University of Utah

CS 3710/ Fall 2015

The Claw Project

The X-Men:

Nathan Donaldson

Keith Madsen

John Mckay

Clint Wilkinson

The project that we created was a crane game. We built an xyz axis crane with 3 motors and placed in a square like platform. The objective of the game was to pick up as many things as you could in a minute and drop them in a designated area using a NES controller. We started from the ground up. We designed and implemented everything in our processor. First we designed the ALU, which performed operations on registers and immediates and saved the outcome in registers or memory. Then we designed the register file, which would be used by the ALU to do register to register operations or register to immediate operations. We then designed memory, which would be used by all portions of the processor. The memory held our instructions and other information needed for other parts of our program. After we finished that we created our processor controller. The processor controller was in charge of setting up all parts of the processor depending on the instruction coming from memory currently. The program counter was also implemented in this state, which keeps track of where we are in memory. All of this was done together, but in the final stretch of the project we each split up and focused on separate things and brought it all together. For our project, we had Clint working on the motor and NES controller implementation. Nathan and John worked on the VGA. And Keith worked on the Compiler/Program, as well as a few other extra modules to help with functionality of the project.

Physical Design/Hardware

Our ALU was basically the same as everyone elses ALU. There might have been subtle differences but for the most part they were all the same. Our instruction set consisted of:

ADD = 00000101

ADDI = 0101????

ADDU = 00000110

ADDUI = 0110????

ADDC = 00000111

ADDCU = 00000100

ADDCUI = 0001????

ADDCI = 0111????

SUB = 00001001

SUBI = 1001????

CMP = 00001011

CMPI = 1011????

CMPU = 00001000

CMPUI = 0010????

AND = 00000001

ANDI = 1010????

OR = 00000010

XOR = 00000011

NOT = 00001100

LSH = 10000100

LSHI = 1000000?

ASH = 01001111

ASHI = 1000001?

WAIT = 00000000

MOV = 00001101

MOVI = 0011????

The don't cares in these instructions were for immediate portions of each instruction's opcode. The don't cares in the logical and arithmetic immediate shifts were a sign instead of an immediate. 1 would make the value negative and 0 would make it positive. There were 2 extra opcodes we added at the end when nearing the deadline to get our final demo to work, and those were:

GET = 10000101

START = 00001111

ENCHECK = 10000111

These instructions were made specifically to get our stuff going. START would send a signal to the timer to start timing. GET would get the current time of the timer and store it in register 0. ENCHECK would do a comparison with an enable signal sent in and the current time to either send out a disable or enable signal to the motors.

Our register file was quite simple as well. We had 16 registers that held 16 bit values. The 16^{th} register was used for our flags when comparisons were being done. We didn't even use all of our comparison flags but that was no big deal in the end. The register file had two options for what was to be input into a register. Either the result from the ALU or the data coming out of memory port A would go into a specific register determined by a control signal. The control signal was just part of an instruction inside of memory that was sent to the FSM to decode and decide which register we would write to. We basically made two big buffers filled with smaller buffers to decide which registers to write to and which registers to read from. The registers we would read from would go to our bus B and bus A for the inputs to the ALU.

Our memory was very easy to set up at first. All we really did at first was read from memory and also set up direct wires into the addresses and data inputs and set the write and read enables high when we wanted to for testing purposes. Memory became a little more difficult when it came to implementing it with the rest of our CPU. Our memory has two ports, A and B. For each port we have an enable and write signal. Enable allows data to flow out of memory, so basically we can read from memory when enable is high. Write allows us to write to memory at a given address. We have an address input that sets which address we are jumping to or which address we are writing to. The FSM decides which address we are going to. We either go to an address from the bus B going into the ALU or to the address in the program counter. We also have an input that (when write is high) takes information on that input bus to the address that we are currently going to based on the address input.

The program counter is nothing but what keeps track of what address we are on in memory. If no operation that changes the PC is happening, then all we do is increment by 1 after an instruction finishes. If that doesn't happen, then we are doing a jump, branch, or JAL. So to take those into account we have a couple control signals for branch and jump. The jump control signal chooses whether to increment by 1 or jump to a specific address that is given in an address on ALU bus A. The branch control signal chooses whether to increment by 1 or increment or decrement by an immediate value. For the JAL we make sure jump is chosen and we save the current address. When the ret instruction comes that saved address plus 1 becomes the new address value.

The FSM controls every control signal and reset signals on everything. Based on the instruction output from memory we decide which signals to enable. The control signals are as follows:

Mem_weA – Writing to memory port A.

Mem_enA – Reading from memory port A.

Mem_addA_ctrl – Address we are reading from or writing to in memory. Which is PC or ALU bus B.

Link – Saves addr + 1 in register for JAL instructions.

PCen – Allows PC to increment.

ret – Sets PC output to the address saved in the register for JAL addresses.

Jump_ctrl – Controls whether or not we jump to address given by ALU bus A or just increment by 1.

Branch_ctrl – Controls whether or not we jump a certain amount given by an immediate to an address or just increment by 1.

Mem_doutA_ctrl – Controls whether or not the register file input is from the memory A port output.

Read_imm – Controls whether or not ALU bus A will take in the register information or the immediate from the instruction that was sign extended.

Write – Controls which register we are writing to in the register file.

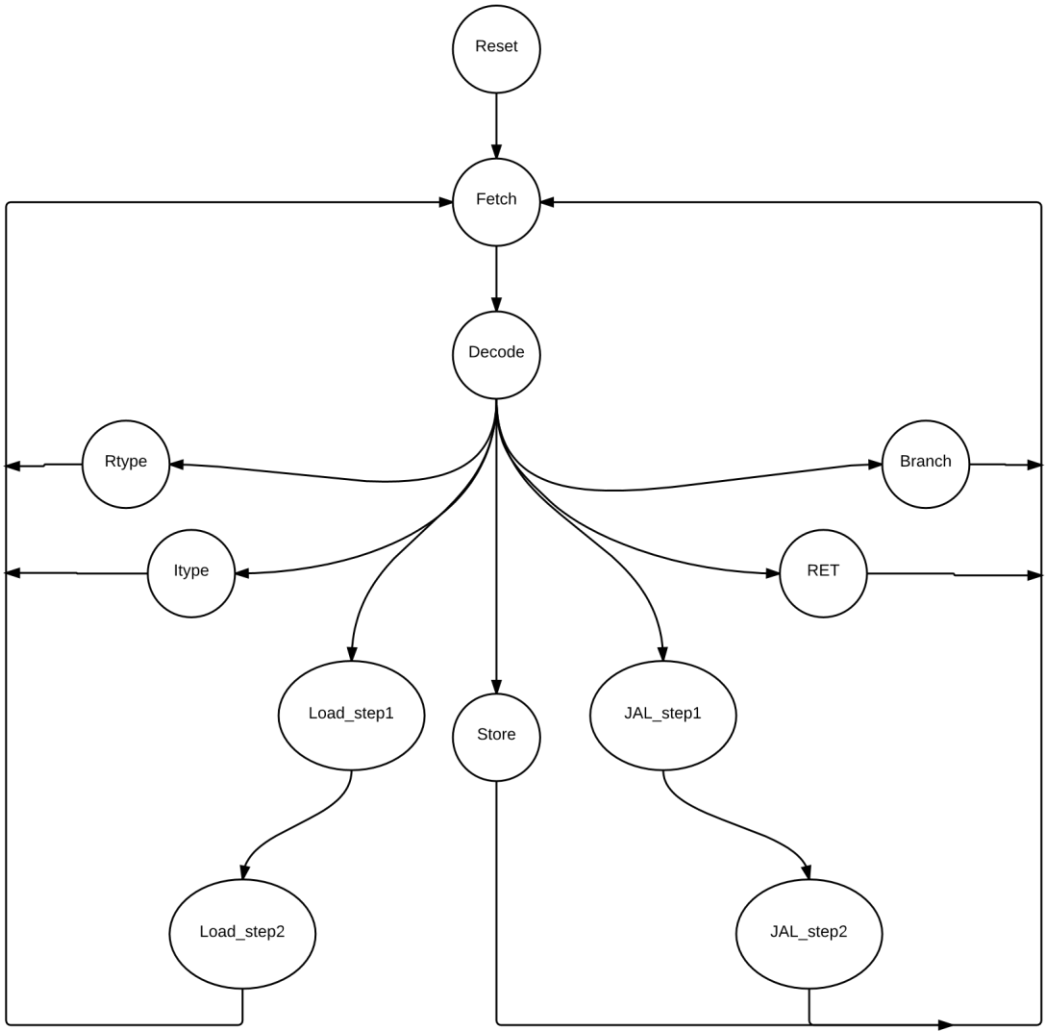
Read_Reg_BusA – Controls which register we are reading from on ALU bus A.

Read_Reg_BusB – Controls which register we are reading from on ALU bus B.

Cin – Controls whether there is a Cin value or not in the ALU.

ALU_Buff_ctrl – Controls whether or not the register file is receiving the ALU output as an input.

Flags_enable – Controls when flag updates are enabled.

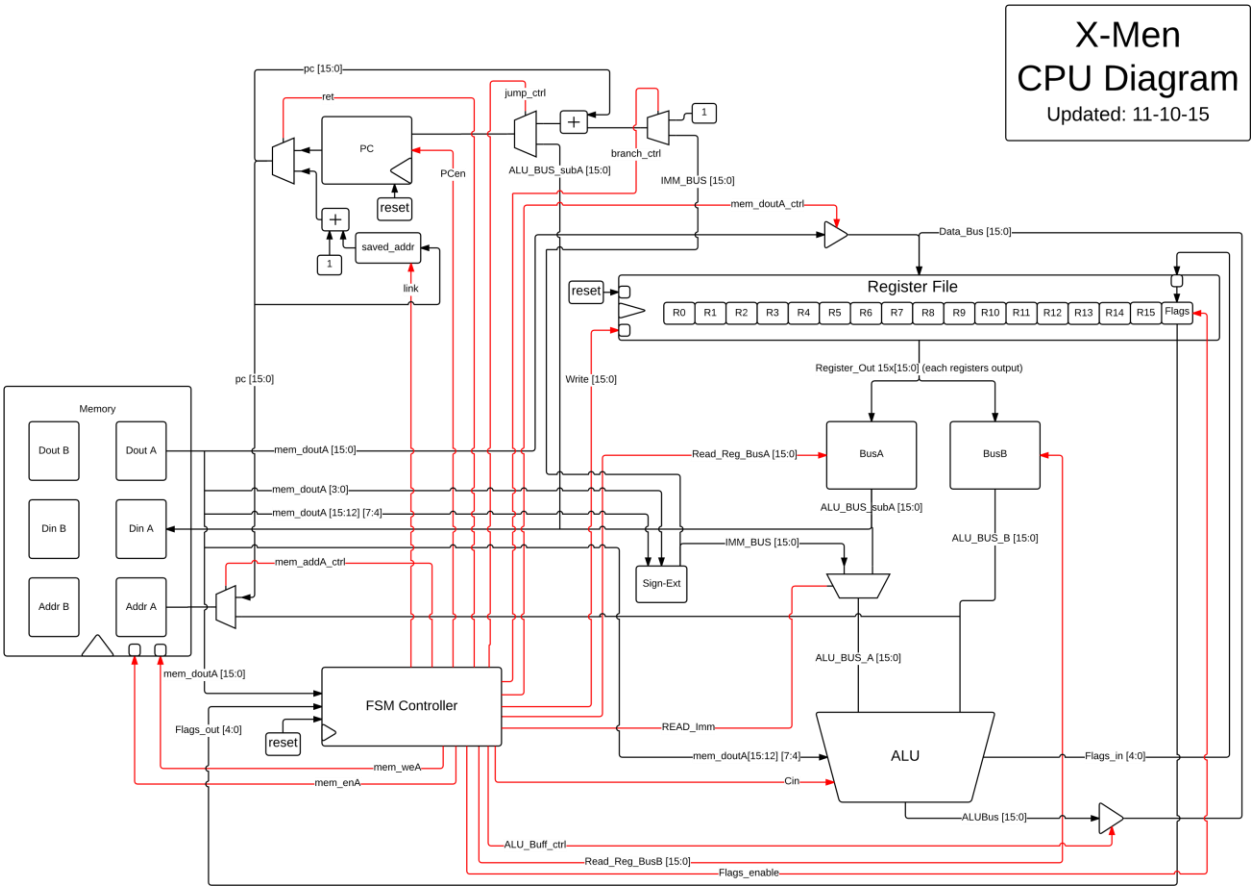


The FSM is split into multiple types of instructions. First are the R-type and I-type instructions. Basically register to register actions and Immediate to register actions. These two types of instructions only take one cycle. Branch instructions only take one clock cycle as well. PC gets incremented by an immediate amount. Return also takes one clock cycle. It takes the saved address in the pc and outputs it. One more operation that only takes one clock cycle is

the store instruction. It takes a register that holds an address and another register that holds information and stores it at the address. The two instructions that take 2 clock cycles are the load and JAL instructions. Load takes two clock cycles because we have to load an address into a register and then load that address into memory so we can get it out from memory and back into a register. JAL takes two instructions because the first jump is one instruction and then the link portion is a second instruction. So technically JAL is two separate instructions but it is going toward the same goal.

The VGA Controller and the Bit Generator were only connected to port B. Normally we would connect to both ports if using a glyph library and frame buffer, but we found a way to do it with only a frame buffer since we were only using black and white. We actually ended up using multiple frame buffers. What we did was preload all the screens we were going to use and then based on the state of the program, change the offsets to display that frame buffer. We did that by always enabling port B and setting write on port B to ground since we would never be writing to port B. We also set the Data input to ground as well since we would never be writing. The VGA Controller does nothing but take in a clock and send the hSync and vSync signals to the VGA pins and also scans the screens using counters at 25Mhz. The counters are sent to the Bit Generator, which decides which color to display at that moment in time. There are these things called back and front porches on the VGA, which basically prep the scanner to be ready to display. During this time we are reading from a memory location that is being written to during our program that has a control signal that chooses which frame buffer we are going to display. Once we get off the porch to the display area, we read from the frame buffer in memory from the Bit Generator and calculate the color to display based on the hCount and vCount (where we are on the screen). The bright signal is what determines which address we are reading from in memory port B. It could be the address that determines which offset to use or the address that contains the color value. The memory port B output is always outputting something because we are always updating the screen. That is what is nice about our setup, we don't have to attach anything to the VGA but the memory.

This is a schematic of our CPU before adding the VGA, Timer, Motors, and NES Controller:



When we reached the end of our project. We realized we needed something to determine whether we were accessing memory locations for the program or accessing memory locations for the motor limits. So we basically fed the data out of port A into an I/O Controller and fed it out to the sign extender and FSM and same for the data in. We determine whether or not it is a motor limit based on the instruction codes.

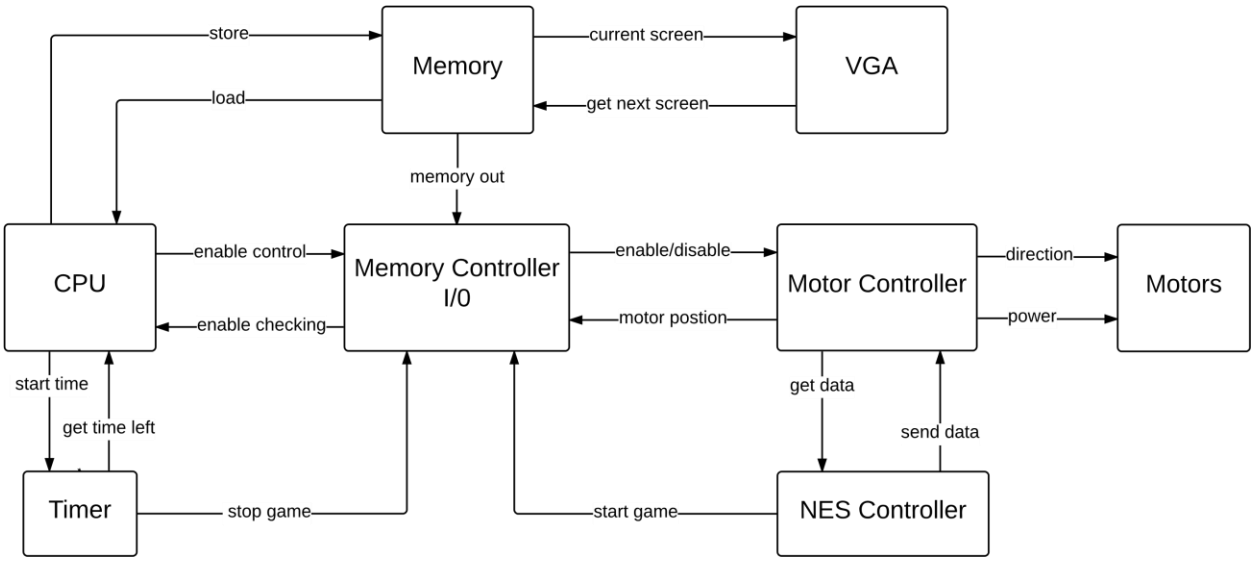
For the motors and NES Controller. The NES controller sent out data and also took in inputs that were a clock and a latch signal. The latch signal is sort of like a pulse telling the controller to latch the state of all buttons internally. The data sent out is what drives the motors. It is sent to the Motion module that controls the drivers. When a button was pushed it would send a signal to the driver that was connected to the motor and amplify the signal for the motor to work correctly. Also based on whether or not the data we are reading from memory Port A is a limit or not, it enables movement of the motors or not. This is based on the state of the program. We would do compares inside the Controller with what was in memory to decide whether or not the motors could move. During this portion we learned that sourcing physical parts is extraordinarily time consuming and difficult.

The timer module in the CPU is used when start is pushed on the NES controller. The start button is fed into NES I/O controller and is sent as the Memory port A output if it is pushed. Then it is fed into the ALU where the START instruction is used and sent to the timer. The timer then starts. It constantly does a compare on a 60 second immediate, and once it

reaches that it stops motor control. It does this by sending the ENCHECK signal to the IO Controller when the compare has reached an equal. That signal is sent to the Motion module.

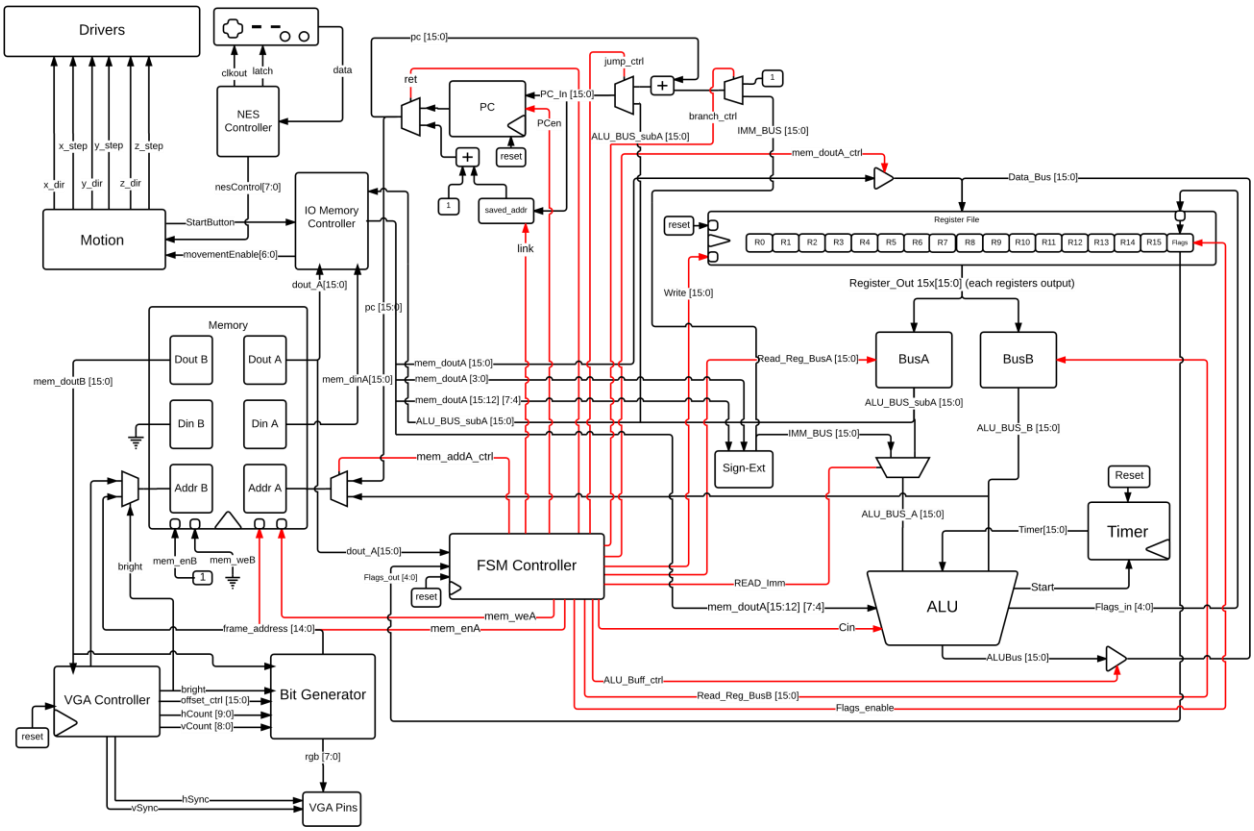
Just to quickly add as well, we have a sign extension module that sign extends immediates depending on the instructions. We only have a few immediate instructions to do that for as well. The immediates are used for the ALU and PC branching.

Overall, this is our final block diagram:



Basically our program loops until the NES controller pushes start. It sends that to the CPU, where it goes to the ALU and goes to the timer module and the timer starts. The timer continuously does checks with the ALU and Regfile(which also used memory to load time max into a register) to check if time limit has been reached. Once that has happened the Timer sends out a stop game signal via the Regfile to the Memory Controller I/O, which is then sent to the Motor Controller and finally the Motors, and disables them. During all this time the NES Controller is sending data to the Motor Controller which sends data to the drivers that control the motors. Also during this time, the VGA is constantly running with memory.

Our final schematic was the following:



Software

The software instruction set began taking shape during the first few weeks of the term. As the ALU was being developed there had to be a set of instructions and opcodes ready for its design. A list of desired commands and opcodes that would be implemented from the ALU was made. The initial list grew as the term went on. A problem we faced came when the memory was complete and the FSM was in the early design phase. It was apparent that our list of commands lacked memory interface and jump capability. The choice was made to add a store and load command to access memory. The commands were built into the FSM since no ALU access was needed to transfer elements to and from memory. Both the load and store commands were implemented in the FSM as R-type commands. The requirement was that the register specified on the right hand side of the command must have the address that will be accessed in memory. The next capability needed was jumping or branch capability. The ALU was capable of doing comparisons and setting flags, but up to this point in the design they were not used. The jump commands were implemented as follows:

BRANCH = 1100???? Normal immediate command except Rdest = conditions:

0000 – jump

0001 – JG

0010 – JL

0011 – JE

0100 – JLE

0101 – JGE

0110 - JNE

Default = jump

The branch was used as a generic label to represent the jump capability. The actual commands implemented in the end were the condition names (jg, jle, jge…). The FSM was responsible for taking care of actually implementing the jumps as they happened.

The final jump command added was a jump and link (JAL). The jump and link was implemented by the FSM, but the actual return address was stored inside of the PC. The specifics of the design at the time seemed to necessitate a JAL, but with only one link taking place at any given time. The constraint was that if a JAL had been initiated then a return must take place before the next JAL. Basically only one return address could be stored at any given time.

With all the jumps in place and the FSM completed it was time to run a test of the software. No assembler had yet been made so the test had to be done by manually converting pseudo code into hex code that could then be downloaded into the memory. A change was made to the processor to allow it to display the memory out port on the LED display connected to the board. A timer was inserted into the FSM module that slowed it down by sending an enable pulse every two seconds. On each enable pulse an instruction would execute and then hold for the next pulse. This change made it possible to watch a program execute on the board slowly enough that the output could be seen for each command.The initial test followed the following pseudo code:

Move a 5 into all registers

Move a 1 into register 0

Add r0 to r1 and store in r1

Add r0 to r2 and store in r2

Follow the addition until r15

Subtract r0 from r15

Compare r15 to 0

JNE back to the addition start point

Build an address and store it in r0

Store register 4 to memory at address r0

Load into r15 the value at the address r0

Since this was the first time the processor had ever actually had any software a test bench was used to simulate it. All the registers updated properly and the memory updated with the store command. Everything looked great and so the test was moved to the board for an actual running hardware/software test. The LED display updated showing each instruction as it executed. No problems were discovered during this test and it showed a functioning processor design capable of running the designed instruction set.

The processor required further testing to be sure of the functionality. A second test was written to jest the conditional jumps and the JAL instruction. The pseudo code looked like this:

Move a 0 into register 0

Move a 1 into register 1

Move a 3 into register 3

Continue until 15 into register 15

Jump + 2 skipping a wait command

Compare

Test jg

Compare

Test jl

Compare

Test je

Compare

Test jle

Compare

Test jge

Build a return command and store it into memory at position ffff

Jal to ffff

Move 1 into r1

The test ran with some errors that helped make clear that the compare instructions did not work as expected. The FSM had some compare responses reversed. This was corrected and then the test was run through again. The test passed and it was able to be seen that the instructions all executed normally.

With the instruction testing out of the way it was time to write the assembler. Java was the programming language chosen to write the assembler in. The challenge of writing the assembler was deceivingly more complicated than originally thought.

The assembler was written in two parts. An object was created and named Instruction. The Instruction object would take in the instruction name, opcode, registers, jump labels, and memory position. The main program would read through a text file and fill in the Instruction object with the known values from the file. Once an instruction is read in it is given the correct opcode.

Jump instructions needed to be parsed differently than the others due to labels. In order to know where to jump to a label system was implemented inside the assembler. A jump instruction references a label. Since each Instruction object contained a label space it was easy to calculate the difference between the jump command and the destination. After the calculations were done the Instruction object would update with the distance. The jump commands in the processor would jump by an immediate amount limiting label separation to 8 bits.

JAL commands were unique in that they required an address to jump to rather than an immediate difference. To get the JAL to work correctly the assembler would add 6 commands before the JAL inside the Instruction object. Once the program had been completely read in the assembler would find the destination for the JAL and then fill in the six blank commands with the appropriate commands necessary to build the address.

The addresses for a JAL were built as follows:

Move into r15 the top two hex values of the address

Left Shift those values by 8

Move into r14 the next single hex value

Left shift it by 4

Add the final hex value to r14

Add r14 to r15 and store in r15 the result

When the commands had executed the address needed was always stored into register 15. This meant that if a JAL command was given it would always jump to the address found in r15. Part of the jumping process was to store the current PC value into a register internal to the PC.

Return commands as well as wait commands and later other singular argument commands had to be dealt with individually in the assembler. The assembler had a special section designed to deal specifically with singular argument commands. It automatically filled in the register values to be what was needed to form the final hex command for download to the processor.

When the VGA work started it became clear that there needed to be a way of dealing with it in the software. The specifics of the project made it such that VGA screens would not play a major role in the final presentation. After discussing the situation with those working on VGA it was determined that the VGA would look to a specific memory location for which screen to display. The processor was responsible for filling that location with the code representing the desired screen. The memory location chose was 20100. There were 13 screens total and so the numbers 0-12 were used to designate which screens to display.

The game needed a timer to work correctly. The idea was to give individuals 60 seconds to move the machine around before it would stop. A 10 second countdown would display on the VGA. The original method for implementing the timer was software. The idea here was that by filling a register to capacity by doing addition by 1 it could be used as a delay that could be calculated. The addition would be repeated as often as needed until the desired delay had passed. This method worked fine, but it became apparent that any changes to the software would mess the timing up due to introducing extra clock cycles not being devoted to the delay.

A hardware timer was a much better method of getting this done. A timer module was written that would count in seconds. It was set for 60 seconds and would count to 0. New software commands were now needed to deal with the timer. The timer connected to the ALU allowing new commands to be built there that would deal with it. Two commands were added and they are shown below:

GET = 10000101

START = 00001111

Get would get the current time on the timer and store it in register 0. Start would send a start signal to the timer allowing the countdown to start.

With the timer in place and a way to control it it was time to test the VGA with software and a timer. A program was written that cycled through screens. The pseudo code is shown below:

Make the address for VGA storage using 6 commands.

Start the timer

Check the timer until 2 seconds has passed.

Switch to a new screen

Check again for two seconds

Switch screens

Continue until all screens have been displayed and restart

The program worked without any problems and proved a working timer as well as a working VGA that could be dealt with by memory manipulation.

The final hurdle for the software/hardware was to check the motor limits and stop motors moving if boundaries were breached. The original plan was to have a memory interface inside the motor control block that would save current motor conditions to memory. The processor would then grab those values from memory and compare the to the known safety limits. If a problem was found a memory write to a specific location would tell the motors to stop. This method of implementation was very problematic and in the end had to be scrapped. A software program was written to deal with this method, but since the hardware never worked the software was never implemented.

The way this issue was resolved was to add a new input to the processor and a new output as well. The new input in the top module was a signal that indicated the start button had been pressed. This new signal necessitated a new software instruction. The new instruction is shown below:

ENCHECK = 10000111

Encheck passed the enable signal through to whatever register was specified. A compare could then be done to see if the motors could run.

The way the system worked is that two FPGA boards were used. One would run the motor control hardware and send out a signal if start was pressed. The other board ran the software that would change screens at the correct moment and send a disable signal when the time was over.

The dual board approach paid off and allowed the project to run correctly. The software written for the final presentation worked like this:

Build the VGA address

Store start screen code to memory for VGA

Loop until start is pushed on the controller

If start pushed start the timer

Update VGA to display running screen

Wait until 10 seconds are left.

Display the 10 second screen

Continue checking and displaying countdown until 0

Send the end signal

Update VGA to show game over screen

Start the timer

Wait 7 seconds

Update VGA to show welcome screen

Start over by waiting for start

This software was made in a file called dual control. It ran through without error and allowed the project to be used as intended.

The software commands and information on how to properly format them were put into a word document and distributed to all members of the group so that anyone could write working software that the assembler could understand.

The Work/Conclusion

Overall the work was split up pretty evenly throughout the semester. We all worked on the ALU together, as well as the Regfile, Memory, FSM, PC, etc. At times it may not have seemed like we were making progress or that one person may have been working on something, and sometimes it was. Sometimes Clint would be deep into finishing up something and we needed to be patient and offer help and insight while he did his thing. We would actually think of problems we might encounter during times like this and discussed future goals. A few times Keith would layout a skeleton and we would all fill it in. For instance, the ALU and FSM design was done by Keith. We all worked on the ALU and Nathan did most of the FSM. We did make changes and fixes to both as the semester went on but we all worked on them. John made a huge difference in the group with his schematics made on lucidchart. When we were connecting everything together to get our FSM working and also the final CPU together, we all referenced Johns schematic. And when we needed a change, John would change it and print out another copy. That schematic alone made a huge difference. When we got to the last stretch of the project, we all went our separate ways. Nathan and John worked on the VGA as well as some stuff for the final poster. We had to collaborate with Keith and his program once we got the VGA working. We basically only had knowledge of our own separate portions. So at

the end when putting it together, we needed to tell each other what was needed of one another. Keith was in charge of the assembler and program. He did a great job and has a great understanding of Java and what our compiler was supposed to do. He also made a timer module at the end to add into the CPU for the final demo. Clint was in charge of all the mechanical devices. He had to figure out how the motors worked, how the drivers for the motors worked, and how the NES controller worked. He basically built the physical portion of the project. We all had something to offer and I think we made a pretty good team. Some of us had strengths and some of us had weaknesses.

The main lesson that we learned is that modules should be tested periodically as they are constructed to verify expected behavior. We also learned that modules may work individually, but that does not guarantee that they will behave as expected when assembled. We learned that sourcing physical parts is extraordinarily time consuming and difficult. Some of the smaller lessons that we learned and relearned while coding and designing are the importance tracking and consistency when declaring variables, and the importance of maintaining physical drawings to keep track of all connections within the project. A single missed wire is enough to cause the entire design to fail.

Originally, our plan was to make the same xyz crane, but instead of a hook, use an electromagnet attached at the end. We would let the user determine when the electromagnet was on and off with the NES controller. Then we would have them drop whatever we were going to have them pick up into a bin of some sort. We were thinking that the bin could have a laser that, if broken, would cause an increase in a score. It is crazy to think that many companies today have very limited time to finish projects and get them out by a certain time, and a lot of times they have bugs that need to be patched as time goes on. Originally we were planning to even have our hook game at least be on one board, but we actually split it up into two boards. One board controlled the VGA and program. While the other board controlled the physical devices. The two boards would send enable and disable signals back and forth to each other. The enable signal was from the NES controller and started the program on the other board. Then when the time limit ended on the other board, a disable signal was sent to the board controlling the mechanical portion, which disabled the motor movement. We ran out of time and did the best we could to get it working to our best intentions. Yes, we all wanted the project to be better and different, but we were still happy with the results.

References

http://www.mit.edu/~tarvizo/nes-controller.html - NES controller interface

http://www.schmalzhaus.com/EasyDriver/ - Motor drivers

https://www.youtube.com/watch?v=WKmzlMvxC7I – Crane design/construction

We also used many of the links listed on http://ece.utah.edu/~kalla/index_3710.html, which helped us with every aspect of the design; Links that helped us from the ALU to the CPU, and also the VGA; Manuals for the FPGA Nexys3 board we were using, as well as the CR16 instruction set architecture.