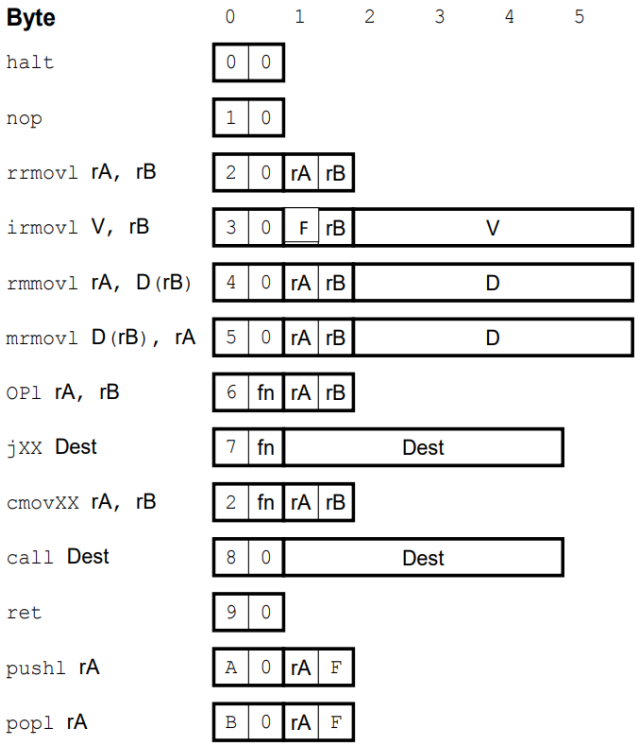
CSCE 312 Project: Pipelined CPU for Y86 ISA in Logisim

By Ananth Madan, Section 503

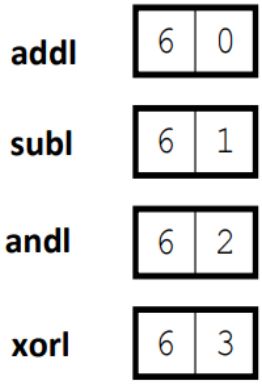
ISA Description

I implemented a CPU that runs code assembled using the Y86 instruction set:



The branch conditions are the same as the Y86 versions:

(The above pattern is the same for the conditional move operation.)
Similarly, my arithmetic operations are the same as Y86's:



My register file is indexed the same as in Y86:

%eax	0
%ecx	1
%edx	2
%ebx	3
%esi	6
%edi	7
%esp	4
%ebp	5

Test Code

I wrote three programs to test my CPU:

1. array_sum: a program that finds the sum of an array in memory
2. mat_add: a program that adds two matrices together and writes the resulting matrix to a destination matrix in memory
3. selection_sort: a program that sorts an array in memory using a variant of the selection sort algorithm

Together, these programs use the entire Y86 instruction set and trigger both data and (all) control hazards, and therefore are a good, basic test suite for determining the functionality of my CPU. They are found in (hopefully well-named) subdirectories in the “testcode” folder.

ASM-to-RAW Conversion

I wrote a python script to convert a .yo file to a raw ROM file, called “yo2rom.py”. It can be done using the command line, as in executing “python3 yo2rom.py [--dst path_to_raw] path_to_yo” on the command line. This will convert the .yo file at “path_to_yo” into a raw Logisim ROM file (which can optionally be specified by “path_to_raw”, but will default to the same path-name as path_to_yo minus the “.yo” suffix).

Expected Behavior of Test Code

The expected behaviors of the files are given in “expected_behavior.png” files alongside their respective test code in the “testcode” subdirectory. These .png files are outputs of the ./yis command in the Linux VM — the ./yis command displays the changes to registers and memory after the program is completed.

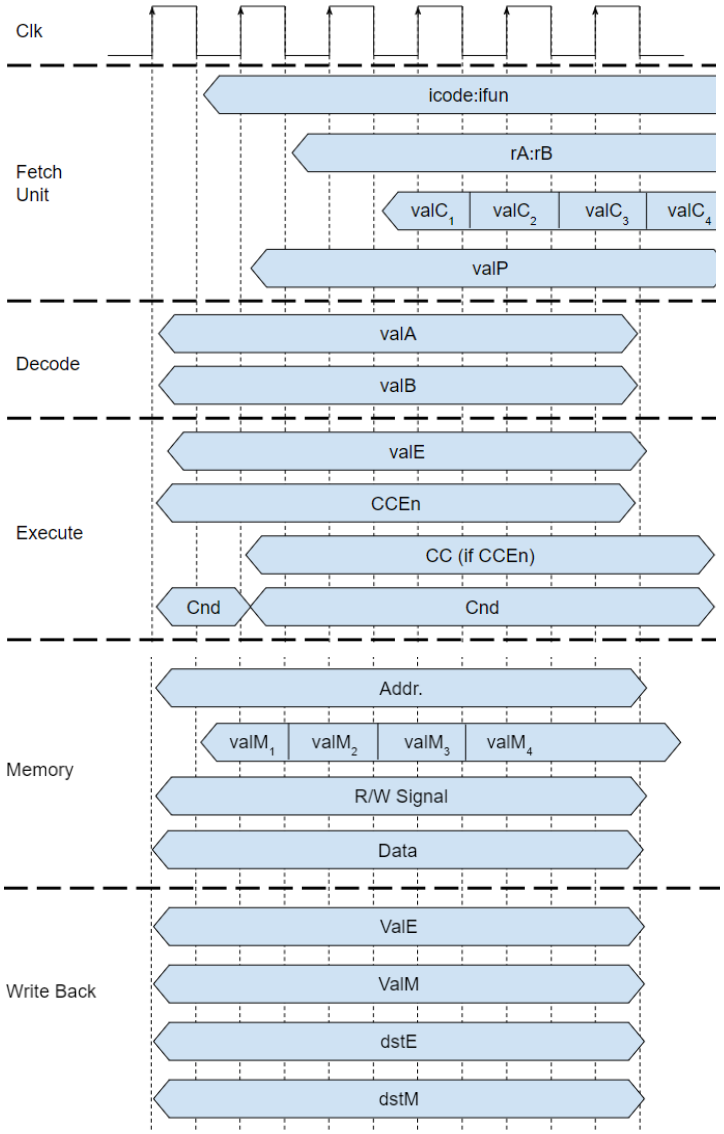
(Note that for each program, it seems like the %ebp value on the stack — the first value on the stack — is not always the actual top of the stack. I assume this is from the internals of the Y86 runtime, but I could be wrong. This seems to impact nothing, as subsequent values using %ebp and %esp work perfectly, so I assume this is just a pedantic difference.)

Logisim Circuit File

The Logisim circuit file is simply one file labeled “CPU.circ”. The main circuit is called “main”, and this circuit contains the actual CPU. The other circuits contain all the necessary components for the CPU to run and execute Y86 code.

Timing Diagrams

The following diagram shows the timing diagram for the various important signals in the various units of the CPU:



There are many other individual signals as well, but these are all the important signals that dictate the micro-operations of the CPU. Also, all the buffers between the stages read and write every 6 cycles, on the rising edge of the 1st cycle.

Expected Inputs and Outputs

The CPU can be run using the following steps:

1. Reset the state using Ctrl+R
2. Loading the raw code file into the Instruction Memory
    a. Instruction Memory is the ROM module in the component labeled Instruction Memory, which can be accessed after double-clicking on the Instruction Memory component in the main viewer
3. Loading the same code file into the Data Memory
    a. Data Memory is the RAM module in the component labeled Data Memory, which can be accessed in the same way, just for Data Memory
4. Pressing Ctrl+K to run the code
5. Pressing Ctrl+K to stop the clock once the halt signal goes high
    a. The signal goes high when the furthest-left green wire turns bright

The processor has many components and subcomponents, but the main ones — their operation and expected inputs and outputs — are what this section will focus on.

Global Count (Cnt):

The main purpose of Global Count is two things: output buffer transfer timings, and output the clock. Global Count has two inputs: *halt* (the halt signal) and *clk* (the clock signal). Global Count has four outputs: *Cnt*, *cntIsFive*, *clk*, and *cntIsZero*. If *halt* is low, then the clock (*clk*) signal will pass through unchanged. The counter will cycle through the numbers 0, 1, 2, 3, 4, and 5 each clock cycle, and corresponding boolean signals (*CntIsFive* when *Cnt* is 5, and *CntIsZero* when *Cnt* is 0) will be evaluated If *halt* is high, the *clk* output signal will be 0 and *Cnt* will not be updated.

IF:

IF is a buffer that stores the next, predicted PC value. Every 6th clock cycle, IF will store the updated, predicted PC and forward that to the Select PC component. IF has four inputs: *PredPC* (the predicted PC), *clk*, *cntIsFive*, and *stall*. The clock cycle (*clk*) triggers the internal register to update (of which there are two, one of which forwards the value to the forwarding register, which will then update its value to what is being forwarded only on the 6th clock cycle). *cntIsFive* controls when the forwarding register updates: the forwarding register is a falling-edge-triggered register whose clock is *cntIsFive*, meaning when the 6th clock cycle happens (and *cntIsFive* becomes low again), the register stores the predPC and forwards it. *stall* simply deactivates this process from happening, causing a stall (which is important for hazards).

Select PC:

Select PC selects which PC to use: the predicted PC value, or one from the Memory and/or Write Back stages (which are stored and forwarded via the stage buffers). Select PC has six inputs: *PC_pred* (the predicted PC forwarded by IF), *M_icode* (the icode currently at the Memory stage), *M_cnd* (the condition currently at the Memory stage), *M_valA* (valA currently at the Memory stage), *W_icode* (the icode at the Write Back stage), and *W_valM* (valM currently at the Write Back stage). Select PC has only one output: the PC. Select PC checks *M_cnd* is false (low) and *M_icode* is 7 (a jump instruction); if this is the case, then the PC becomes *M_valA*, as *M_valA* was once *valP* (as we see later, this PC uses an AT approach for branch prediction).

This deals with mispredicted branches. For returns, we see if *W_icode* is 9; if it is, the PC becomes *W_valM* (which is the PC value pulled off the stack). If none of those are the case, we simply forward *PC_pred*.

Instruction Memory (Fetch Unit):
This component actually fetches the instructions from the instruction ROM based on the count and PC. Instruction Memory takes in four inputs: *clk*, *Cnt*, *cntIsZero*, and *PC*; it outputs: *icode*, *ifun*, *rA*, *rB*, *valC*, *valP*, *PC*, and *Length* (length of the current opcode). Instruction Memory reads 6 times from memory and fills up two buffers — one contains *icode*, *ifun*, *rA*, and *rB*, and the other contains *valC* — while reading to get the appropriate values. It adds *Cnt* to *PC* to get the current address from which to read, and uses *icode* (which is gotten on the first cycle) to determine the *Length* and, therefore, the max address from which to read; if *Cnt* plus *PC* is larger than the max address, the PC replaces *Cnt* with *Length* to get the current address (which will be the same as the max address), and reads from there until the 6 cycles are up.

PC++ (PC Increment):
PC Increment just adds PC and Length to get the next PC value. It takes in *Old PC* and *OpLen*, and it returns the sum as *valP*.

Pred (Predict PC):
Pred just chooses a PC value depending on *icode*. It takes in three inputs: *icode*, *valP*, and *valC*, and uses them to predict its output, *PC*. If *icode* is 7 (jmp) or 8 (call), Pred chooses *valC* as *PC* (it assumes jump and call will always be taken, which is true for call and is known as the always taken strategy for branch prediction); otherwise, it chooses *valP*.

IF/ID:
IF/ID is the buffer between the Fetch and Decode stages. Every 6th clock cycle, it updates the internal storage and forwards the previous values to the next stage like IF does. Its inputs (and outputs) are *icode*, *ifun*, *rA*, *rB*, *valC*, and *valP*. Like IF, IF/ID can be stalled using the *stall* flag, which acts the same as it does for IF; IF/ID also can be bubbled using the *bubble* flag. When bubbled, IF/ID takes in the previous stage's outputs, but instead of forward things, it forwards a NOP instruction (*icode* is 1, addresses are Fs).

srcA & srcB & dstE & dstM:
These components choose what register files will be read and written from (into valA and valB now, and by valE and valM later, respectively); they take in *icode* and *rA* (srcA and dstM) or *rB* (srcB and dstM), and output signals bearing their names.

icode	srcA	srcB	dstE	dstM
0	0xF	0xF	0xF	0xF
1	0xF	0xF	0xF	0xF

2	rA	0xF	rB	0xF
3	0xF	0xF	rB	0xF
4	rA	rB	0xF	0xF
5	rA	rB	0xF	rA
6	rA	rB	rB	0xF
7	0xF	0xF	0xF	0xF
8	0xF	0x4	0x4	0xF
9	0x4	0x4	0x4	0xF
A	rA	0x4	0x4	0xF
B	0x4	0x4	0x4	rA

Register File (Decode Unit):
This is a Register File that reads from the registers at *srcA* and *srcB*, and writes *valM* and *valE* to *dstE* and *dstM*, respectively, on the inbound ports (these signals are from the Write Back stage, not the Decode stage). The Decode Unit will only write to valid registers: any values corresponding to addresses above 0x7 (unsigned) will be discarded. The unit also outputs all the registers purely for display purposes — only *valA* and *valB* are forwarded to other components.

Fwd B (Forward B):
This component uses logic to select which value to forward for valA if there is a data hazard; it sees if *srcB* is equivalent to any of the *dstM* and *dstE* signals currently in Execute, Memory, or Write Back in that order; the first equivalent address found, and its corresponding address, will be forwarded as *valB*; in none are found, the original *valB* will forwarded. An important implementation detail is that any invalid addresses (such as 0xf) will not activate a forward, as these addresses are used by the CPU to specify dummy addresses for any destination addresses that won't actually be required (e.g. an opq instruction won't require *dstM*, so *dstM* will be 0xf).

Fwd A (Forward + Select A):
This component acts similarly to Forward B, but it acts using *srcA* and upon *valA*, and it also determines whether *valP* should be forwarded instead of any other value; *valP* is chosen when *icode* is either 7 or 8). Again, forwarding is ignored for any invalid addresses above 0x7 for similar reasons.

ID/EX:

This is the buffer between the Decode and Execute stage. It acts the same as the other buffers, except this one supports only bubbling (NOP replacement).

En (CCEn):
This component determines whether the ALU should set the CCs. It only returns true (i.e. its output is high) when the *icode* is 6 (any op instruction).

CC:
This component stores the condition codes (CCs) inside via registers. It updates the registers only when CCEn is high; else, it keeps its previous state. The CCs are obtained from the ALU.

Cnd:
This component determines the boolean value of any conditional operation using the following logic:

ifun	Boolean Expression
0	True
1	(!SF && OF) || (SF && !OF) || ZF
2	(!SF && OF) || (SF && !OF)
3	ZF
4	!ZF
5	(SF || !OF) && (!SF || OF)
6	(SF || !OF) && (!SF || OF) && !ZF

This component is always active, because any other component that utilizes *Cnd* for its logic checks for the *icode* to be one that requires a conditional value (either 7, jump, or 2, compare move).

ALU A & ALU B:
These components determine which values to forward to the ALU, using the *icode* and some logic:

icode	ALU A	ALU B
0	valC	0x0
1	valC	0x0
2	valA	0x0

3	valC	0x0
4	valC	valB
5	valC	valB
6	valA	valB
7	valC	valB
8	0x-4	valB
9	0x4	valB
A	0x-4	valB
B	0x4	valB

ALU:
The ALU returns either the sum of the arguments A and B or the value of the specified operation on A and B (or just A), depending on *icode* and *ifun*. Operations performed in the ALU will return a set of CCs (OF, ZF, SF), regardless of what the *icode* is; the CCs will only be stores, however, if the CCEn unit outputs a high signal, which only happens when *icode* is 6 (i.e. for an arithmetic operation). The result of the ALU operation is outputted as *valE*.

cmov (cmov Addr. Handler):
This component determines whether a cmov instruction will be carried out or not: if the boolean flag is high, then the address passed in as *dstE* will remain unchanged; else, the address will be replaced with 0xf (which is an invalid, "dummy" address). This unit will only activate if the *icode* is 2 (a compare move).

EX/MEM:
This is the buffer between the Execute and Memory stage. It acts the same as the other buffers, except no control logic is supported: it only stores and forwards values on the 6th clock cycle, without any possibility of stalling or bubbling.

Addr (Mem. Addr.):
This component simply selects which value, between *valE* and *valA*, to use as the address from/to which data will be read/written in Data Memory. It operates using the following logic: Choose *valA* if *icode* is either 9 (ret) or B (pop); else, choose *valE*.

WR Sig (Mem. Sig.):
This component determines whether Data Memory will be read from, written to, or neither. It operates using the following logic: *write* is high if *icode* is 4 (rmmov), 8 (call), and A (push); *read*

is high if *icode* is 5 (mrmov), 9 (ret), and B (pop). Note that there exists no valid case where Data Memory would be both read from and written to.

Data Memory:
Data Memory is where data is read from and/or written to, depending on the output of WR Sig. The component uses *Count*, which is the signal outputted by Global Count, to determine the offset from *Addr* from which to read and write (data is in little-endian order, so this means we can use the unmodified count as an offset). Since RAM can only read 1 byte at a time, we also use count as a mux selector that, after splitting our *Data* into four 1 byte chunks, forwards our *Data* appropriately. Depending on the output of WR Sig, we forward either *Data* or data from our Data Memory into *valM* (using a 4x1B register file to store the individual chunks). When count reaches 4, we disable RAM, because it should only take 4 clock cycles to read from the RAM module (which, in practice, is very much not the case, but it works in Logisim…)

MEM/WBN:
This is the buffer between the Memory and Write Back stage. It acts the same as EX/MEM.

Halt:
This component determines when to stop the CPU, by issuing a halt flag which, when said flag goes high, will cause Global Count to stop forwarding the clock, and instead forward a low signal. This is done by incrementing a counter every 6th clock cycle when the input *icode* is 0; when the counter reaches 5, the halt flag is issued. The reason the counter needs to count for 5 is upon start up, the CPU's buffers are all 0 by default, meaning it'll take 4 iterations of 6 cycles before any actual instruction reaches the Halt unit (because there are 4 stages between the Halt unit and the Fetch unit).

Hazard Controller (Control Hazard Handler):
This component deals with control hazards by, for various combinations of *icodes* and input signals, raising stall and bubble flags to the appropriate buffers at appropriate times. The hazard controller raises flags according to the following rules:

Condition	Trigger
Processing ret	IRET in { D_icode, E_icode, M_icode }
Load/Use Hazard	E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }
Mispredicted Branch	E_icode = IJXX & !e_Bch

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

Combinations of hazards may occur — namely, a load-store hazard could occur with both a mispredicted jump and a return — but we can use simple precedence rules to coerce desired

behaviors in these situations: namely, stalling takes precedence over bubbling, and bubbling takes precedence over doing nothing.

Individual Report

I worked on a pipeline CPU individually. All of the components and logic were designed by me, using principles of pipelining learned in class and from Chapter 4 of Computer Systems, by Byrant. Specifically, I referenced the HDL code in Bryant's book to build specific components, such as the PC selection and prediction logic and the forwarding logic — however, the implementation details were designed by me, without any references to pre-existing implementations. I referenced the flag tables for the design of the control hazard unit, as well as the tables for desired behaviors for combinations of control hazards, but again, all of the implementation was done by me. For the Halt, Memory Read/Write, Conditional, and Clock components, I used my knowledge of the desired behavior, as well as some empirical and theoretical reasoning, to design working components that fit my CPU design; for Halt, I just made a simple counter that outputs a halt signal when it reads 5 halt codes, simply because 4 halt codes will pass through regardless because of the number of stages between the first Fetch cycle and the program start; Memory Read/Write, ALU A and B, and the src/dst components were designed via tables I made using the desired behaviors of each of the micro-operations based on a given opcode. The Control Code logic was made by observing boolean expressions of the conditional flags that made sense given a subtraction operation — we use subtraction because that is the best way to compare two values numerically, and we assume that if any comparison is done in code, it mainly utilizes numerical subtraction. The Clock is a simple clock with a 1-cycle offset to enforce a count of 0 lining up with the first clock cycle, instead of 1. The buffers were simple buffers that I spent a considerable amount of time fine-tuning such that they would store and forward values properly on the 6th cycle, regardless of hazards or jumps or a normal state. Each of the main components (Fetch, Decode, ALU, and Data Memory) were made in previous labs, but were improved upon throughout the course of the project. There wasn't much that needed to be changed for these components overall — the biggest change was fixing some timing offsets and oddities in the Fetch Unit, as well as fixing some big- vs. little-endianness in the Data Memory — but integrating them with the other pipelining components required some refactoring (notably of the Fetch Unit, which was split into many separate components, such as Select PC and Predict PC).

The most difficult part of this project was fixing timing mismatches between each stage, because a sub-cycle delay would throw off an entire stage's inputs and outputs, and from there, the program would be incorrect. Error fixes were usually simple, but also nontrivial, so while fixing errors was simple, finding where they occurred wasn't; furthermore, I feel like many of my fixes weren't the most elegant approach (such as the double-register buffers, which eliminated a lot of delay-based bugs in execution, especially pertaining to glitches and signal timing mismatches between error detection, the enable signal, and the clock signal), and so my design can definitely be streamlined. Furthermore, looking back at the book, my Fetch unit is a bit bulky and still possibly refactorable — the book splits up Fetch into many more components (such as aligning split bytes, determining their flows into registers afterwards, updating the PC without any instant reads or count signals, etc.), and I don't doubt many other components are as well.

Overall, my CPU is able to execute all y86 instructions, and I would guess that 95% of well-written code works on the CPU; furthermore, my CPU correctly implements pipelining, and executes around 5 times faster than my peers'. I learned a lot about pipelining and CPU design, and I am now much more interested in learning about lower-level design implementations, both in hardware and software.