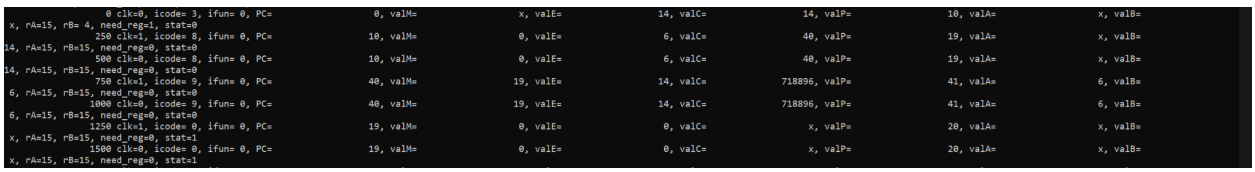


Instruction set 3: (call and ret)

Code:

```
irmovq $14, %rsp

call 40

halt

Ret   //At 40

irmovq $10, %r8

irmovq $2, %rsi

irmovq $3, %r12

halt
```

Output:

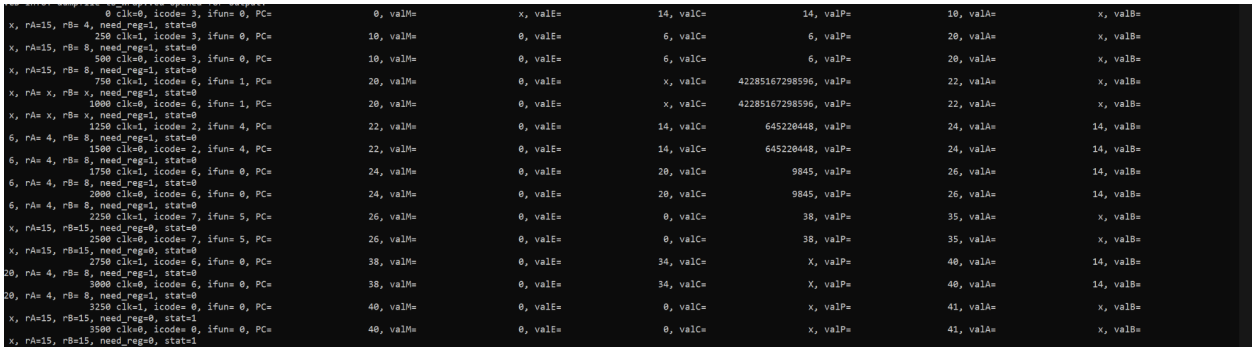


Instruction set 4: (working jump and cmov)

Code:

```
irmovq $14, %rsp

irmovq $6, %r8

subq  %rsp, %r8

cmovne %rsp, %r8

jge 38

(at 38) addq %rsp %r8

halt
```

Output:



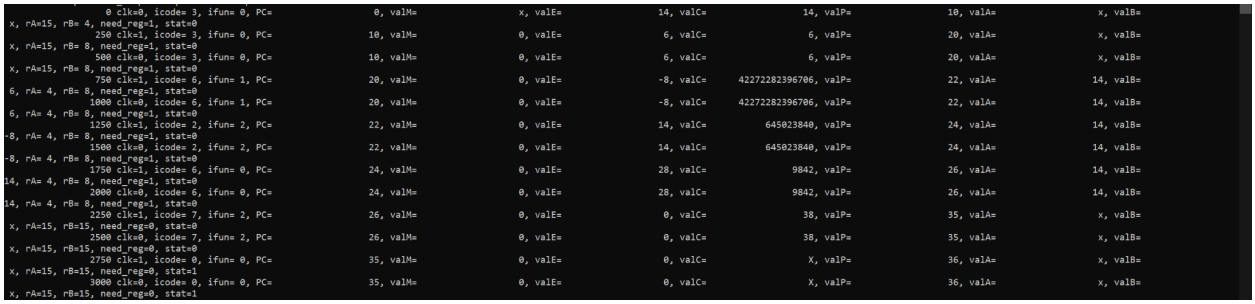
Instruction set 5: (no jump)

Code:

```
irmovq $14, %rsp

irmovq $6, %r8

subq  %rsp, %r8

cmovne %rsp, %r8

jl 38

halt
```

```
(at 38) addq %rsp %r8

halt
```

Output:



Pipeline Hardware Implementation:

Pipeline Hardware Implementation has the same working blocks of the sequential hardware, but the way of working is different in between both. As defined by name, sequential implies that each instruction starts when the preceding instruction has been completed the whole cycle whereas in pipeline hardware, the next instruction starts implementation when the preceding instruction passes through the fetch block and goes to decoding stage which means in pipeline processor, all the stages can contain instructions at a time. But as the instructions we give might depend on each other, the second instruction won't get the values written by the first instruction until the first instruction completes the process. Also, we need to predict the PC here when jump and return instructions are used which causes an error as the next instruction will reach at least the execute stage when the instructions get completed which is also an error most of the time. These data dependencies cause errors, which need to be modified to run the given data. The errors possible are data hazards, mispredicted branch and processing ret instruction. All the instructions here can be easily modified using 3 or more 'nop' instructions which do nothing while the instructions which cause the errors get processed through all the stages. But if we do not wish to give at least 3 'nop' instructions for our data(as every instruction increases the delay and we want it to be as less as possible), we need to use a logic which is data forwarding, only possible for some cases of data hazards and stalling the instruction so that the instruction stops in some block while its preceding instruction get processed to the next stage. Then we can data forward in the data hazard cases when we have the prior information of the previous instruction we want and stalling more if required in the mispredicted branch and processing ret cases.

Implementing data forward:

Data forward is done in the decode stage using select+forward A and forward B which reads the required values either from the register rA and rB or the values written by the previous instructions to the register whose values are stored in e_dstE, M_dstE, M_dstM, W_dstE, W_dstM where e_dstE being the special case which depends on the cnd value of the instruction in execute stage. The priority is given in such a way that the value from the nearest stage is read when there are multiple cases of different values written to the same register in different stages (from execute to write back).

Handling the data hazards:

But there will be some limitations for data forwarding in cases where there will be no value to be read from the next stages. To overcome this error, we use stalling and injecting bubbles in between the two instructions. When we do this, the instruction gets stopped in that stage where it is in whereas the previous instructions go to the next stages thereby writing the values in the registers we want to read or jumping to a specific instruction or returning to the next instruction of call.

The process of stall and bubble is done for the errors in different stages as:

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use hazard	stall	stall	bubble	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal

Changes made in sequential blocks to accommodate in pipeline hardware:

Added pipeline registers in between which hold the value of the instruction at each stage.

The Regfile module is split into two modules, one containing the register file used in sequential hardware and the other is done using the logic of srcA,srcB,dstE and dstM.

In sequential hardware, stat was getting calculated/evaluated only in the memory stage but in pipeline hardware, the value of stat is calculated for every instruction when it passes through the fetch stage and it depends on the values of f_icode, f_imem_error and f_instr_valid. The final stat value of the instruction gets calculated when it reaches the memory stage using the m_dmem_error and the stat value originated/calculated in

the fetch stage. Using the output of datamem_wrap module, m_rstat output of datamem_wrap depends on dmem_error, icode, imem_error and instr_valid calculated in the sequential hardware and then the d_rstat and the originated stat from fetch stage will act as inputs to the stat block in memory stage in pipeline hardware to give the final stat value of output.

e_dstE is another block added in the execute stage of pipeline hardware that forwards the value of E_dstE only if cnd is true in the case of cmovXX instruction. Otherwise the value of e_dstE will always be forwarded to the M pipeline register as E_dstE.

To represent the conditional circuits:

In combinational logic, always @(*) is used in some cases because if, else and case blocks can only be used within procedural blocks.

Instructions working:

- halt
- nop
- cmovXX
- irmovq
- rmmovq
- mrmovq
- opq
- jXX
- call
- ret
- pushq
- popq

Instructions given and their outputs:

Instruction set 1: (Data Hazard)

Code:

```
irmovq $5,  %r11

irmovq $10,  %r10
```

```
addq %r11, %r10

irmovq $5, %rsi

rmmovq %r10, 0(%rsi)

mrmovq 0(%rsi), %r8

addq %r8, %r10

halt
```

Instruction set 2: (Push and Pop)

Code:

```
irmovq $5, %r11

irmovq $10, %r10

addq %r11, %r10

 irmovq $0, %rsi

rmmovq %r10, 0(%rsi)

irmovq %10, %rsp

pushq %r10

popq %r10

addq %r10, %rsp

nop

halt
```

Instruction set 3: (processing ret)

Code:

```
irmovq $14, %rsp
```

```
call 40

halt

(at 40) ret

irmovq $10, %r8

irmovq $2, %rsi

irmovq $3, %r12

halt
```

Instruction set 4: (working jump and cmov)

Code:

```
irmovq $14, %rsp

irmovq $6, %r8

subq  %rsp, %r8

cmovne %rsp, %r8

jge 38

(at 38) addq %rsp %r8

halt
```

Instruction set 5: (mispredicted branch)

Code:

```
irmovq $14, %rsp

irmovq $6, %r8

subq  %rsp, %r8

cmovne %rsp, %r8
```

```
jl 38

halt

(at 38) addq %rsp %r8

halt
```

The outputs of both the sequential processor and pipeline processor for the given sets are too big/hold enormous data to be shown using screenshots. Hence, the outputs are submitted in a different folder '**InstructSet_and_OUTPUT**' according the the naming of the instruction sets.

Paste the required instruction in 'inst_mem.v', after removing the instructions already present there.

NOTE: When the status code is not AOK, then the register files stop updating, except the fetch register, as it is not stalled according to the stall condition given.

Challenges Faced:

Using the same blocks from sequential in pipeline: Some subtle changes were needed to make the modules, used in sequential, usable in the pipeline implementation. These changes were mentioned above.

Memory Size: The compile time was too long for large memory size, so we had to reduce it.

Condition code setting: Condition code and Cnd were needed to be correct so that cmov worked correctly.

Making the instructions: Converting y86 code to binary and then coding it into the instruction memory was a tedious process.