Implementation of a Two-Stage Pipelined Processor using RISC-V ISA and an Optimized Cache Organisazion for Sieve of Eratosthenes Algorithm

Niyana Yohannes
School of Electrical and Information Engineering, University of Sydney
nyoh0352@uni.sydney.edu.au
500 498 752

***Abstract* — Reduced Instruction Set Architecture (RISC) is an ISA that is known for its simplicity and extensibility allowing hardware optimization and easy pipelining. In this paper, a 2-stage pipelined processor with hazards resolved is developed using the RISC-V ISA. It is then tested on a program that implements the Sieve of Eratosthenes algorithm. Following this, a discussion on cache is had a an optimum cache configuration for the program on a RISCV is Found.**

I. Introduction

Instruction Set Architecture (ISA) defines the user observable behavior of a processor. It acts as an interface between the hardware and software, specifying what the processor is capable of doing as well as how it gets done. Of all the ISAs available, RISC and in particular, RISC-V is known for its simplicity and extensibility. This ISA has simpler hardware design, allowing hardware optimization and easy pipelining. For this reason, the 32-bit address space variant of RISC-V (RV32) was the ISA used for this processor. This report will discuss the implementation of a 2-staged pipelined processor using the RV32 ISA. Hazards arising from pipelining are also addressed through the implementation of Hardware Interlocks, Forwarding and Jump Kills. The processor was then tested on a program that implements the Sieve of Eratosthenes algorithm written in assembly language. Following this, a discussion on cache and the optimum cache organization for a RISCV processor running the Sieve of Eratosthenes algorithm program, takes place.

II. Background

As mentioned previously, the RISC-V ISA is used for the processor and for the test program implementing the Sieve of Eratosthenes algorithm. For a detailed discussion on the RISC-V ISA, see the background section of the report named 'Implementation of a 2-stage pipelined processor using RISC-V ISA' by Niyana Yohannes.

The Sieve of Eratosthenes algorithm is an ancient algorithm that is used to find all the prime numbers less than a given number T. It was conceived of by a scholar named Eratosthenes of Cyrene ca 240BC. There are three main steps to the algorithm.

1. Write down all the numbers from 1 to the given n. Composites will be eliminated by marking them. At the start, all the numbers are unmarked.
2. Number 1 is marked as a special number as it is neither prime nor composite
3. Set k = 1 and until k exceeds the square root of n loop through:
   - Find the first number in the list greater than k that has not been identified as composite. (The very first number found is 2.) Call it *m.* Mark all the multiples of m (these are non-prime).
   - m is a prime number, so leave it unmarked.
   - Set k=m and repeat.
4. Finally, all the prime numbers will be unmarked, and the prime numbers marked

The figure below shows this in c. An array of size n is filled with 0s. If an index of the array is a composite number, the 0 at that index is filled with a -1. Thus, at the end, the prime numbers can be found by looking at the indexes where the value at that index is 0.

```
void
sieve()
{
    int i, m, k;

    /* Mark the composites */
    /* Special case */

    for (i = 0; i < NLIMIT; i++)
          mark[i] = 0;

    mark[1] = -1;

    /* Set k=1. Loop until k >= sqrt(n) */
    for (k = 1; k <= KLIMIT; k = m)
    {
        /* Find first non-composite in list > k */
        for (m = k + 1; m < NLIMIT; m++)
            if (!mark[m])
                break;

        /* Mark the numbers 2m, 3m, 4m, ... */
        for (i = m * 2; i < NLIMIT; i += m)
            mark[i] = -1;
    }
}
```

Figure 1: Sieve of Eratosthenes Algorithm

III. Architecture

A. Program Characterization

The program that contains the Sieve of Eratosthenes (program.c) is converted to assembly language based of the RISCV ISA and stored in the file 'program.s'. This can be seen in Appendix F. The Sieve of Eratosthenes subroutine and all its callees can also be seen in Appendix F. As the processor only needs be able to execute the subroutine and its

callees, an ebreak is placed that ends the program before the main routine is reached. Also, to make testing easier, the program was adjusted to make NLIMIT = 10 (KLIMIT = 4). The modified program.c file that produces the program.s used for testing can be seen in Appendix G An explanation of how the algorithm works in assembly language is below. Figure 2 below shows the C variables used in the algorithm (program.c) and the RISCV registers where these variables are stored for the algorithm in assembly language (program.s)

RISC-V Register	C Equivalent
a0 / x10	NLIMIT
t1 / x6	KLIMIT
a2 / x12	NLIMIT - 1
a3 / x13	m & i
a5 / x15	k

Figure 2: C variables and RISCV registers they are stored in for NLIMIT=10 and KLIMIT=4

B. 2-Stage Pipelined Processor

Pipelining is an implementation technique that can increase the performance of a processor. It involves multiple instructions being executed simultaneously. Pipelining increases performance as it reduces the cycle time since there is less work to be done each stage. However, as will be seen, pipelining introduces hazards which need to be resolved. Pipelining is achieved using pipeline registers. For the 2-stage pipelined processor, the processor is split into 2 stages with the pipeline registers being placed in between the ID and EX stages. The final 2-stage pipelined processor can be seen in Figure 6.

The hazards that arise from pipelining are data hazards and control hazards. Data hazards refer to when an instruction depends on the result of a previous instruction and that result of instruction has not yet been computed. Read after Write (RAW) hazards which occur when a later read happens before an earlier write are an example of this. Appendix H is an example of this. When RAW hazards occur, the pipeline needs to be stalled and a nop inserted so the result of the instruction in the second stage can be computed and written back to the destination register. This is referred to as hardware interlocks. Another way to resolve RAW/data hazards in hardware is to route the data required as soon as possible after it is calculated to the earlier pipeline stage. This is known as forwarding. These were done in the processor by adding the control signals and muxes seen in the Figures 3, 4, and 5. The conditions which control these stall and forward signals can be seen in Appendix J and K**.**

Control hazards also arise from pipelining a processor. Control hazards occur when the decision of what instruction to fetch has not been made by the time the next instruction has been fetched resulting in the incorrect instruction being in the pipeline. These occur from branch and jump instructions. Appendix I is an example of this**.** Following a jump or branch (if true) instruction, the next instruction has already been fetched. However, since this is the wrong instruction, the processor can kill the instruction by inserting a nop in its place. This is not a stall as the pc is not stalled. This technique will also be implemented in the next section. This was also done in the processor by adding the control signals and muxes seen in the Figure 3 and 4. The condition which controls the kill signal can be seen in Appendix L**.**

Forwarding was used for ALU_OP and ALU_OP_IMM instructions, and Hardware Interlocks were used for LOAD instructions. Jump kills were used for jump and branch instructions.

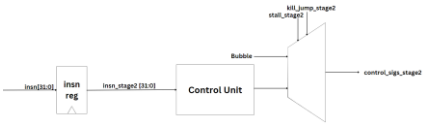


Figure 3: Hardware Interlock and Jump Kill Implementation

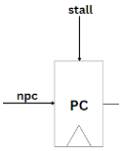


Figure 4: Stalling the PC, Hardware Interlock

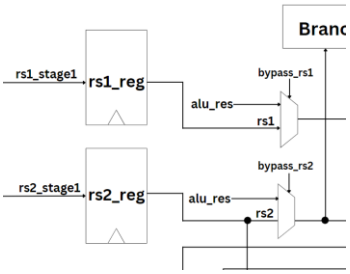


Figure 5: Forwarding Implementation

The program implementing the Sieve of Eratosthenes algorithm contains both data and control hazards. Thus, executing the program on the 2-stage pipelined processor will show if the hazard detection methods are working.

Lines 16 and 17 of program.c (See Appendix F) is an example of a RAW/Data Hazard. Appendix M shows how the processor implements forwarding to overcome this data hazard. The data that is required for the instruction in the first stage is forwarded as soon as it is calculated to the first stage pipeline stage.

Line 28 in program.c (See Appendix F**)** is an example of a Control Hazard. Appendix N shows how the processor ignores the incorrect instruction already loaded in the first stage, flushing it out and then continues with the instruction at the location where it has jumped.

C. Cache Aritechture (Including I and D access patterns for program)

Performance of high-speed computers is usually limited by memory bandwidth (number of accesses per unit time) and latency (time for a single access). This is what's referred to as the CPU-Memory Bottleneck. To limit this bottleneck, Cache memory is used. Cache is memory that is placed between the processor and main memory (usually on the same die as the

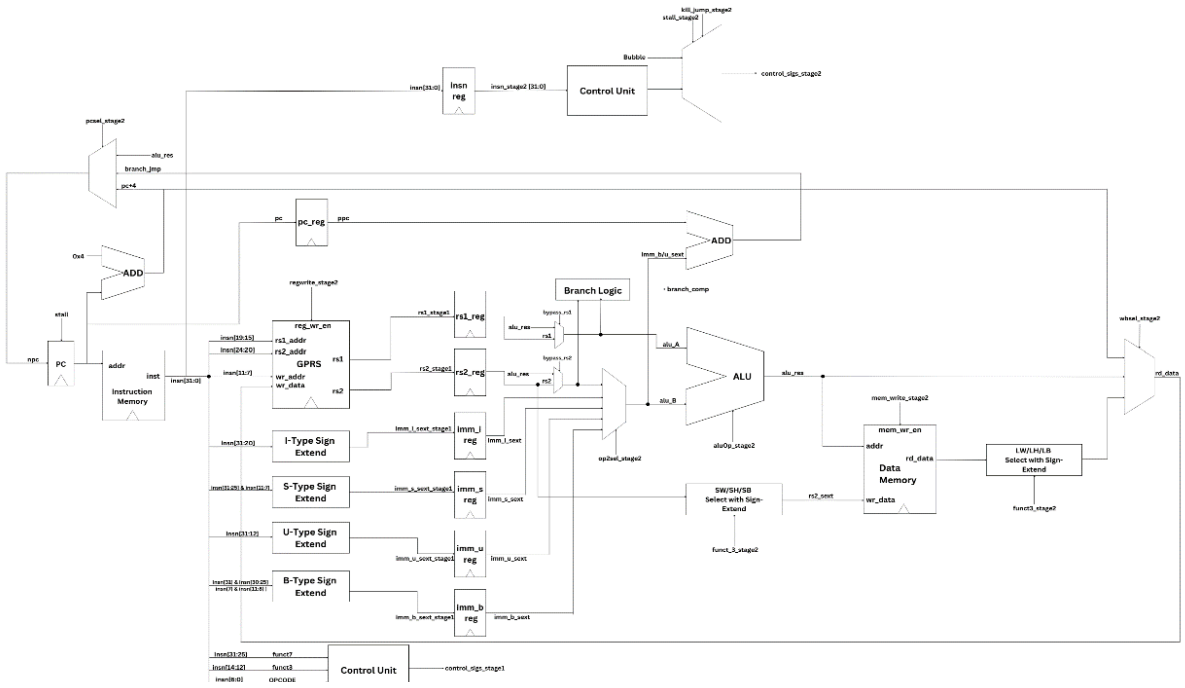


Figure 6: 2-Stage Pipelined Processor

CPU) and is high speed memory that holds a temporary copy of frequently used data. It usually has low latency and limited capacity (compared to main memory). A processor can have multilevel cache memory. Figure 7 below shows a cache memory structure. The three import cache configurations are Set (S), Ways (W), and Block size (B). Set is the line number, ways is the number of locations where each block can be placed, and B is the size of the blocks. Different configurations of these parameters result in different cache configurations, such as fully associate, n-way set associative, and direct-mapped.

RISCV uses separate caches for Instruction Memory (I-cache) and Data Memory (D-cache).

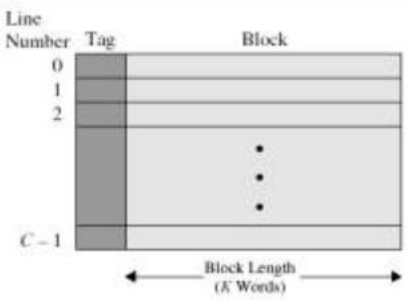


Figure 7: Cache Memory Structure

In choosing the cache configuration for this program, the memory access patterns of the program need to be assessed. As the program deals mainly with an array which is contiguous locations in memory, spatial locality of the program is high. Thus, for the D-cache, to exploit this, a high block size is required.

Also, as the algorithm deals mainly with loops, this means the same batch of instructions are referenced repeatedly. This results in a high level of temporal locality, Thus, for the I-cache, to exploit this, a high number of sets is required.

Thus, the final D and I cache parameters for the RISCV processor running the program should be a high block size for the D cache and a high number of sets for the I cache.

IV. Results

A. Processor Simulation on Program

The algorithm with NLIMIT set to 10 and KLIMIT set to 4 results in an array like Figure 8 below. The prime numbers (excluding 0^{th} index) which have a 0 at their index are 2, 3, 5, and 7. The non-prime numbers which are 1, 4, 6, 8, and 9, (also 10 which is not shown) which have a 1 at their index are the non-prime numbers. When the Sieve of Eratosthenes Algorithm in Assembly Language (RISCV ISA) is executed using the 2-Stage Pipelined Processor, it is not possible to read the array which is located in memory and compare it to the expected array. Thus, to be sure that the processor is executing the algorithm correctly, the data memory access patterns (data that is loaded and stored to memory and the number of times this occurs) as the processor is running through the program is compared to the data access pattern that is expected from the algorithm. To calculate the data access pattern that is expected from the algorithm, the algorithm is ran using c.

```
0 -1 0 0 -1 0 -1 0 -1 -1
```

Figure 8: Expected result for NLIMIT=10 and KLIMIT=4

Figure 9 below shows the data memory access pattern for the algorithm with NLIMIT = 10 and KLIMIT = 4. Data is stored in memory whenever an index of the array mark is changed (mark[i] = 0 and mark[i] = -1 in Figure 1) and data is read from memory whenever the contents of the array mark is required (if(!mark[m]) in Figure 1). When executing the program (in Assembly Language) using the 2-stage pipelined processor, the same data memory access pattern and actual contents of the data is expected.

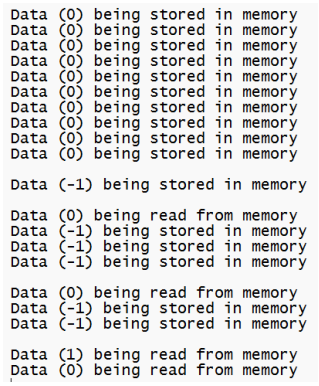


Figure 9: Data memory access pattern for NLIMIT=10 and KLIMIT=4

When running the algorithm on the 2-stage Pipelined Processor, the same memory accesses are seen.

Firstly, as can be seen from Appendix A, mem_wr_enable is enabled and data (0 / 0x0000000) is written to memory (memory location of array) 10 times. This corresponds to the first 10 data memory accesses in Figure 9. Following this, as can be seen in Appendix B, mem_wr_enable is enabled and data (-1 / 0xFFFF0000) is written to memory at the address at the start of the array. This corresponds to the first 'Data (-1) being stored in memory' memory access in Figure 9. Following this, as can be seen from Appendix C, mem_rd_enable is enabled and data (0 / 0x00000000) is read from memory once. Then, mem_wr_enable is enabled and data (-1 / 0xFFFFFFFF) is written to memory 3 times. This corresponds to the read followed by the 3 stores memory access in Figure 9. Following this, as can be seen from Appendix D, mem_rd_enable is enabled and data (0) is read from memory once. Then, mem_wr_enable is enabled and data (-1 / 0xFFFFFFFF) is written to memory 2 times. This corresponds to the read followed by the 2 stores memory access in Figure 9. Following this, as can be seen from Appendix E mem_rd_enable is enabled and data (1 then 0) is read from memory twice. This corresponds to the final 2 read memory access in Figure 9.

Thus, it is clear to see how the 2-stage Pipelined Processor correctly executed the Sieve of Eratosthenes algorithm in Assembly Language (RISCV ISA). The 2-stage pipelined processor executed the program correctly 136 cycles.

B. Cache Simulations

To perform cache simulations for the program using a RISCV processor, the Spike simulator is used. As mentioned previously, a RISCV processor has a separate D and I Cache. For the cache simulations, the NLIMIT was returned to 100, and the KLIMIT to 11.

Firstly, the simulator was ran using the cache parameters provided, which was ic=2:4:8 and dc=2:4:8 (S:W:B). This results in the cache performance seen in the Figure below. Assuming a hit time of 1 cycle and a miss penalty of 10 cycles, the Average Memory Access Time (AMAT) can be calculated to be 3.2213 for the D Cache and 2.9065 for the I Cache. This can be improved.

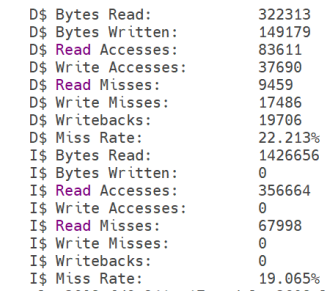


Figure 10: Cache Performance for ic=dc=2:4:8

Like Lab 7, assuming memory costs ($0.1 + $0.0001*b) where b is cache size in bytes, a D and I Cache configuration can be found that minimizes AMAT*memory cost. The script to do this can be seen in Appendix O**.** This results in a D and I Cache with parameters 1:8:512 and 8:8:128 respectively. Running the simulator with these parameters, results in the performance seen below. This results in AMAT of 0.0597 for the D Cache and 0.0117 for the I Cache. This is a major improvement as the miss rates have improved drastically.

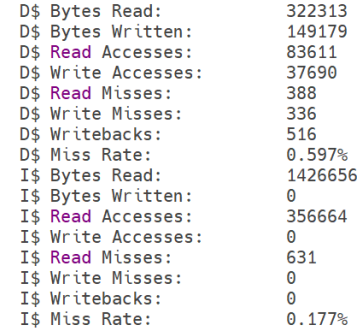


Figure 11: Cache Performance for ic=8:8:128 & dc=1:8:512

V. Discussion and Conclusion

The program successfully ran on the 2-stage pipelined processor. To reduce the number of cycles it takes, branch prediction could have been implemented. For the cache simulation, the final cache parameters of 1:8:512 (D) and 8:8:128 (I) align with what was discussed previously. The D cache has a significantly large block size than the I cache as expected and the I cache has a larger number of sets than the D cache as expected. However, the assumptions that of 1 cycle for the hit time and 10 cycles for the miss penalty may be incorrect. More accuracy for these would produce a more accurate representation of the best cache parameters for the program.

In conclusion, a 2-stage pipelined processor with hardware interlocks, forwarding and jump kills resolved resulted in the Sieve of Eratosthenes program being executed correctly. Also, the 'best' cache configuration for a RISCV processor that runs the program was determined.

VI. References

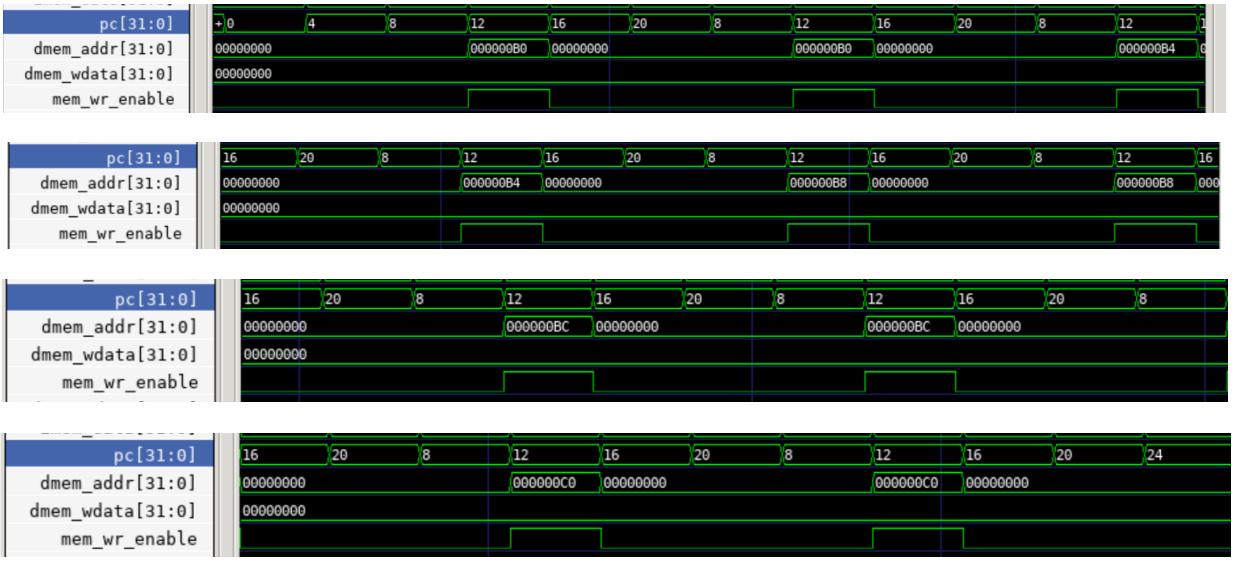
[1] Leong, P. H. W, "ELEC3608 Computer Architecture – Pipelining," Lecture Slides, w5.

[2] David A. Patterson, and John L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface", Chapter 4.

APPENDICES

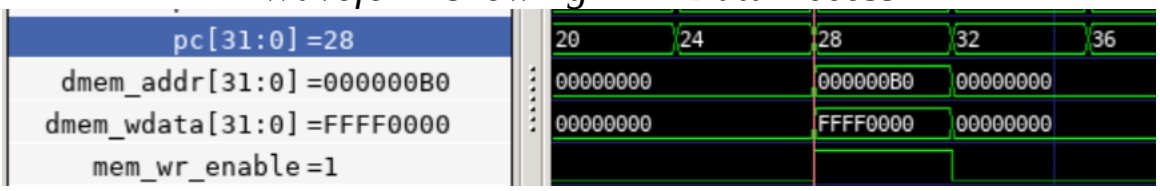
Appendix A

Waveform showing first 10 Data Accesses



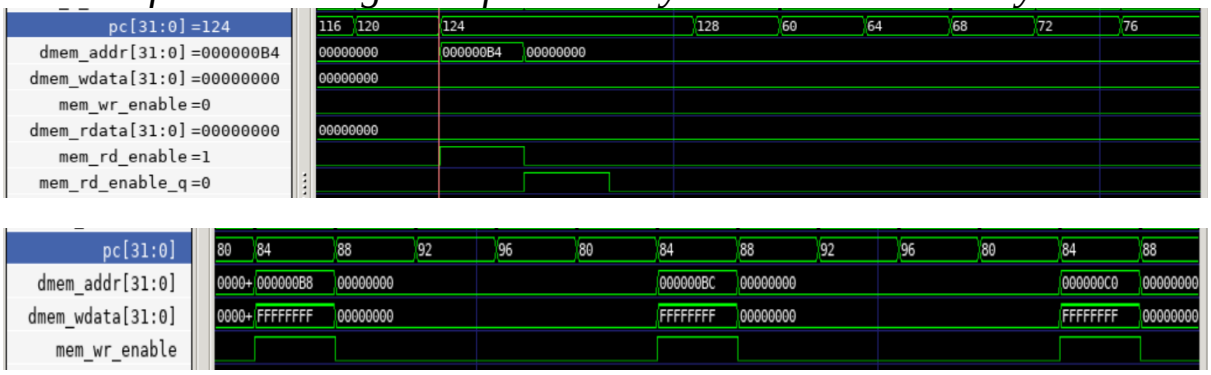
Appendix B

Waveform showing 11th Data Access



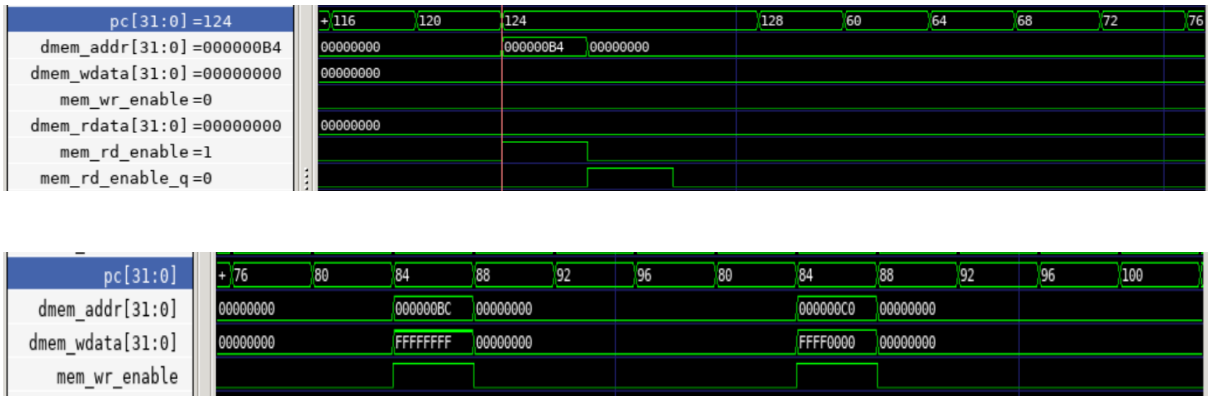
Appendix C

Waveform showing read followed by the 3 stores memory access



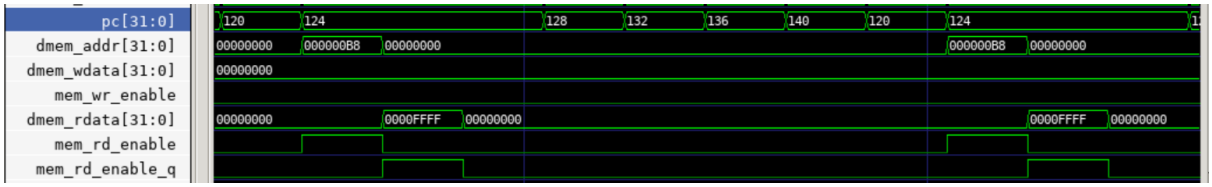
Appendix D

Waveform showing read followed by 2 stores memory access



Appendix E

Waveform showing last 2 memory accesses



Appendix F

Sieve of Eratosthenes Program in Assembly Language for NLIMIT=10 and KLIMIT=4

```
        .file "program.c"
        .option nopic
        .attribute arch, "rv32i2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align      2
        .globl      sieve
        .type sieve, @function
sieve:
        lui   a5,%hi(.LANCHOR0)
        addi  a5,a5,%lo(.LANCHOR0)
        addi  a4,a5,20
.L2:
        sh    zero,0(a5)
        addi  a5,a5,2
        bne   a5,a4,.L2
        lui   a5,%hi(.LANCHOR0+2)
        li    a4,-1
        sh    a4,%lo(.LANCHOR0+2)(a5)
        li    a5,1
        li    a2,9
        lui   a7,%hi(.LANCHOR0)
        addi  a7,a7,%lo(.LANCHOR0)
        li    a6,-1
        li    a0,10
        li    t1,4
        j     .L7
.L8:
        mv    a5,a3
.L3:
        slli  a3,a5,1
        bgt   a3,a2,.L5
        mv    a1,a3
        slli  a4,a3,1
        add   a4,a7,a4
.L6:
        sh    a6,0(a4)
        add   a3,a3,a5
        add   a4,a4,a1
        ble   a3,a2,.L6
.L5:
        bgt   a5,t1,.L13
.L7:
        addi  a3,a5,1
        bgt   a3,a2,.L8
        slli  a5,a3,1
        add   a4,a7,a5
        mv    a5,a3
.L4:
        lhu   a3,0(a4)
        beq   a3,zero,.L3
        addi  a5,a5,1
        addi  a4,a4,2
        bne   a5,a0,.L4
        j     .L5
.L13:
        ebreak
        .size sieve, .-sieve
        .align      2
        .globl      main
        .type main, @function
main:
        addi  sp,sp,-16
        sw    ra,12(sp)
        call  sieve
        li    a0,0
        lw    ra,12(sp)
        addi  sp,sp,16
        jr    ra
        .size main, .-main
        .globl      mark
        .bss
        .align      2
        .set  .LANCHOR0,. + 0
        .type mark, @object
        .size mark, 20
mark:
        .zero 20
        .ident      "GCC: () 12.2.0"
```

Appendix G

Sieve of Eratosthenes Program in c for NLIMIT=10 and KLIMIT=4 that Generates Appendix F

```c
#include <stdio.h>

#define NLIMIT      10      /* maximum number to scan */
#define KLIMIT      4       /* should be sqrt(NLIMIT)+1 */

/* Usage: Sieve n
 * where n = largest integer to test for prime
 * (default = 1000)
   Ref: http://www.utm.edu/research/primes
 */

unsigned short   mark[NLIMIT];              /* array to mark sieve values */

void
sieve()
{
    int i, m, k;

    /* Mark the composites */
    /* Special case */

    for (i = 0; i < NLIMIT; i++)
        mark[i] = 0;

    mark[1] = -1;

    /* Set k=1. Loop until k >= sqrt(n) */
    for (k = 1; k <= KLIMIT; k = m)
    {
        /* Find first non-composite in list > k */
        for (m = k + 1; m < NLIMIT; m++)
            if (!mark[m])
                break;

        /* Mark the numbers 2m, 3m, 4m, ... */
        for (i = m * 2; i < NLIMIT; i += m)
            mark[i] = -1;
    }
}

int
main()
{
    //  int i;

    sieve();
    // /* Now display results - all unmarked numbers are prime */
    //  for (i = 1; i < NLIMIT; i++)
    //  {
    //      if (!mark[i])
    //          printf("%d ", i);
    //  }
    //  printf("\n");


    return 0;
}

```

Appendix H
Example of a Data Hazard

```
i1. R2 <- R5 + R3
i2. R4 <- R2 + R3
```

Appendix I
Example of a Control Hazard

Appendix J
Stall Condition

```
//stall condition
if ( ((insn_rs1 == insn_rd_stage2) && (next_wr == 1) && (re1_enable == 1)) || ((insn_rs2 == insn_rd_stage2) && (next_wr == 1) && (re2_enable == 1)) ) begin
    stall = 1;
end
```

Appendix K
Forwarding Condition

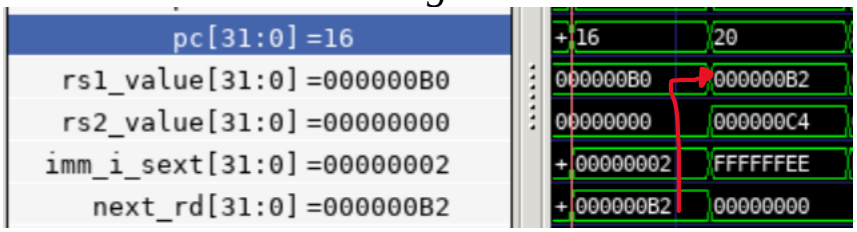
```
//bypass condition
if ((insn_rs1 == insn_rd_stage2) && (next_wr) && (re1_enable == 1))  begin
    bypass_rs1 = 1;
end
else if (((insn_rs2 == insn_rd_stage2) & (next_wr) && (re2_enable == 1))) begin
    bypass_rs2 = 1;
end
```

Appendix L
Kill Condition

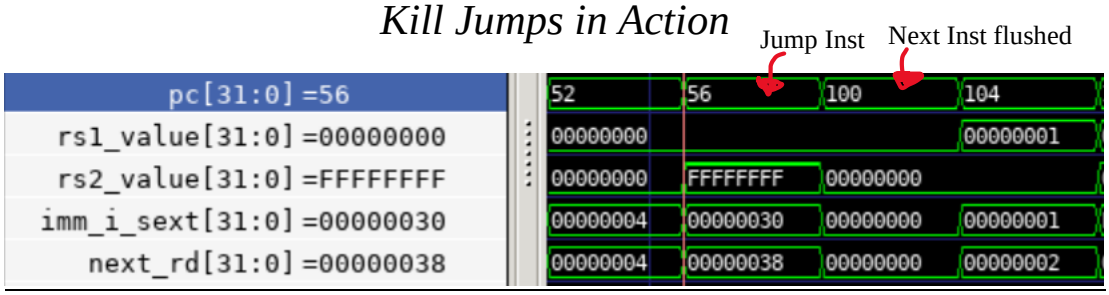
```
301             // Jump And Link (unconditional jump)
302             OPCODE_JAL: begin
303                 next_wr = 1;
304                 next_rd = ppc+4;
305                 npc = ppc + imm_j_sext;
306                 kill_jmp = 1;
307                 if (npc & 32'b 11) begin
308                     illinsn = 1;
309                     npc = npc & ~32'b 11;
310                 end
311             end
312             // Jump And Link Register (indirect jump)
313             OPCODE_JALR: begin
314                 kill_jmp = 1;
315                 case (insn_funct3_stage2)
316                     3'b 000 /* JALR */: begin
317                         next_wr = 1;
318                         next_rd = ppc+4;
319                         npc = (rs1_value + imm_i_sext) & ~32'b 1;
320                     end
321                     default: illinsn = 1;
322                 endcase
323                 if (npc & 32'b 11) begin
324                     illinsn = 1;
325                     npc = npc & ~32'b 11;
326                 end
327             end
```

```
328             // branch instructions: Branch If Equal, Branch Not Equal, Branch Less Than, Branch Greater Than, Branch Less Than Unsigned, Branch Greater Than Unsigned
329             OPCODE_BRANCH: begin
330                 case (insn_funct3_stage2)
331                     3'b 000 /* BEQ  */: begin if (rs1_value == rs2_value) begin npc = ppc + imm_b_sext; kill_jmp = 1; end end
332                     3'b 001 /* BNE  */: begin if (rs1_value != rs2_value) begin npc = ppc + imm_b_sext; kill_jmp = 1; end end
333                     3'b 100 /* BLT  */: begin if ($signed(rs1_value) < $signed(rs2_value)) begin npc = ppc + imm_b_sext; kill_jmp = 1; end end
334                     3'b 101 /* BGE  */: begin if ($signed(rs1_value) >= $signed(rs2_value)) begin npc = ppc + imm_b_sext; kill_jmp = 1; end end
335                     3'b 110 /* BLTU */: begin if (rs1_value < rs2_value) begin npc = ppc + imm_b_sext; kill_jmp = 1; end end
336                     3'b 111 /* BGEU */: begin if (rs1_value >= rs2_value) begin npc = ppc + imm_b_sext; kill_jmp = 1; end end
337                     default: illinsn = 1;
338                 endcase
339                 if (npc & 32'b 11) begin
340                     illinsn = 1;
341                     npc = npc & ~32'b 11;
342                 end
343             end
```

Appendix M
Forwarding in Action



Appendix N
Kill Jumps in Action



Appendix O

*Script to get cache parameters that minimizes AMAT*Memory Cost*

```
import subprocess

d_missrate = []
i_missrate = []
cache_size = []
cache_params = []

for i in range(3,20):
    B = str(2**i)

    for j in range(20):
        S = str(2**j)

        for z in range(4):
            W = str(2**z)

            i_cache = "--ic="+S+":"+W+":"+B
            d_cache = "--dc="+S+":"+W+":"+B
            result = subprocess.run(["spike", "--isa=rv32i", i_cache, d_cache, "/opt/riscv/riscv32-unknown-elf/bin/pk", "program"], capture_output=True)
            result_str = str(result.stdout)
            result_arr = result_str.split("\\n")

            for res in result_arr:
                if "D$ Miss Rate: " in res:
                    d_missrate_arr = res.split()[-1] #Gets the miss rate
                    d_missrate_float = float(d_missrate_arr[:-1]) #Removes the %
                    d_missrate.append(d_missrate_float)
                if "I$ Miss Rate: " in res:
                    i_missrate_arr = res.split()[-1] #Gets the miss rate
                    i_missrate_float = float(i_missrate_arr[:-1]) #Removes the %
                    i_missrate.append(i_missrate_float)

            cache_params.append([int(S), int(W), int(B) ])
            cache_size.append(int(B)*int(S)*int(W))

d_minimized = []
i_minimized = []
for i in range(len(cache_size)):
    mem_cost = 0.1+0.0001*cache_size[i]
    d_AMAT = 1 + 10*d_missrate[i]
    i_AMAT = 1 + 10*i_missrate[i]
    d_minimized.append(mem_cost*d_AMAT)
    i_minimized.append(mem_cost*i_AMAT)

d_final_minimized = min(d_minimized)
d_final_minimized_index = d_minimized.index(d_final_minimized)
d_final_cache_param = (cache_params[d_final_minimized_index])

i_final_minimized = min(i_minimized)
i_final_minimized_index = i_minimized.index(i_final_minimized)
i_final_cache_param = (cache_params[i_final_minimized_index])

print(d_final_cache_param)
print(i_final_cache_param)

```