Midterm Report - Single cycle and Pipelined Processor

ECE437 : Computer Design and Prototyping

TA : Jason Holmes

March 1, 2013

Alex van Almelo and Vineeth Harikumar

Executive Overview

For the lab section of ECE 437, our group was tasked to design a single-cycle processor using the MIPS ISA, then implement a pipelining control system for the overall design, increasing the processor's throughput of instructions. While a single cycle processor design is relatively simplistic, it has a very low throughput of commands and instructions, making the overall speed of the processor slow. Implementing pipelining would speed up the processor, but introduces many hazards and corner-case logic that we would have to take into account. However we have successfully created a single-cycle processor, then improved it by implementing pipelining and properly dealt with each hazard and forwarding logic case that could possibly come up.

Included in the rest of this report will be our personal choices for the single-cycle processor, since that was an individual project there will be two designs, one for Vineeth's design and one for Alex's design. For the rest of our project, we chose Vineeth's single cycle design as the design to convert into a pipelined processor, because Alex's design had a few errors and would not support a few commands that required some revision. After our designs are presented, the report will cover how our pipelined processor was built and debugged. After our processor debugging section, we will cover the results of our processor's testing and implementation, explaining what we tested, and how we tested it.

Processor Design

For Vineeth's single cycle processor (Appendix A Figure 1.1), a majority of the design followed the one suggested by the book. I broke up the entire design into three main sections which are the program counter block which contained all the sections required to update the next program counter, the controller block which generated the control signals for the various blocks based on the instruction, and the instruction execution block which does the I/O from memory, reads and writes to the register file, and computes instruction results in the ALU.

The program counter block contains a program counter register, which holds the present value of the program counter. There are three muxes, each of which have write enables that are flags generated by the controller block and allow either the next program counter(program counter + 4), the jump address, the branch address or the return address to pass through to the program counter register. The program counter register on every rising edge of a clock and when the write enable signal is asserted updates the program counter with the next value on the input.

The controller block is where I deviated a bit from the suggested design. The suggested design in the book had a controller which decoded the instruction into flags based on only the instruction opcode, and had a separate ALU control block that accepted the ALUOp flag and the function code and generated the ALU instruction to be executed. I decided to simplify this by simply having my controller block be a master decoder, which accepts two inputs - the instruction opcode and the function code and decodes all the necessary flags based on both these two inputs.

Finally, the instruction execution block is the one where the actual instruction is executed and the results are generated. It starts with a register file from where the values are written to and read from. These values are then sent to the ALU, where based on the ALUOp flag a computation occurs. Once the result is computed it may either choose to write or read a value from memory. From here the resulting value makes its way through multiple muxes back to be written to the register file. There are a few differences in my design in this block compared to the design suggested in the book. For the shift left or shift right instructions, since the shift amount information is contained in the instruction and not in the register file, I decided to add a mux in front of the second input to the ALU. The mux enable is controlled by the shift flag. So every time there's a SLL or SRL instruction, the mux will allow the shift amount(shamt) bits go through the mux to the ALU instead of the regular data read from the register file. Another way that my design was changed was for the handling of SLTU, SLTIU, SLT and SLTU. Instead of incorporating the execution of these instructions inside the ALU, I decided to simply add an external mux. For every SLTU/SLTIU/SLT/SLTU operation, the two inputs being compared are sent to the ALU and a subtraction is executed. The resulting negative flag bit is zero extended to 32 bits and forwarded to the mux. This mux chooses between the extended negative flag and the regular ALU result that is computed.

For Alex's single cycle processor, I followed a similar design to the book as well, breaking most parts up into individual blocks and following the suggested progression. However I did deviate from the book a little to make the design a bit more modular and steps a little easier to handle.

For the program counter, I had a total of 3 multiplexers to handle any branching and jumping logic, along with halt statements. The PC block itself was a simple state machine

with 3 states, allowing the program counter to take into account a “load” instruction, delaying it for the one clock cycle that is necessary, and also making sure that a “halt” instruction will send the processor into a halt state and won’t accidentally continue running.

The controller block operates normally by taking in the entire instruction code, and using a lot of selection statements, decodes exactly what instruction is being called, and asserting the proper signals to make sure my design operates properly. I had to include a signal for “Branch”, “Branch if not equal”, and “Jump” signals to enable the multiplexers and logic gates accordingly, allowing the program counter to take in the new count from the right source.

My execute block was the only part of my design that I did not have 100% working. I followed the book’s guide pretty closely, making sure that my design was capable of accepting my small changes and alterations. The design would make sure that the ALU would either take the extended-immediate value, or the register value given in the instruction, and did the arithmetic properly. Once this was finished, my ALU would decide, based off a few signals, where to write the newly computed data. I realized later that by forwarding the instruction data to the ALU for any shifting logic (SLL, SRL, etc) would have made my design a lot easier to handle, so adding a multiplexer and selection logic was necessary. I also had to modify my ALU to compute the LUI instruction, along with the SLT instructions. Once I had completed the modifications, I worked to make my design error free, however I was not able to have my design work for all cases.

Once the single cycle processor was completed, work on the pipelined processor began. For the pipeline processor design, a lot more complex design decisions had to be made from the beginning. The pipelined processor we built contains five unique stages - the instruction fetch stage (IF), instruction decode stage (ID), execute stage (EX), memory stage (MEM) and the write-back (WB) stage. The main difference between the single cycle and the pipelined processor is the fact that in a single cycle processor only one instruction can execute in a given clock cycle whereas in a fully pipelined five stage processor such as ours, up to five instructions will be using the processors various resources in the different stages of the processor within a given clock cycle.

Firstly, four pipeline registers were created, each of them were placed in between each one of the five pipeline stages. These registers all have a write enable which writes the data on the inputs to the outputs on every rising edge of the clock, and a reset which resets all the outputs of the registers to their default values (all bits set to zeroes). The first stage, the instruction fetch stage is responsible for fetching the instruction from RAM. The second stage, the instruction decode stage is responsible for decoding the instruction and setting the correct flags. The third stage, the execute stage is responsible for accepting the data and carry out the computation required in the ALU And produce a result. The fourth stage, the memory stage is responsible for any reads or writes that needed to happen with the memory. The final stage, the write back stage is the one responsible for writing the data back to the register file when necessary. The various functional units from the single cycle processor were broken up and placed strategically into the different pipeline stages. There is also a hazard detection unit external to each of the five stages that detects and rectifies data hazards. A forwarding unit was also created to enable any instruction to receive any data dependencies without having to stall the pipeline and wait for the data to be written back to the register file. A memory arbiter was also required to control the reads and writes to memory from different stages of the pipeline.

One of the design decisions made was regarding the function of the hazard

detection unit. This unit was required for instructions where there was a load word followed by an immediate use of the loaded word. It was decided that this unit would be allowed to de-assert the program counters write enable directly, the IF/ID pipeline's write enable and zero out all the control flags to the ID/EX pipeline. Doing this allows a NOP to be inserted between the load instruction the instruction following it that uses the loaded word so as to prevent data hazards.

The biggest design decisions were made for the memory arbiter. The function of the memory arbiter is to allow both the instruction fetch and the memory stage to access the RAM as and when needed. In its regular mode of operation an instruction is fetched every cycle from RAM, but when there's either a store word or load word in the memory stage the arbiter needs to give priority to the instruction in the memory stage. After careful consideration it was decided that a mealy machine would be used as opposed to combinational logic or a moore machine. This enabled us to assert the right output flags from the memory arbiter because all the flags need to be asserted on the same clock cycle as opposed to possibly one clock cycle late. The mealy machine was designed so as to give preference to LOAD and STORE instructions. If the state machine detected one of those two instructions in the memory stage, it would jump to a load or store state as opposed to an instruction fetch state. Once the load or store is completed, it jumps back to the regular instruction fetch state if there's a instruction memory request. Another decision with the memory arbiter was to allow the memory arbiter to stall everything in the pipeline during memory access latency. This included having to disable program counter register from incrementing and also allow it to stall all the four pipelines registers when there was a load or store word instruction the memory stage of the pipeline.

Processor Debugging

Looking at the memout.hex it is clear that there are multiple reasons that would cause that output. The first one is that the BNE instruction should've evaluated to true and branched to foobar2000 but it did not. There could be three possible hardware design errors that could've caused this. For a BNE instruction in the execute stage, the two operands being checked are sent through the ALU and a subtraction operation is executed on them. This is where the first hardware design error could've occurred. For this specific BNE instruction, it should've resulted in the zero flag of the ALU to be set to a value of '0' but if the ALU wasn't implemented correctly it could've resulted in the zero flag being set to a value of '1'. In the output waveforms at this stage we would look to check if the zero flag resulting from the ALU is a 0 and not a 1. This bug can be tested by checking the ALU independently and ensuring that on every type of subtraction command the zero flag being set is expected and correct. After this, there exists a mux which has two inputs, one being a zero flag and the second being the inverted zero flag and each one of forwarded depending on whether the controller block set the BNE_flag to the correct value. If the controller did not set the right flags, it could be the second hardware design error which would've resulted in the wrong value to be forwarded. In the output waveforms at this stage we would look to ensure that the BNE_flag was set correctly and thus forwards the correct zero value through the mux. If its incorrect then you would have to back up to the decode stage and figure out why the controller block did not decode the correct flags. Finally the zero flag is to be logically anded with the branch flag and this resulting value is used as a mux enable to

forward the branch address to the program counter register instead of the next program counter value(AKA PC+4). If this mux enable was not correctly set, then the program counter will never get updated with the branch address of foobar2000 and instead will keep updating itself with the next program counter value. In the output waveforms at this stage we would look at the mux enable that chooses between the branch address and the next program counter to ensure that the mux enable bit it correct and that the value being forwarded to the program counter is correct. These last two design errors can be tested by passing values through the mux and setting the mux enables to be both ‘0’s and ‘1’s and to check that the correct value is forwarded in each one of the cases. Either one of the three hardware design errors could’ve independently resulted in the branch not being taken even though it should’ve been.

The second bug that is evident is that even though the BNE instruction was incorrectly evaluated, the HALT instruction immediately following it should’ve halted the entire pipelined processor. It should’ve done this by asserting the halt signal going out of the mycpu and holding the asserted value indefinitely and also disabling the program counters write enable, thus preventing the program counter by incrementing. After this it should’ve squashed both the instructions in the instruction decode and the execute stage. But clearly that was not the case because the four instructions after the first halt were also fetched and executed and allowed to make its way through the pipeline stages. This bug could’ve been caused because the halt signal wasn’t stable and held indefinitely at a ‘1’. In the waveforms we would check to ensure that the halt signal was held indefinitely at a ‘1’. It also would’ve happened if the program counter wasn’t disabled and prevented from incrementing, which would’ve resulted in the RAM providing the next instructions and executing them through the pipeline. Finally the previous pipeline stages should’ve been squashed by asserting their reset signals and resetting every signal to their default values.

Results

Single Cycle - Vineeth

The single cycle design executed each instruction in one clock cycle and hence had to have elongated clock cycle to accommodate the worst case critical path. This design is incapable of achieving high clock rates because of this very reason. The critical path ended up being the one from memory to the register with a path time of 23.094ns.

Table 1.1 - Vineeth's Single Cycle metrics

Assembly file	Clock cycles (clock cycle period)	CPI(Clock cycles/Instructions)	MIPS (Clock cycle period)
fib.asm	208	1.28	7.81 (100ns)
search.asm	320	1.15	8.69 (100ns)
mult.asm	830	1.01	9.9 (100ns)

The estimated frequency of my single cycle processor was 22.95MHz (period = 43.574ns).
Average Cycles per Instruction (CPI) = (1.28+1.15+1.01)/3 = 1.146
Average Instructions per Cycle (IPC) = 1/CPI = 0.872
Average latency of one instruction (100ns clock period) = 114.6ns
FPGA Resources required for design:

- Total logic elements ; 2,729 / 33,216 (8 %)
  - Total combinational functions ; 2,711 / 33,216 (8 %)
  - Dedicated logic registers ; 1,233 / 33,216 (4 %)

- Total registers ; 1233

Single Cycle - Alex

Alex's single cycle processor design had come into many issues when running the three scripts (fib, search, and mult), due to a lack of complete support for all instruction sets. Therefore, we based most of the numbers and estimates off of the working scripts that the processor could run.

With an extended clock cycle period (due to single cycle design constraints) the critical path of the processor design was 25.09 ns, routing from the memory block to the register block. This was expected and identified in the preliminary design of the processor.

Table 1.2 - Alex's Single Cycle metrics

Assembly file	Clock cycles (clock cycle period)	CPI(Clock cycles/Instructions)	MIPS (Clock cycle period)
test.loadstore.asm	17	1.38	7.05 (100ns)
test.ldst3.asm	22	1.28	7.72 (100ns)
test.rtype.asm	36	1.33	7.50 (100ns)

The estimated frequency of Alex's single cycle processor was 21.91MHz (Period = 45.633ns).
Average Cycles per Instruction (CPI) = (1.38+1.28+1.33)/3 = 1.33
Average Instructions per Cycle (IPC) = 1/CPI = 0.751
Average latency of one instruction (100ns clock period) = 133 ns
FPGA Resources required for design:

- Total logic elements ; 2,979 / 35,232 (9 %)
  - Total combinational functions ; 2,965 / 35,232 (9 %)
  - Dedicated logic registers ; 1,233 / 35,232 (4 %)

- Total registers ; 1233

Pipeline Processor

After synthesizing our design with a latency of 0, our processor ended up with a worst-case critical path of 17.956 nano-seconds. This path is not very long, due to the fact that with a pipelined processor, there are registers scattered about the entire design, and hard to have any setup and hold time violations. The path went from the top-level pins in the "mycpu" top-level block, and ended at the memory arbiter. This path was expected in our design, and is acceptable, since it is not a very large violation of time.

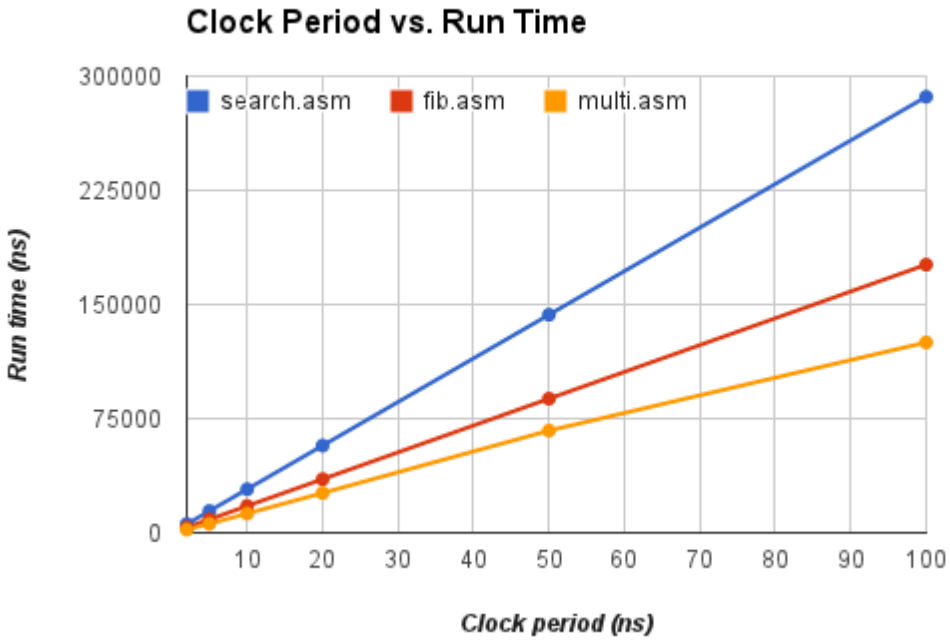
To calculate our processor's CPI and IPC, we would run three different algorithms (shown in the table below) and open up the ModelSim simulation. Once in the simulation, we determined the runtime of the algorithm, and derive the number of clock cycles used to complete the script. From there, we determined the number of instructions completed in each script by running the "sim" command in terminal, which would list the number of instructions completed.

The CPI is a basic arithmetic equation of number of clock cycles divided by the number of instructions. The IPC is just the inverse of the result. The MIPS rating is a rating given to a processor to benchmark how fast the processor can operate and execute commands. Take the number of instructions executed, and divide it by the multiplication of the time it took to complete (in seconds) and 1 million.

Table 1.2 – Pipeline processor's metrics

Assembly file	Clock cycles (clock cycle period)	CPI(Clock cycles/Instructions)	MIPS (Clock cycle period)
fib.asm	589	3.63	2.75 (100ns)
search.asm	956	3.43	2.91 (100ns)
mult.asm	770	3.88	2.57 (100ns)

Average CPI = (3.63 + 3.43 + 3.88) / 3 = 3.64
Average IPC = 1/CPI = 0.274
Average Latency per Instruction (100 ns period) = 364 ns
Estimated frequency = 68.98 MHz (Period = 14.496 ns)



The graph above demonstrates what happens to the runtime of different assembly scripts as their clock period is reduced. Using a range of periods (2, 5, 10, 20, 50, 100 nanoseconds, respectively) we saw a linear rise in the run time of our algorithms. In theory, if our processor's period headed to infinity, so would our run time. In the real world, this is very impractical for obvious reasons. However a smaller clock period has downfalls as well, since running a processor faster than normal causes a very large increase in current draw, along with a rise in heat. Both are related to each other, and both are very bad.

When writing to the FPGA device, our synthesis log reported back that the design took up a total of 4,230 logic elements. 2,974 of those logic elements were simply combinational functions, while the other 1,722 elements were dedicated logic registers. See Figure 2 in the Appendix for a more detailed output of the resource usage.

Overall, the design of our pipelined processor is complete, and works well. It successfully runs our scripts, at 0 latency, and at low clock periods (we even got it to work with 10 ps periods). However, when comparing to the single-cycle processor designs, our pipelined processor performed worse. This was an issue that was expected, since there are many times where a pipelined processor would begin to fetch and decode instructions, but then push those instructions back due to an instruction bubbling the pipeline, or simply not processing the instruction fully because of a load, store, or branch instruction. We predict that once our processor implements caching, the MIPS rating of our processor will increase, and the time spent executing scripts will drop too.

Conclusions

In conclusion, our group was successfully able to each design a single-cycle processor, and then convert one of the designs into a fully pipelined processor, complete with a hazard detection unit, and a forwarding unit. A major hurdle in our design was the removal of the second memory unit, making us create and implement a memory arbiter to handle the data. But once the memory arbiter was implemented, we were able to successfully run assembly scripts with an instruction latency of 0, and with clock periods ranging from 100 nano-seconds to 10 pico-seconds.

However, after running calculations on the processor's CPI, IPC, and MIPS rating, we discovered that technically speaking, our pipelined processor performed worse than our single cycle processor. This can be attributed to the manner in which a pipeline works, inserting "bubbles" into the processor to process special instructions would be a major factor in why our design ran worse. While the throughput of a pipeline processor is theoretically higher, in practice it turns out that a single cycle processor works more efficiently. We believe that once we can implement caches into the design that the power of a pipelined processor will increase dramatically. Another benefit of the pipelined processor is its reduced clock period, because the constraint of a critical path is lowered substantially, compared to a single-cycle processor.

Appendix A

Figure 1.1 - Vineeth's single cycle block diagram

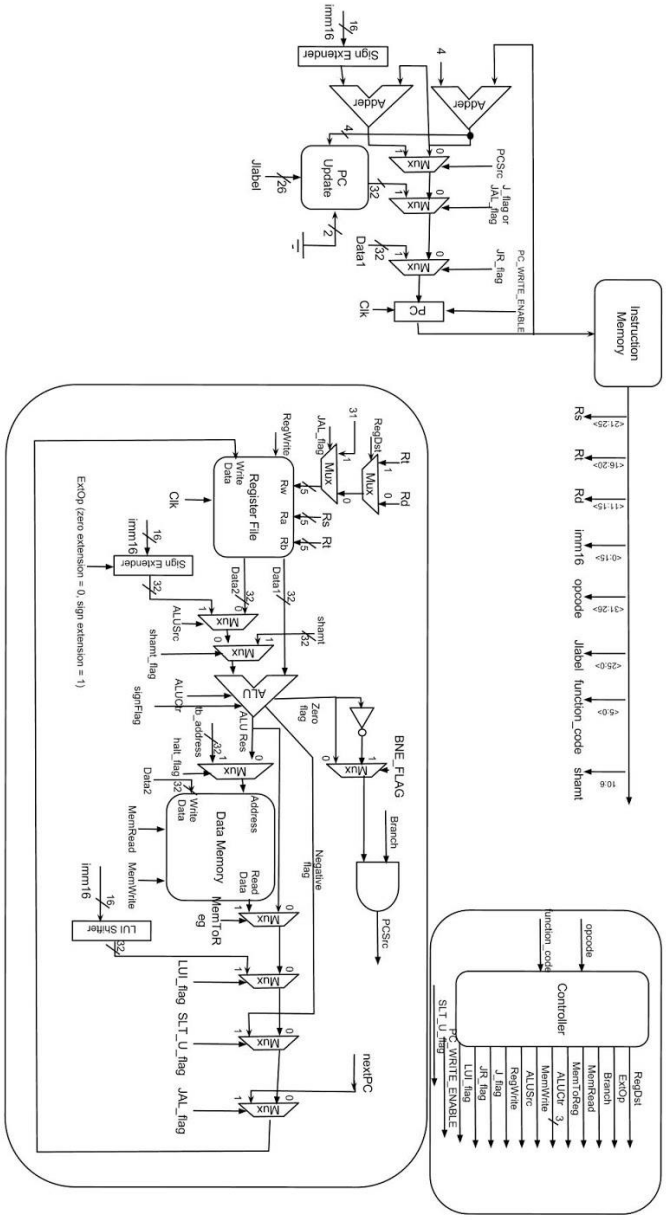


Figure 1.2 – Alex's single cycle block diagram

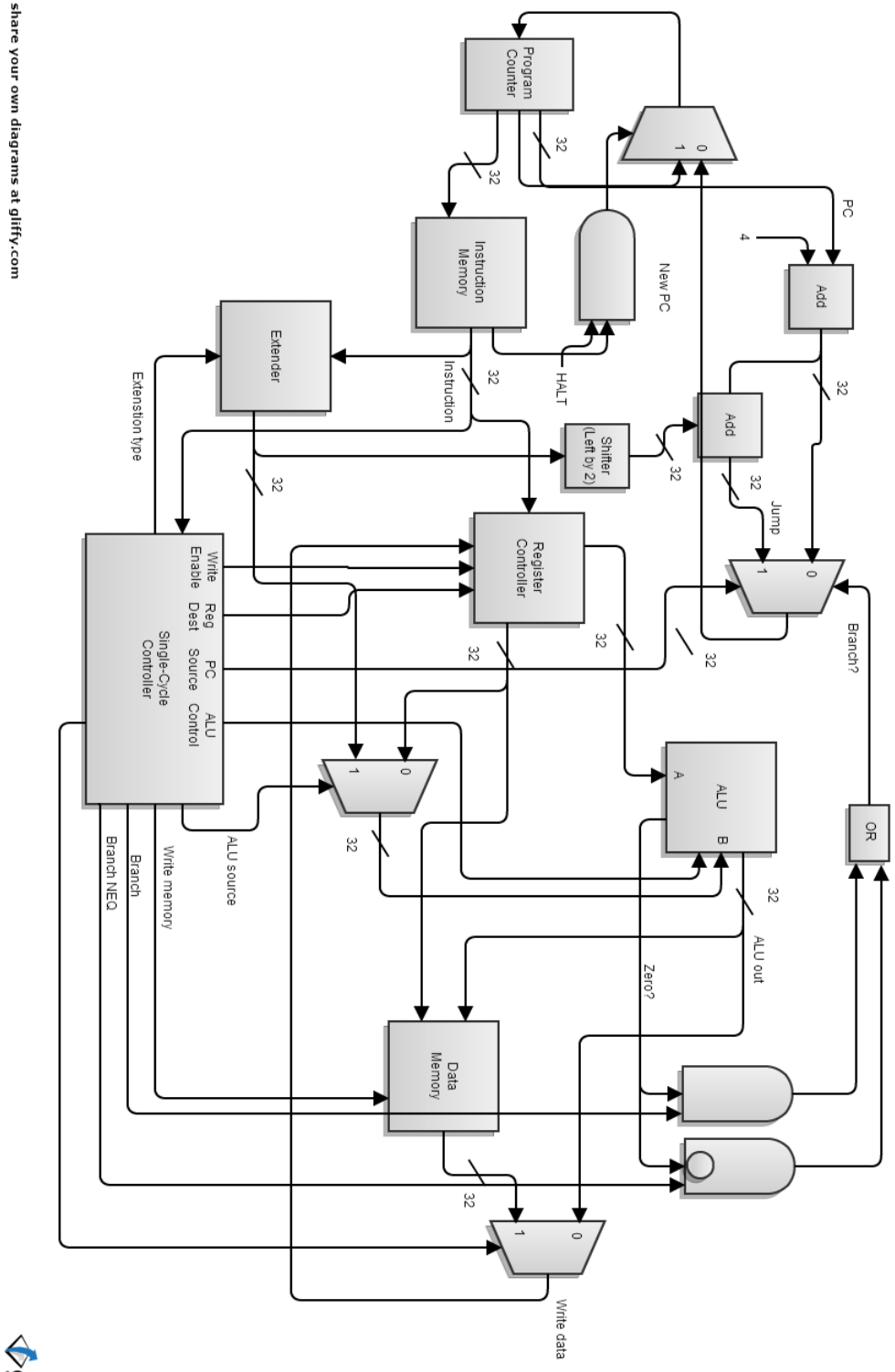
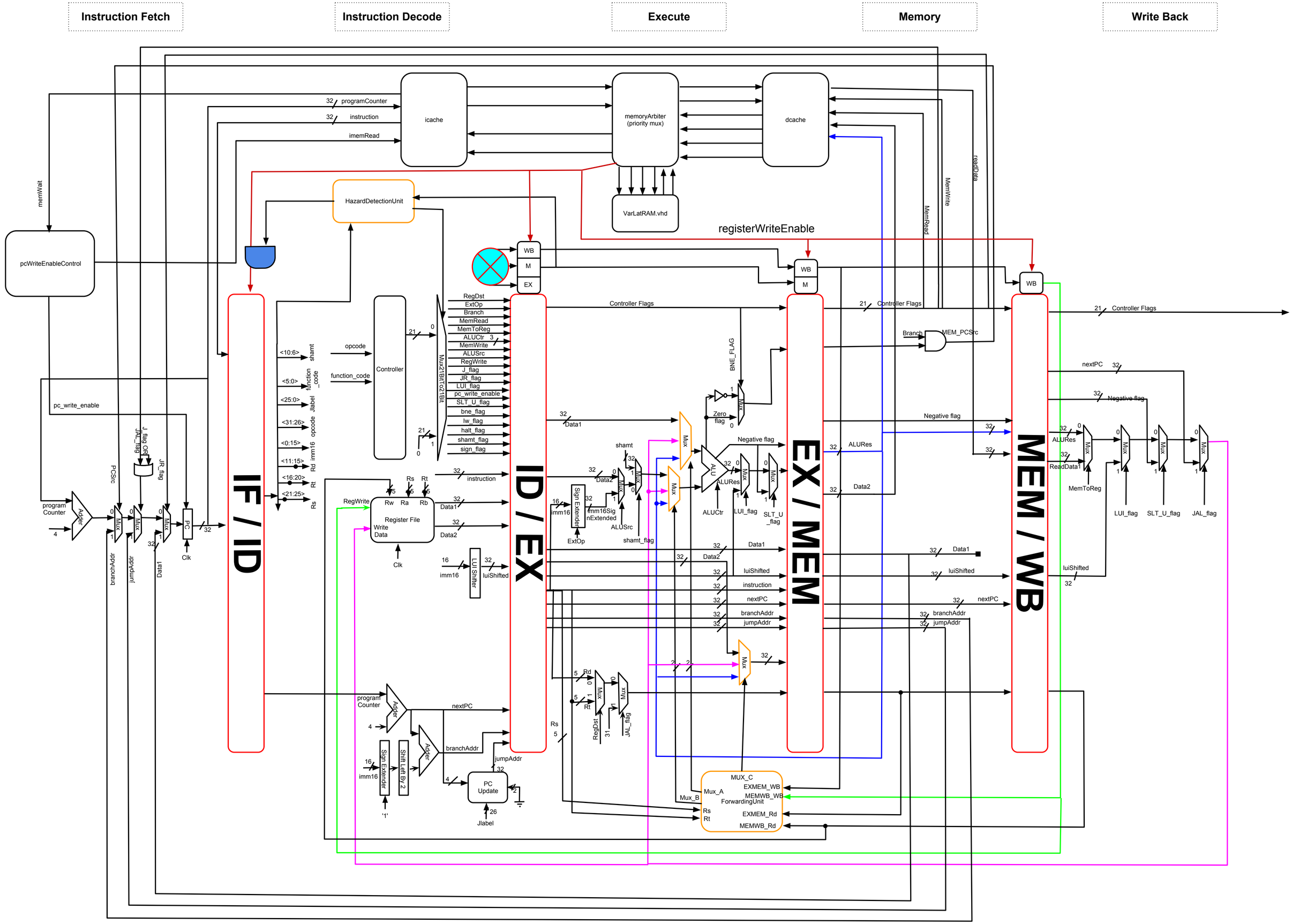
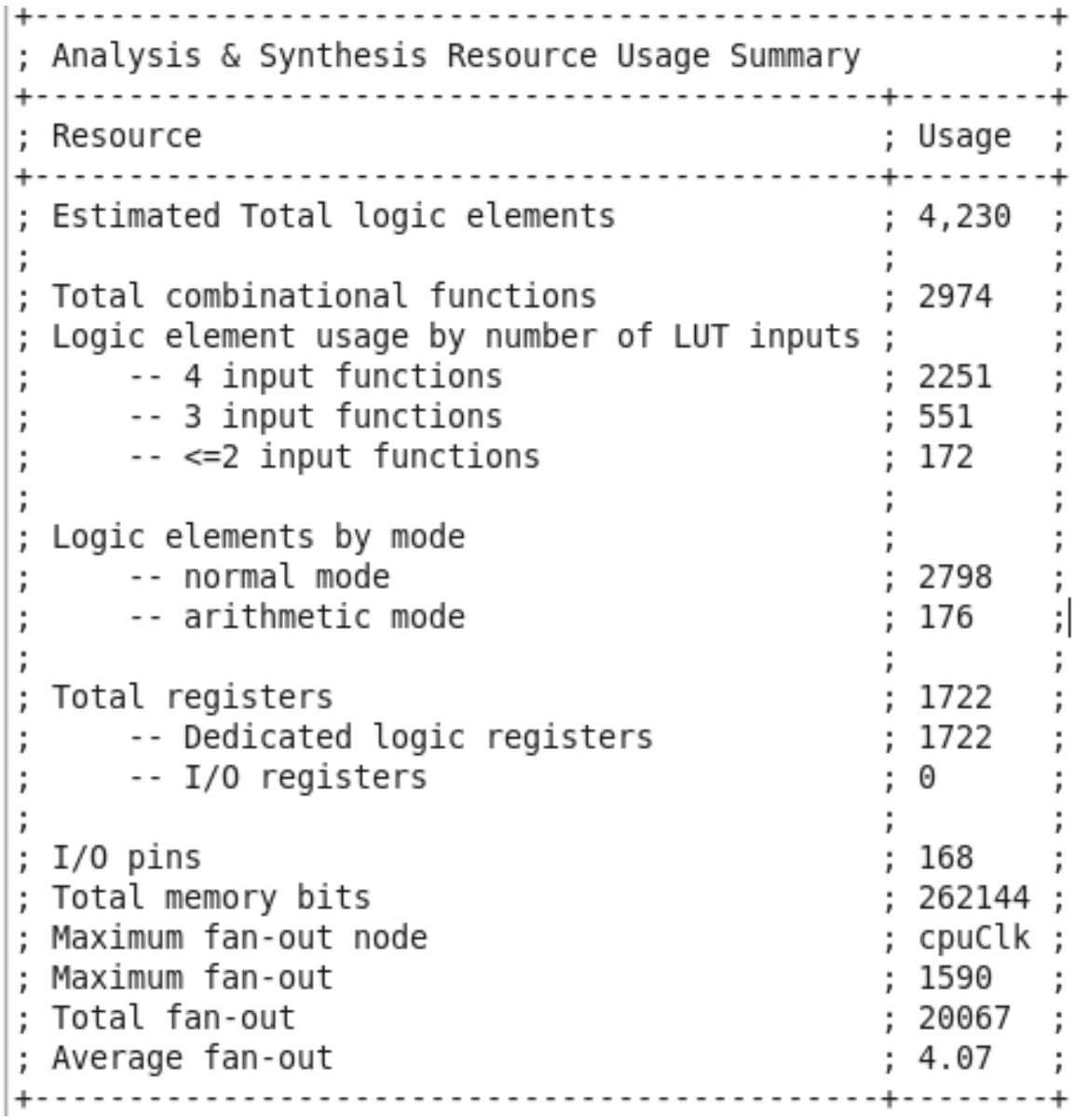


Figure 1.3 Pipelined Processor Block Diagram



Appendix B

Figure 2.1 – FPGA resource statistics



Analysis & Synthesis Resource Usage Summary	
Resource	Usage
Estimated Total logic elements	4,230
Total combinational functions	2974
Logic element usage by number of LUT inputs	
-- 4 input functions	2251
-- 3 input functions	551
-- <=2 input functions	172
Logic elements by mode	
-- normal mode	2798
-- arithmetic mode	176
Total registers	1722
-- Dedicated logic registers	1722
-- I/O registers	0
I/O pins	168
Total memory bits	262144
Maximum fan-out node	cpuClk
Maximum fan-out	1590
Total fan-out	20067
Average fan-out	4.07

Appendix C - Bonus Credit

Insertion Sort in C

```
#include<stdio.h>
int main()
{
 int n, c, d, t;
 int array[] = {
0x087d, 0x5fcb, 0xa41a, 0x4109, 0x4522, 0x700f, 0x766d, 0x6f60, 0x8a5e, 0x9580,
0x70a3, 0xaea9, 0x711a, 0x6f81, 0x8f9a, 0x2584, 0xa599, 0x4015, 0xce81, 0xf55b,
0x399e, 0xa23f, 0x3588, 0x33ac, 0xbce7, 0x2a6b, 0x9fa1, 0xc94b, 0xc65b, 0x0068,
0xf499, 0x5f71, 0xd06f, 0x14df, 0x1165, 0xf88d, 0x4ba4, 0x2e74, 0x5c6f, 0xd11e, 0x9222,
0xacdb, 0x1038, 0xab17, 0xf7ce, 0x8a9e, 0x9aa3, 0xb495, 0x8a5e, 0xd859, 0x0bac,
0xd0db, 0x3552, 0xa6b0, 0x727f, 0x28e4, 0xe5cf, 0x163c, 0x3411, 0x8f07, 0xfab7, 0x0f34,
0xdabf, 0x6f6f, 0xc598, 0xf496, 0x9a9a, 0xbd6a, 0x2136, 0x810a, 0xca55, 0x8bce, 0x2ac4,
0xddce, 0xdd06, 0xc4fc, 0xfb2f, 0xee5f,0xfd30, 0xc540, 0xd5f1, 0xbdad, 0x45c3, 0x708a,
0xa359, 0xf40d, 0xba06, 0xbace, 0xb447, 0x3f48, 0x899e, 0x8084, 0xbdb9, 0xa05a,
0xe225, 0xfb0c, 0xb2b2, 0xa4db, 0x8bf9, 0x12f7
 };
 n = 100;
 for (c = 1 ; c <= n - 1; c++) {
   d = c;
   while ( d > 0 && array[d] < array[d-1]) {
     t = array[d];
     array[d] = array[d-1];
     array[d-1] = t;
     d--;
   }
 }
 for (c = 0; c <= n - 1; c++) {
   printf("%d\n", array[c]);
 }
 return 0;
}
```

Insertion Sort in MIPS Assembly

```
      org 0x0000

       ori     $sp, $zero, 0x3ffc      # stack
       ori     $1, $zero, 0x190 # 100*4bytes numbers to be sorted #int n
       ori     $2, $zero, 0x0004 # for loop counter #int c
```

```
    ori     $3, $zero, sortdata # int array[]
    ori     $10, $zero, 0x0004 # used for [d-1]

outerLoop:
        beq     $2, $1, endOuterLoop
        addu    $4, $2, $zero # int d

innerLoop:
    beq     $4, $zero, endInnerLoop
        lw      $6, sortdata($4) # array[d]
        subu    $5, $4, $10
        lw      $7, sortdata($5) # array[d-1]
        sltu    $8, $6, $7 # array[d] < array[d-1]
        beq     $8, $zero, endInnerLoop
        sw      $6, sortdata($5) # store array[d] to location of array[d-1]
        sw      $7, sortdata($4) # store array[d-1] to location of array[d]
        subu    $4, $4, $10 # d--
        beq     $zero, $zero, innerLoop



endInnerLoop:
        addiu   $2, $2, 0x0004 # increment for loop count by 1
        beq     $zero, $zero, outerLoop

endOuterLoop:
        halt


    org 0x00F00
sortdata:
cfw 0x087d
cfw 0x5fcb
cfw 0xa41a
cfw 0x4109
cfw 0x4522
cfw 0x700f
cfw 0x766d
cfw 0x6f60
cfw 0x8a5e
cfw 0x9580
cfw 0x70a3
cfw 0xaea9
```

cfw 0x711a
cfw 0x6f81
cfw 0x8f9a
cfw 0x2584
cfw 0xa599
cfw 0x4015
cfw 0xce81
cfw 0xf55b
cfw 0x399e
cfw 0xa23f
cfw 0x3588
cfw 0x33ac
cfw 0xbce7
cfw 0x2a6b
cfw 0x9fa1
cfw 0xc94b
cfw 0xc65b
cfw 0x0068
cfw 0xf499
cfw 0x5f71
cfw 0xd06f
cfw 0x14df
cfw 0x1165
cfw 0xf88d
cfw 0x4ba4
cfw 0x2e74
cfw 0x5c6f
cfw 0xd11e
cfw 0x9222
cfw 0xacdb
cfw 0x1038
cfw 0xab17
cfw 0xf7ce
cfw 0x8a9e
cfw 0x9aa3
cfw 0xb495
cfw 0x8a5e
cfw 0xd859
cfw 0x0bac
cfw 0xd0db
cfw 0x3552
cfw 0xa6b0

cfw 0x727f
cfw 0x28e4
cfw 0xe5cf
cfw 0x163c
cfw 0x3411
cfw 0x8f07
cfw 0xfab7
cfw 0x0f34
cfw 0xdabf
cfw 0x6f6f
cfw 0xc598
cfw 0xf496
cfw 0x9a9a
cfw 0xbd6a
cfw 0x2136
cfw 0x810a
cfw 0xca55
cfw 0x8bce
cfw 0x2ac4
cfw 0xddce
cfw 0xdd06
cfw 0xc4fc
cfw 0xfb2f
cfw 0xee5f
cfw 0xfd30
cfw 0xc540
cfw 0xd5f1
cfw 0xbdad
cfw 0x45c3
cfw 0x708a
cfw 0xa359
cfw 0xf40d
cfw 0xba06
cfw 0xbace
cfw 0xb447
cfw 0x3f48
cfw 0x899e
cfw 0x8084
cfw 0xbdb9
cfw 0xa05a
cfw 0xe225
cfw 0xfb0c

cfw 0xb2b2
cfw 0xa4db
cfw 0x8bf9
cfw 0x12f7