Midterm Report

ECE 437

Computer Design and Prototyping

TA: Abhisek

3/4/11

Alex Reyes (mg225)

Matthew Barga (mg217)

Executive Overview

The goal of these FPGA designs were to implement functioning processors with support for a limited subset of the MIPS ISA. The initial designs called for a single cycle processor that would finish one instruction per clock cycle, with the exception of load instructions. This design was later built upon to create a pipelined processor capable of completing multiple instructions each clock cycle, thus improving the throughput over the single cycle processor. This report contains discussions on the design process followed for developing each processor, and in addition provides insights into the performance of each design and how the two designs compare and differ from each other.

The greatest success in the design came in placing the branch hardware in the 2nd stage of the pipelined processor. The final design was a success, but it came at a price as the critical path length was increased due to the extra hardware needed for forwarding branch address calculations from the ALU unit. There was a playoff between the increased performance due to the increased average CPI and the decrease in performance due to the increased critical path. The most beneficial insight into processor design was gained from the realization of this performance playoff and how to maximize overall performance by tailoring a processor design to different workloads.

Single Cycle Processor Design (Matthew Barga FIG. 1.0)

The first step in the design of the single cycle processor was to determine the target functionality, determined by the instruction set to be supported. Instructions were grouped by type and by similar hardware requirements. If more than one instruction required similar hardware they were considered as a functional group. An example of this is signed vs. unsigned versions of the same operation (e.g., SLT and SLTU) or branching instructions (e.g., BEQ and BNE). Once a rough outline of functionality was determined by grouping instructions, the actual hardware designs to implement each group of instructions were built using basic logic elements.

In designing the hardware, care was taken to modularize everything as much as was practical. This allowed some components to be reused or slightly modified to create new components. The best example of this was modifying the ALU to create 32-bit adders for use in address calculation. Another design choice that was made was to implement comparisons for branch logic using the ALU for subtraction and the zero flag of the ALU for control. This solution was the most elegant as it reduced the amount of new hardware necessary and saved the trouble of having to modify the ALU module to support comparison operations.

The final modules that were implemented were the CPU and ALU control modules. Another design decision in the case of these units was to include the controls for r-type instructions in the ALU control block. As the instruction opcode for r-types was encoded in the lowest 6 bits of the instruction and these bits were already being passed to the ALU control for determining the ALU opcode, the control signals associated with these instructions were also generated in the ALU control block for simplicity reasons.

Single Cycle Processor Design (Alex Reyes FIG. 1.1)

During the design of the single cycle processor some of the modules that allowed the

most design freedom with respect to their implementation were the ALU, the Control Unit and the PC. Apart from the separate modules there was some flexibility in the hardware implementation of the instruction set given for implementation.

The control unit was designed to generate all control signals, including ALU control signals. Generating all of the control signals from a single controller was chosen over the alternative of having a separate ALU controller for ease of implementation when connecting all of the modules together.

The PC register needed to be stopped both on a halt signal or on a pause signal from memory. The PC register was made as simple as possible with only one enable signal and the simple circuitry to control the enable signal was managed outside the module. This helped the development of the pipeline processor by the PC being able to be reused, regardless of how many signals pause or halt the PC register.

Pipelined Processor Design (FIG. 1.2)

The largest hurdles in the pipelined processor design were implementing hardware for allowing nops and bubbles in the pipeline, allowing for single memory read/write control and placement of branch control hardware. The problem of introducing bubbles and stalls in the pipeline was related to both of the other problems. The pipeline registers needed to be stalled if there was a forward dependency on a value that had yet to be loaded from memory. With memory latency enabled, an 'allwait' signal was sent to every pipeline register to stall them while waiting for a memory read to complete if needed. Nops also had to be introduced to one register after a branch was taken to flush the invalid instruction it contained. This was implemented using a nop enable signal that set all control signal values for that instruction to 0 when the nop enable was asserted.

One last design decision was to move branch logic to the 2^{nd} stage of the pipeline. This

involved forwarding the ALU result signal to the second stage if the branch calculation depended on an ALU result that had not been written to a register yet. For the pipelined design, most of the code from the single cycle designs was reused as the foundations of the designs were very similar. A consideration needed for the pipelined design that was not needed in the single cycle design was how to divide up the stages. This came up in placing branch logic and some multiplexers selecting among calculated values to pass forward. Looking back, more care should have been taken during this process to reduce the critical path length.

FIG. 1.0

Matthew's

Single Cycle

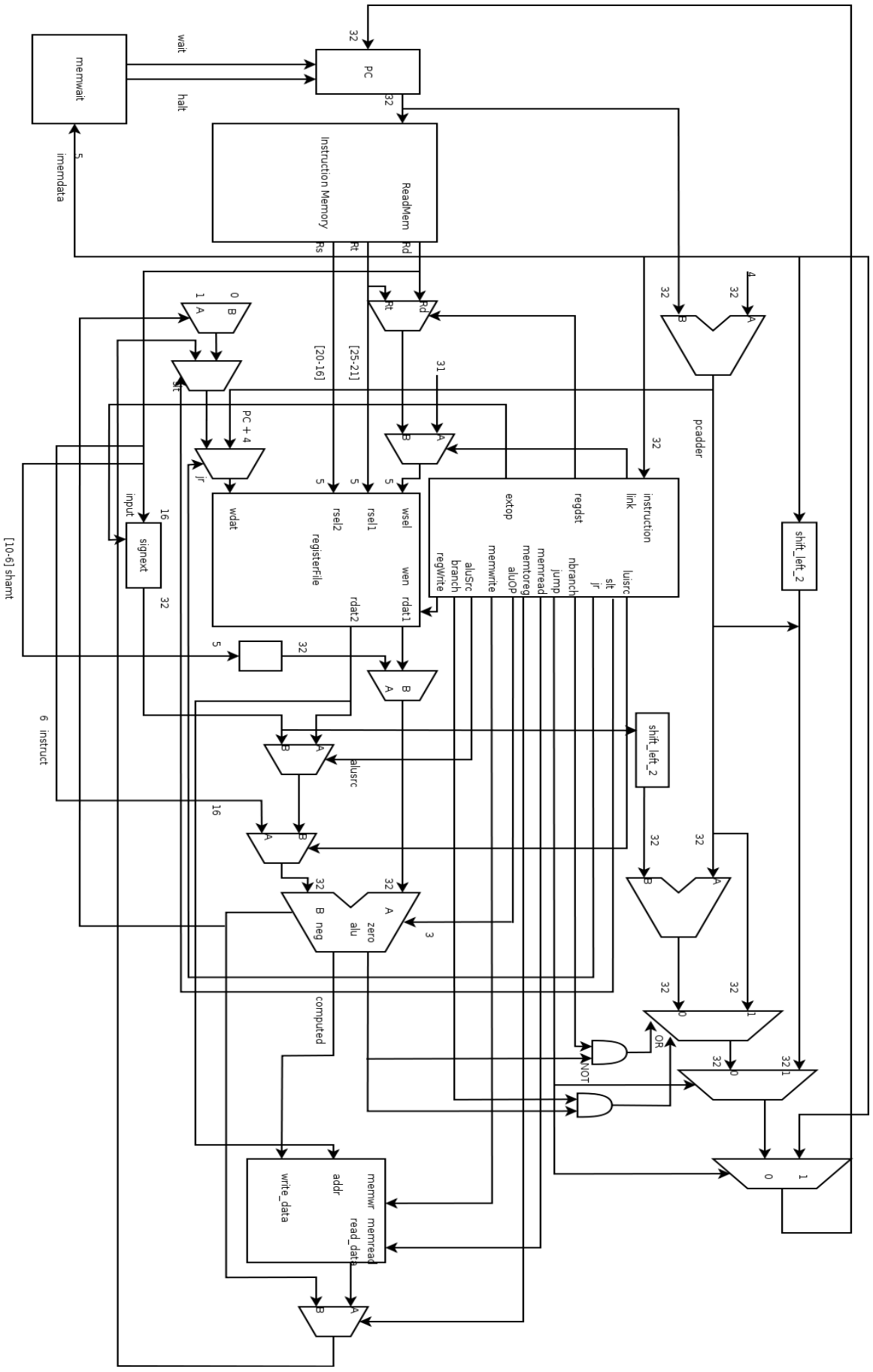


FIG. 1.1

Alex's

Single Cycle

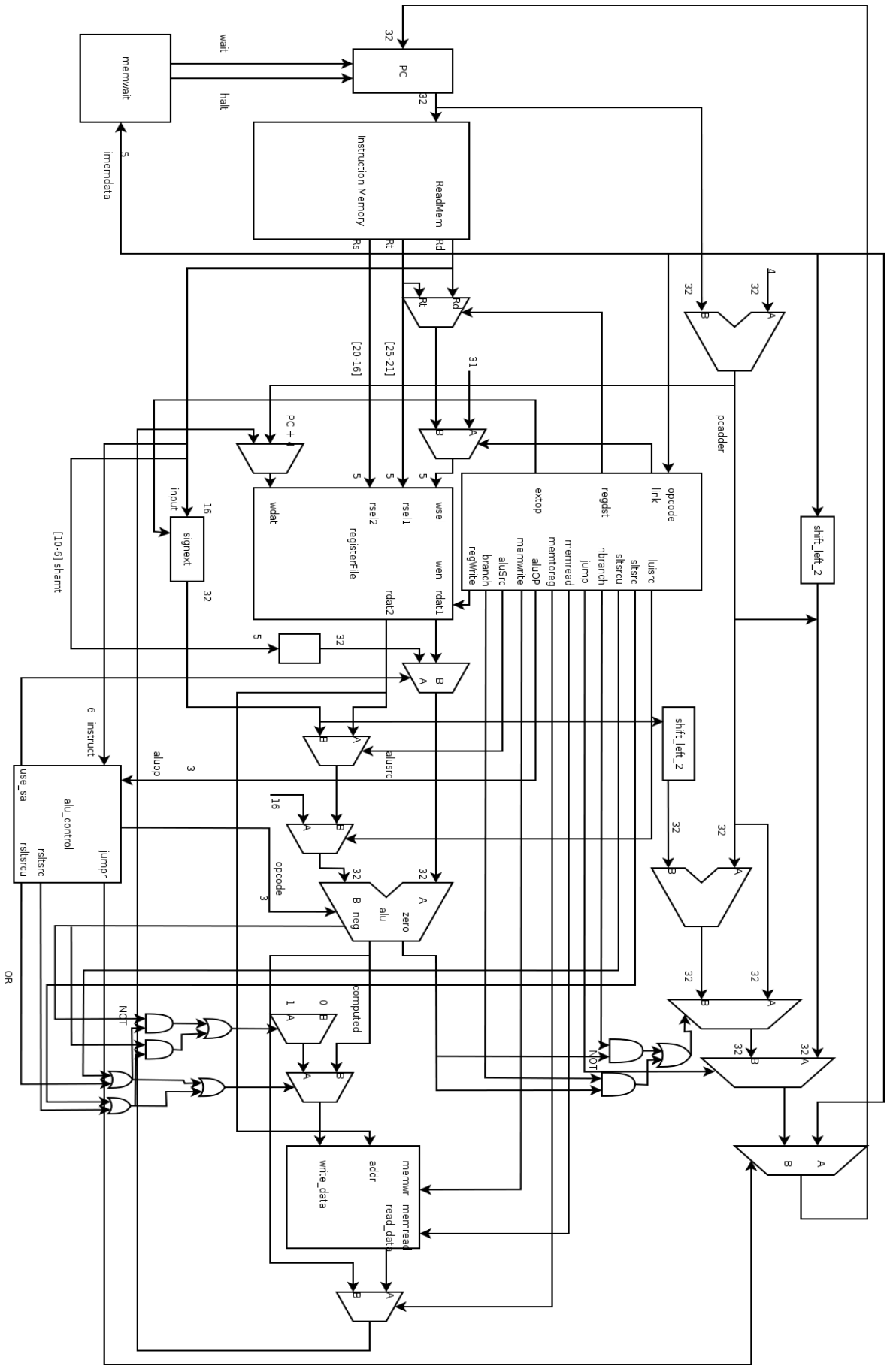
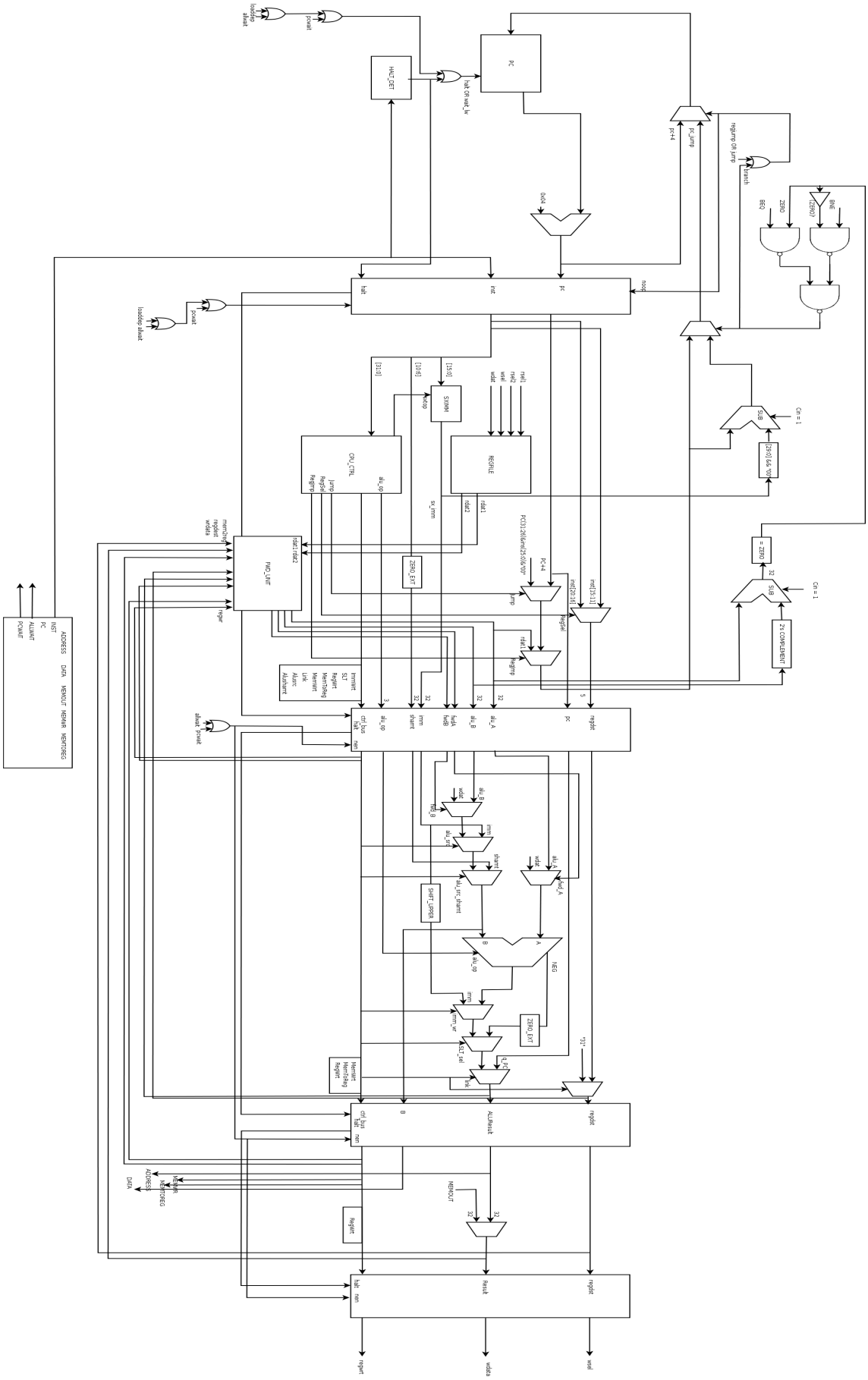


FIG. 1.2

Pipelined

Design



Results

For the single cycle and pipelined processor designs, benchmark calculations were based on run statistics for three separate program binaries. The performance metrics worth making note of are the MIPS, CPI and instruction latency measurements for each processor and each program binary. The MIPS (millions of instructions per second) number was calculated using the following formula:

$$MIPS = \frac{no.\ instructions}{1x10^6} \times \frac{1}{clock\ period \times total\ cycles \times sec.\ per\ cycle}$$

Instruction latency was calculated by taking the workload percentage for each instruction type (branch, load, others) and multiplying that percentage by the number of cycles needed for that instruction type to complete. This number was then multiplied by the clock cycle time for the design in question. This is shown in the following formula:

$$latency = (\%\ branch\ inst. \times 2\ cycles) + (\%\ load\ inst. \times 6\ cycles) + (\%\ other\ inst. \times 5\ cycles) \times \frac{sec}{cycle}$$

Finally, the CPI was determined by dividing the number of cycles needed for the program to complete by the number of instructions executed. All of these benchmark calculations are outlined in FIG. 2.0 in the appendix.

The single cycle processor designs separately required 3,737 and 3,569 logic elements and 1,233 and 1231 registers. Critical paths for these designs were 32ns and 35ns. These were substituted in as the processor clock period values for the calculations in the figure. The program counter calculation was where the bulk of the critical path was present in each design. The start of the paths were in the instruction fetches from memory, from where they propagated through the register file, an adder and then through a series of multiplexers back to the program counter register. The critical paths were hypothesized to be in the branch address calculation hardware as a 32 bit ripple-carry adder is present along with multiple

multiplexers selecting values from the register file and selecting among multiple program counter calculations to pass to the program counter register.

The final design to be analyzed was the pipelined processor. The block diagram for this design is FIG. 1.2 above. The same three program binaries used for the single cycle benchmark calculations were used for the pipeline design benchmarks as well. The benchmark specs for this design are outlined in FIG. 2.1 in the appendix.

The final chip area for the pipelined processor design required 4091 logic elements and 1463 registers. The critical path for this design was 40ns. The majority of the path is present in the ID stage where branch-taken control is generated, however the values used to determine the branch taken signal are forwarded from the ALU unit in the EXEC stage. The path begins at the output of the ALU unit, and propagates back to the program counter register. This path was hypothesized to be the critical path even before the analysis was completed as it propagates through two ripple-carry arithmetic units (ALU and subtractor) in addition to multiple other logical elements.

The theoretical execution times for each program binary calculated using a range of clock period values are charted below in FIG. 1.3 and FIG. 1.4. for the single cycle and the pipelined designs. The total execution time for each binary decreases at a linear rate for increasing clock frequency. This matches intuition as the execution time is the linear function:

$$execution\ time = no.\ prog.\ instructions \times Avg.\ CPI \times clock\ period$$

Theoretically, the execution time would tend to 0 if the clock frequency could increase to infinity. This however is not possible, as the critical path will determine the nominal clock period for any design and the critical path will always be non-zero.

FIG. 1.3

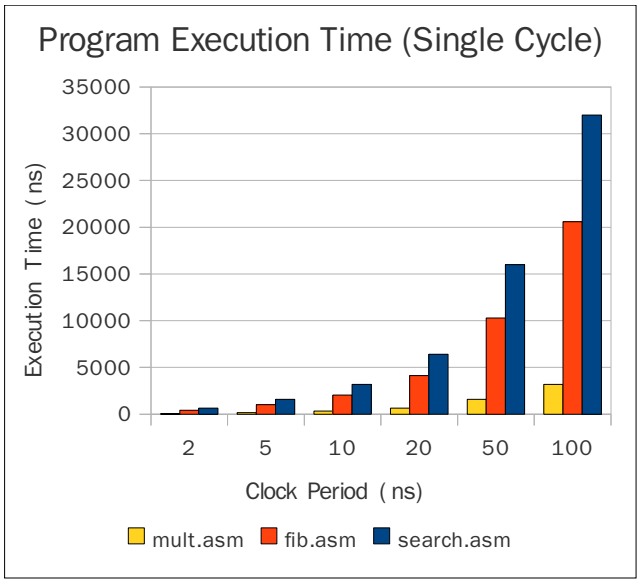
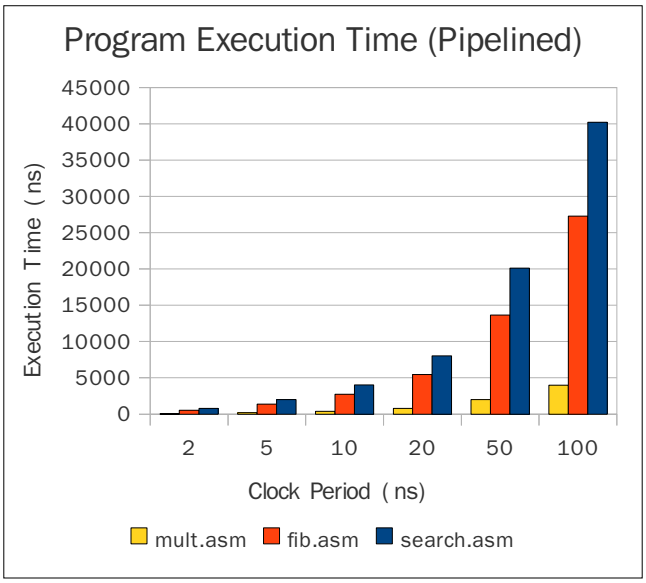


FIG. 1.4



In the benchmark analysis done in this report the single cycle processor designs had and increase of around 8-10MIPS for each program binary. This was contradictory to the predicted performance of the designs as the purpose of creating a pipelined design was to increase the throughput of the processor. However, the results show that the throughput decreased for the pipelined processor design. The reason for this result was the increased clock period time for the pipelined design. A performance gain in a pipelined design will only be observed if the clock period decreases below that of a single cycle design. This issue could be addressed by reducing the critical path for the pipelined design resulting in a faster clock period.

Conclusions

The three processor designs outlined in this report were able to execute all desired functionality without fault. The single cycle processors had reasonable critical path lengths and the benchmark specs showed similar performance between the two designs implemented. The pipelined design was also able to execute all desired functionality. The average CPI for the

pipelined design was also slightly over 1 for the tested workloads, which was the targeted number as it means that for the 5 stage design, on average a little less than 5 instructions complete per cycle. The critical path was however surprisingly large in the pipelined design due to the location of branch logic in the 2nd stage. This decrease in the number of wasted cycles for a branch taken did not compensate for the increase in clock period as the single cycle design was able to outperform the pipelined design in throughput and latency. More consideration should be given to reducing critical path time in future designs.

The main advantage of the single cycle processor design was the simplicity of it. There may theoretically be a performance increase for a pipelined design vs. a single cycle design, but this comes at the cost of hazard risks and hardware needed to solve the problems created by these hazards. The obvious advantage of the pipelined design is the increase in throughput that is a result of the pipeline architecture. The implementation of the pipelined design was however considerably more difficult than that of the single cycle.

Appendix

FIG 2.0

Performance Metrics for Single Cycle Design

Program Binaries	search.asm	fib.asm	mult.asm	
No. Instructions	278	160	29	
No. Clock Cycles	320	206	32	
Avg. IPC	0.87	0.78	0.91	
Avg. CPI	1.15	1.29	1.10	
Branch %	41.37%	13.75%	0.00%	
Load %	14.75%	28.12%	6.90%	
Others %	43.88%	58.13%	93.10%	
Avg. Inst. Latency (ns)	125.00	155.80	162.21	mbarga
	136.72	170.40	177.42	agreyes
MIPS	27.15	24.27	28.32	mbarga
	24.82	22.19	25.89	agreyes

	mbarga	agreyes
Clock Period (ns)	32	35
Clock Freq. (MHz)	31.25	28.57
No. Logic Elements	3737	3569

FIG 2.1

Performance Metrics for Pipelined Design

Program Binaries	search.asm	fib.asm	mult.asm
No. Instructions	278	160	29
No. Clock Cycles	402	273	40
Avg. IPC	0.69	0.59	0.73
Avg. CPI	1.45	1.71	1.38
Branch %	41.37%	13.75%	0.00%
Load %	14.75%	28.12%	6.90%
Others %	43.88%	58.13%	93.10%
Avg. Inst. Latency (ns)	156.26	194.75	202.76
MIPS	17.29	14.65	18.13

Clock Period (ns)	40.00
Clock Freq. (MHz)	25.00
Total Logic Elements	4091
Registers	1463

Bonus: MIPS Compiler

The following is the C source code for a bubble sort algorithm:

//bare.c

```
void sort(int count, char argv[]) {
  int i;
  char swapped, temp;

  swapped = 1;
  while(swapped == 1){
    swapped = 0;

    for(i = 1; i < count; i++){
      if(argv[i-1] > argv[i]){
        temp = argv[i-1];
        argv[i-1] = argv[i];
        argv[i] = temp;
        swapped = 1;
      }
    }
  }
}
```

This code was compiled with the mips-cc compiler and after modifying the generated assembly code to be compatible with this processor, the following assembly code was created:

//bare.asm

```
        org 0x0000

        ori     $13,$zero,100
        ori     $10,$zero,1
        ori     $14,$zero,0xB0

        beq     $zero,$zero,L2
        nop
```

```
#inside of while
L6:
ori     $10, $zero, 0
ori     $11, $zero, 1
beq     $zero,$zero,L3
nop

L5:

sll     $15,$11,2
addiu   $2,$15,-4
addu    $2,$2,$14
lw      $3, 0($2)

sll     $15,$11,2
addu    $2,$15,$14
lw      $2, 0($2)
nop
slt     $2,$2,$3

beq     $2,$0,L4
nop

sll     $15,$11,2
addiu   $2,$15,-4
addu    $2,$2,$14
lw      $12,0($2)
nop
sll     $15,$11,2
addu    $3, $15, $14
lw      $3,0($3)
nop
sw      $3,0($2)
nop
```

```
  sll     $15,$11,2
  addu    $3, $15, $14
  sw      $12,0($3)
  nop
  ori     $10,$zero,1

  L4:
  addiu   $11,$11,1

  # i < count
  L3:
  slt     $2,$11,$13
  bne     $2,$0,L5
  nop

  #while(swapped == 1)
  L2:
  ori     $2, $zero, 1
  beq     $10,$2,L6
  nop
halt

  org 0x00B0
  cfw 0x087d
   // data to sort
  cfw 0x5fcb
  cfw 0xa41a
  cfw 0x4109
  ..........
  cfw 0xb2b2
  cfw 0xa4db
  cfw 0x8bf9
  cfw 0x12f7
```