EECE7352 - COMPUTER ARCHITECTURE: HOMEWORK 3

ANNA DEVRIES[1]

23 October 2019

CONTENTS

LIST OF FIGURES

[1] *Department of Electrical and Computer Engineering, Northeastern University, Boston, United States*

LIST OF TABLES

1 PART A

1.1 Pipeline comparison

Description: For any two current embedded processors (e.g., ARM7, ARM9, Intel 8051, TI MSP430, MIPS32), provide the details of the microarchitecture of the pipeline (e.g., the width of the data path, the number of pipeline stages, number of instructions issued per cycles, the number of integer units, the branch resolution unit, the maximum number of instructions in flight, etc.). You might not find every one of these details, but provide at least 3 different pipeline characteristics that you can compare.

1.2 Solution

ARM7 (specifically ARM7TDMI) consists of a three-state pipeline – fetch, decode, execute – and utilizes a 32-bit datapath [2]. The datapath contains 31 general-purpose 32-bit registers, and seven dedicated 32-bit registers coupled to a barrel-shifter, ALU and multiplier as integer units [2]. The majority of instructions (register-to-register) execute within a single cycle; however, memory access instructions require 2 cycles per instruction [2]. ARM7TDMI branches either forwards or backward of up to 32MB and preserves the address of the instruction following the branch in the Link Register [2].
TI MSP430 is similar to ARM7. TIMSP430 also consists of three pipeline stages (fetch, decode, execute) and executes one instruction per clock cycle for register-to-register instructions [8]. TI MSP430, however, consists of a 16-bit datapath containing 12 general-purpose registers and 4 special purpose registers (PC, SP, status register and constant generator) [8]. Additionally, TI MSP430 has a single arithmetic logic unit for addition, subtractions and logical operations; and it handles branches utilizing a group of decision instructions which change the PC based on contents in the status register [8].

2 PART B

2.1 Superscalar vs. superpipelining tradeoffs

Read the paper on superscalar vs. superpipelining by Jouppi and Wall, and the paper on the optimal pipeline depth by Hartstein and Puzak. Then using these 2 papers, discuss the tradeoffs between these two different microarchitectural approaches. You should develop an argument for using one of these two microarchitectures based on your analysis.

2.2 Solution

Superscalar and superpipeline are two methods to improve uni-processor performance through instruction level parallelism (ILP). Superscalar machines issue multiple instructions per clock cycle; however, to fully capitalize on this feature, there must be n instructions capable of executing at all times shown in Figure 1 [10]. Inability to execute all n instructions at once results in stalls and dead time [10]. Superscalar machines capitalize on parallel instruction rate.

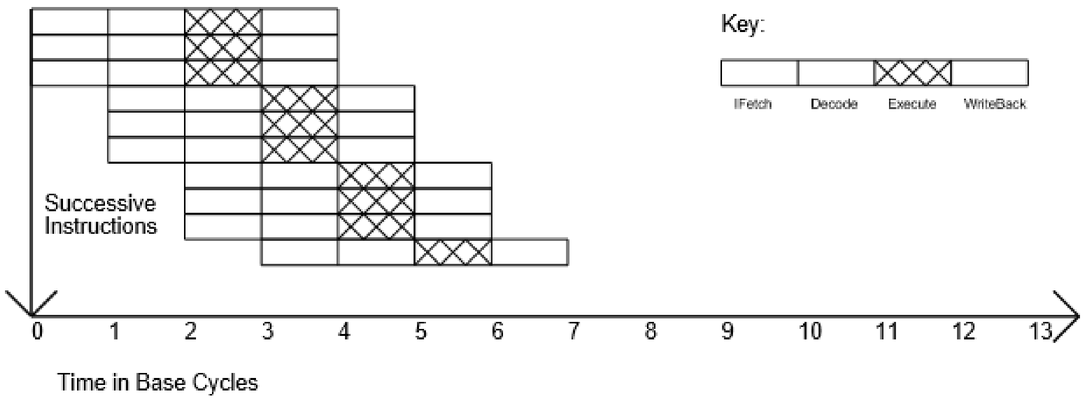


Figure 1: Execution on superscalar machines.

Superpipelined machines issue only one instruction per cycle, but have shorter cycle lengths then functional units such that the cycle time is 1/m of the base machine shown in Figure 2 [10]. Superpipelined machines capitalize on sequential instruction rate.

In the experiment by Wall and Jouppi, several benchmarks tested on the superpipelined and superscalar machines. The experiment ignored all effects of cache misses and system effects (i.e. interrupts and TLB misses), and removed class conflicts to create an ideal superscalar machine [10]. The results yielded a very similar performance between the two architectures as program degree increased; superscalar slightly outperformed superpipelined architecture, shown in Figure 3 [10].

Ideally, both of these architecture designs would have infinite performance gain by an increase in clock cycle or additionally functional units; however, in reality, both of these ILP techniques are limited. I contend that while superscalar outperforms superpipeline on an ideal machine, superpipeline is a better approach for most real systems due to latency, hardware availability and pipeline hazards.

To begin, the experiment by Jouppi and Wall showed that the superscalar architecture slightly outperformed the superpipeline architecture relative to the base machine across different program degrees [10]. Additionally, Jouppi and Wall mention the limitations of instruction-level parallelism. They explained that ILP averages around two without loop unrolling, limiting performance gain by superscalar and su-

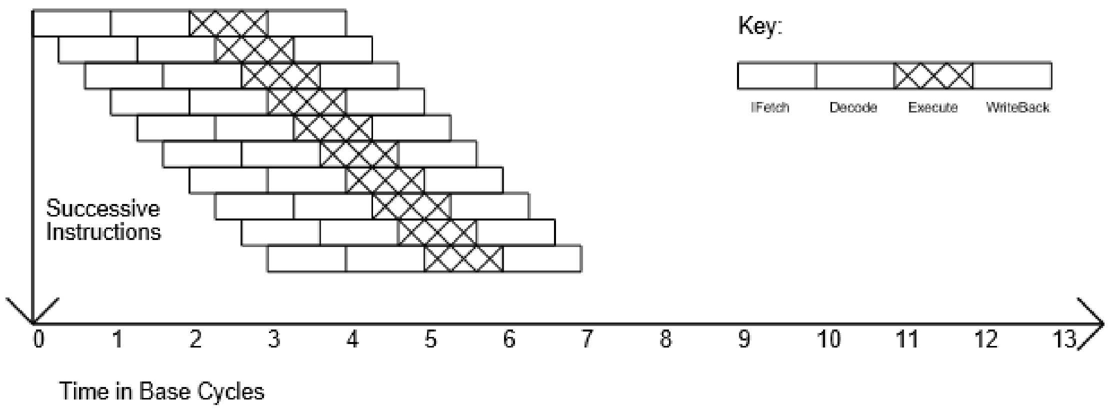


Figure 2: Execution on superpipelined machines.

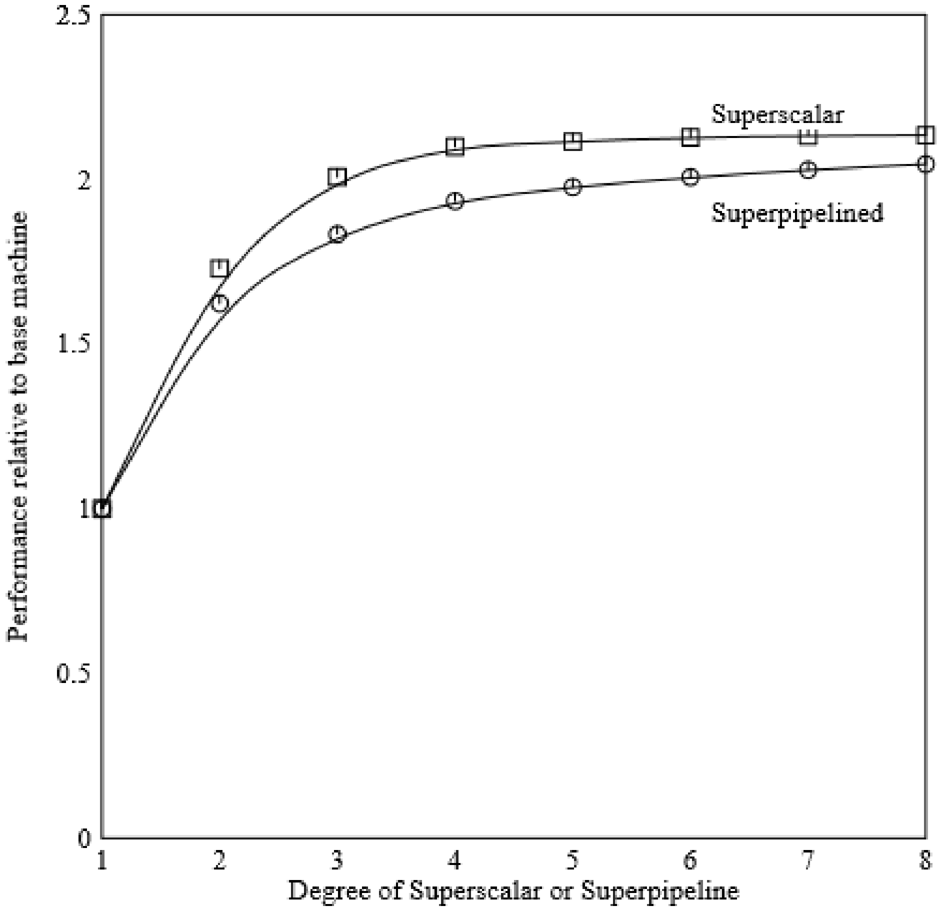


Figure 3: [Relative performance of superpipelined and superscalar architectures versus program degree.

perpipeline architectures around degree three [10]. At degree three, superscalar has a relative performance of approximately 2 and superpipeline has a relative performance of approximately 1.8 [10].

While superscalar appears to initially outperform superpipeline, this experiment completely ignores latency. Latency decreases benefits from ILP, especially in superscalar, by increasing time of each functional unit. Superscalar executes multiple instructions at once, any increased latency in one instruction will result in stalls or dead time in other instructions. While these stalls also affect superpipeline, superpipeline does not run multiple instruction at once. If one instruction takes longer, it will not require other instructions to stall as they execute in parallel. This greatly affects performance in superscalar architecture and removes some of its advantages.

Next, ILP architecture design is limited by hardware availability. Pipeline stages cannot be easily added to off-the-self components, forcing superscalar architecture to utilize multiple chips to increase the functional units available [10]. On the other hand, some chips have slower off-chip signaling and may require superscalar organization [10]. While using a slower off-chip signaling will decrease superpipeline performance, it still would not require additionally hardware configuration/purchase. Superpipeline can more easily be implemented with chips, and does not require extra or specialized chips.

Finally, pipeline hazards create overhead, reducing efficiency of pipeline architecture. Pipeline depth is linearly proportional to stalls, creating a greater penalty with larger pipelines [6]. This performance competition between throughput and hazards limits the effects of superscalar architecture. A superscalar architecture can no longer take full advantage of its design by increasing its depth due to the negative affects it has on pipeline hazards.

While superscalar architecture initially outperforms superpipeline architecture, the addition of latency and pipeline hazards even their relative performance. Furthermore, the hardware availability renders superpipeline architecture both a cheaper and more accessible design.

3 PART C

3.1 Branch predictor

We are providing you with an instruction trace named itrace.out.gz (the file has been compressed with gzip, so you will need to decompress it before using it). The trace simply lists the 64-bit instruction address for each instruction executed (addresses are in hexadecimal). You are working with an x86-64 instruction trace, so you will need to figure out which addresses are taken conditional branch addresses. There are multiple ways to accomplish this. You will be writing a simulator to simulate two simple conditional branch predictors.
The first predictor will have 32, 1-bit predictor, saving the action of the last branch to predict the outcome of the next branch. The second predictor will use 16, 2-bit counters, to predict branches. Assume initially that the 2-bit counters are initialized to be weakly not-taken.
Write a simple simulator that models each of the 2 predictors and processes the instruction trace provided. Submit both the code for your branch predictor simulator (on Turnitin) and report on the number of buffer misses (first time taken branches), and the number of correct and incorrect predictions for this address for this address trace for each predictor. Add a discussion explaining why one predictor does better than the other.
As an added step, experiment with other prediction algorithms (besides a 2-bit counter). You only have 8 entries in total for these alternative algorithms. The best performing algorithm will receive a special prize.

3.2 Solution

Table 1 shows the results of hits/misses for each predictor and the number of buffer misses. The final 32 1-bit predictor history table is [0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1]. The final 16 2-bit counter table is ['00', '00', '01', '10', '01', '10', '01', '10'].
The 2-bit counter is more accurate then the 1-bit history table. The largest shortcoming of the 1-bit prediction is almost every misprediction comes in pairs [7]. For example, if the program is in a nested loop, it may take many branches before not-taking a branch. Then within the next loop, a similar pattern may occur. A 1-bit prediction switches the prediction bit whenever there is a misprediction. In this scheme, the 1-bit prediction will produce two misses for each not-taken branch. Whereas, the 2-bit counter uses patterns and program behavior to make predictions. It acts similar to a finite state machine. If it is strongly taken, the counter will require two misses in a row to predict not-taken. For the nested loop example above, a 2-bit counter would only cause a single misprediction per not-taken branch.

Table 1: 1-bit and 2-bit branch predictor results.

Prediction	Buffer Misses	Hits	Misses
1-Bit Predictor	363	1016567 (98.88%)	9824 (1.12%)
2-Bit Predictor	363	1016567 (99.04%)	9824 (0.96%)

```
# Prediction based on a finite state machine. Prediction is either strong taken, weakly taken,
    weakly not taken
# or strongly not taken.
# 2-bit counter history stored (up to 16 bits) in a FIFO queue
# 11 = strongly taken, 10 = weakly taken, 01 = weakly not-taken, 00 = strongly not taken
def two_bit_counter(line,prev,branches,q2,hits2,misses2,total_branches2):
# Initialize variables
```

```python
    element = 0
# If 16-bit queue is empty, the prediction automatically guesses Weakly Not-Taken and adds "1" (01) to queue
    if len(q2) == 0:
        q2.append(1)
# If 16-bit queue is full, the first element (oldest element) is removed from the queue - FIFO. Numbers stored as integers 0-3 so q2 length must be 8 or less
    if len(q2) >= 8:
        q2.pop(0)
# 2-Bit Counter is initialized as last element in the queue
    counter = q2[len(q2)-1]
# 2-Bit Counter will guess taken if it is Weakly/Strongly Taken (10 or 11)
    if counter > 1:
        guess = 1
# 2-Bit Counter will guess not-taken if it is Weakly/Strongly Not-Taken (01 or 00)
    elif counter <= 1:
        guess = 0
# If the current line address equals the line address in the dictory at the previous line's prediction
# Then the prediction is considered a hit and 2-Bit Counter is updated.
    if line == branches[prev][guess]:
# If 2-bit Counter is strongly taken (11) and prediction is a hit, 2-Bit Counter remains 11
        if counter == 3:
            element = 3
# If 2-bit Counter is weakly taken (10) and prediction is a hit, 2-Bit Counter updates to be 11
        elif counter == 2:
            element = 3
# If 2-bit Counter is weakly not-taken (01) and prediction is a hit, 2-Bit Counter updates to be 00
        elif counter == 1:
            element = 0
# If 2-bit Counter is strongly not-taken (00) and prediction is a hit, 2-Bit Counter remains 00
        elif counter == 0:
            element = 0
# Prediction is added to queue
        q2.append(element)
        hits2 += 1
# Otherwise, prediction is considered a miss. 2-Bit Counter is updated.
    elif line != branches[prev][guess]:
# If 2-bit Counter is strongly taken (11) and prediction is a miss, 2-Bit Counter updates to be 10
        if counter == 3:
            element = 2
# If 2-bit Counter is weakly taken (10) and prediction is a miss, 2-Bit Counter updates to be 01
        elif counter == 2:
            element = 1
# If 2-bit Counter is weakly not-taken (01) and prediction is a miss, 2-Bit Counter updates to be 10
        elif counter == 1:
            element = 2
# If 2-bit Counter is strongly not-taken (00) and prediction is a miss, 2-Bit Counter updates to be 01
        elif counter == 0:
            element = 1
        q2.append(element)
        misses2 += 1
    total_branches2 += 1
    return (hits2,misses2,total_branches2)

# Prediction based on previous branch decision. If the last branch was taken, prediction assumes taken again.
# If last branch was not taken, prediction assumes not taken again.
# Branch history (up to 32 bits) is stored in a branch history table which pops off elements as FIFO
def one_bit_predictor(line,prev,branches,q,hits,misses,total_branches):
```

```
# If 32—bit queue is empty, the prediction automatically guesses Taken and adds "1" to the
    history table
    if len(q) == 0:
        q.append(1)
# If 32—bit queue is full, the first element (oldest element) is removed from the queue — FIFO
    if len(q) >= 32:
        q.pop(0)
# If the current line address equals the line address in the dictory at the previous line's
    prediction
# Then the prediction is considered a hit
    if line == branches[prev][q[len(q)—1]]:
        element = q[len(q)—1]
# Last guess (taken/not taken) is added to the branch history table
        q.append(element)
        hits += 1
# Otherwise, prediction is considered a miss. Last element in branch history table is swapped
# If element previously taken, the branch history table has a new element that is not taken. And
    vice versa.
    else:
        element = abs(q[len(q)—1]—1)
        q.append(element)
        misses += 1
    total_branches += 1
    return (hits,misses,total_branches)

# Finds all conditional branches in itrace
# Maps conditional branch addresses with possible return addresses
def find_branches():
# Opens itrace and initializes functions with zero addresses and no previous lines
    f = open('itrace.out', 'r')
    addresses = {}
    prev = None
    for line in f:
        line = line.strip()
# If first line in itrace, address is added to address dictory. Prev is set to current address
        if prev is None:
            addresses[line] = set()
            prev = line
            continue
# If not the first line in itrace, address is added to address dictory with its previous line
# This links each address with the next address (determining return address/addresses of each
    unique address)
        addresses[prev].add(line)
        if line not in addresses:
            addresses[line] = set()
        prev = line

    f.close()
    branches = {}
# Remove unconditional branches and normal instruction flow
    for k in addresses.keys():
        if len(addresses[k]) > 1:
            branches[k] = list(addresses[k])
    return branches

def main():
    branches = find_branches()
    f = open('itrace.out', 'r')
# Initializes variables
    hits = 0
    misses = 0
    total_branches = 0
    hits2 = 0
    misses2 = 0
    total_branches2 = 0
    flag = False
# Creates queues (FIFO) for each predictor (32 for 1-bit and 16 for 2-bit)
    q = []
    q2 = []
# For each line in itrace, flag is set when conditional branch is determined
# For each conditional branch, the simulator utilizes a predictor and logs hits/misses/total
    branches
    for line in f:
        line = line.strip()
        if flag:
            hits,misses,total_branches = one_bit_predictor(line,prev,branches,q,hits,misses,
    total_branches)
            hits2,misses2,total_branches2 = two_bit_counter(line,prev,branches,q2,hits2,misses2,
    total_branches2)
            flag = False
# If current line is in branches, then line is a conditional branch. Flag is set.
        if line in branches:
            flag = True
        prev = line
# Prints results/accuracy of each predictor
    print("Total Hits:     {} ({}%)".format(hits, (float(hits)/float(total_branches) * 100)))
    print("Total Misses:   {} ({}%)".format(misses, (float(misses)/float(total_branches) * 100)))
    print("Total Branches: {}".format(total_branches))
    print("32 1-bit Predictor History Table:    {}".format(q))
# Converts 16-bit counters to binary for appearance
    n = 0
    for i in q2:
        q2[n] = '{0:0b}'.format(i)
        n += 1
    print("Total Hits:     {} ({}%)".format(hits2, (float(hits2)/float(total_branches2) * 100)))
    print("Total Misses:   {} ({}%)".format(misses2, (float(misses2)/float(total_branches2) *
    100)))
    print("Total Branches: {}".format(total_branches2))
    print("16 2-bit Predictor History Table:    {}".format(q2))
    print("Number of buffer misses (Unique Branches):   {}".format(len(branches)))

    f.close()

if __name__=="__main__":
    main()
```

Algorithm 1: Simulator

4 PART D

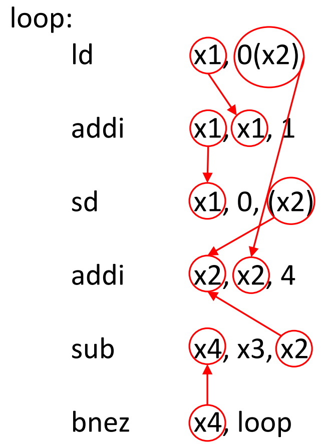
4.1 Textbook problems

To build a better understanding of pipelining concepts, complete Problems C.1 (parts ag), C.2 (parts a-b) and C.7 (parts a-b) in the textbook in Appendix C. Make sure to state any assumptions made when answering these problems.

4.2 Solution

C.1

a)



```
loop:
        ld      x1, 0(x2)
        addi    x1, x1, 1
        sd      x1, 0, (x2)
        addi    x2, x2, 4
        sub     x4, x3, x2
        bnez    x4, loop
```

	Register	SRC Instr.	DST Instr.
1)	x1	ld	addi
2)	x1	addi	sd
3)	x2	ld	addi
4)	x2	sd	addi
5)	x2	sub	addi
6)	x4	bnez	sub

C.1

b) F = Fetch; D = Decode; X = Execute; M = Memory; W = Writeback

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
loop:	ld	x1,0(x2)	F	D	X	M	W													
	addi	x1,x1,1		F	stall	stall	D	X	M	W										
	sd	x1,0,(x2)					F	stall	stall	D	X	M	W							
	addi	x2,x2,4								F	D	X	M	W						
	sub	x4,x3,x2									F	stall	stall	D	X	M	W			
	bnez	x4,loop												F	stall	stall	D	X	M	W
loop:	ld	z1,0(x2)																	F	D

X3 = x2+396 initially
16 cycles for each loop w/o overlapping
18 cycles for last loop

$$\frac{396}{4} = 99 \qquad \text{so 99 loops total}$$

$$\left(\frac{16\ cycles}{loop} * 98 loops\right) + 18 cycles = 1{,}568\ cycles$$

C.1

c) F = Fetch; D = Decode; X = Execute; M = Memory; W = Writeback

			1	2	3	4	5	6	7	8	9	10	11	12	13	14
loop:	ld	x1,0(x2)	F	D	X	M	W									
	addi	x1,x1,1		F	D	stall	X	M	W							
	sd	x1,0,(x2)			F	stall	D	X	M	W						
	addi	x2,x2,4					F	D	X	M	W					
	sub	x4,x3,x2						F	D	X	M	W				
	bnez	x4,loop							F	stall	D	X	M	W		
(incorrect prediction)											F	stall	stall	stall	stall	
loop:	ld	z1,0(x2)										F	D	X	M	W

X3 = x2+396 initially

9 cycles for each loop w/o overlapping

12 cycles for last loop

$$\frac{396}{4} = 99 \qquad \text{so 99 loops total}$$

$$\left(\frac{9\ cycles}{loop} * 98\ loops\right) + 12\ cycles = 894\ cycles$$

C.1

d) F = Fetch; D = Decode; X = Execute; M = Memory; W = Writeback

			1	2	3	4	5	6	7	8	9	10	11	12	13
loop:	ld	x1,0(x2)	F	D	X	M	W								
	addi	x1,x1,1		F	D	stall	X	M	W						
	sd	x1,0,(x2)			F	stall	D	X	M	W					
	addi	x2,x2,4					F	D	X	M	W				
	sub	x4,x3,x2						F	D	X	M	W			
	bnez	x4,loop							F	stall	D	X	M	W	
loop:	ld	z1,0(x2)									F	D	X	M	W

X3 = x2+396 initially

8 cycles for each loop w/o overlapping

12 cycles for last loop

$$\frac{396}{4} = 99 \qquad \text{so 99 loops total}$$

$$\left(\frac{8\ cycles}{loop} * 98\ loops\right) + 12\ cycles = 796\ cycles$$

C.1

e) F = Fetch; D = Decode; X = Execute; M = Memory; W = Writeback

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
loop:	ld	x1,0(x2)	F1	F2	D1	D2	X1	X2	M1	M2	W1	W2										
	addi	x1,x1,1		F1	F2	D1	D2	stall	stall	stall	X1	X2	M1	M2	W1	W2						
	sd	x1,0,(x2)			F1	F2	D1	stall	stall	stall	D2	X1	X2	M1	M2	W1	W2					
	addi	x2,x2,4				F1	F2	stall	stall	stall	D1	D2	X1	X2	M1	M2	W1	W2				
	sub	x4,x3,x2					F1	stall	stall	stall	F2	D1	D2	stall	X1	X2	M1	M2	W1	W2		
	bnez	x4,loop									F1	F2	D1	stall	D2	X1	X2	M1	M2	W1	W2	
loop:	ld	z1,0(x2)										F1	F2	stall	D1	D2	X1	X2	M1	M2	W1	W2

X3 = x2+396 initially

10 cycles for each loop w/o overlapping

19 cycles for last loop

$$\frac{396}{4} = 99 \qquad \text{so 99 loops total}$$

$$\left(\frac{10\ cycles}{loop} * 98\ loops\right) + 19\ cycles = 999\ cycles$$

C.1
f)

5 Stage Pipeline:

0.8ns + 0.1ns = 0.9ns

10 Stage Pipeline:

(0.8ns/2) + 0.1ns = 0.5ns

C.1

g)

5 Stage Pipeline:

$$CPI = \frac{total\ cycles}{total\ \#\ of\ instructions} = \frac{796\ cycles}{(6x99)} = 1.34067$$
$$\approx 1.34\ cycles/instructions$$
$$Average\ Instruction\ Execution\ Time = (CPI)(Cycle\ Time)$$
$$= (1.34)(0.9) = 1.206\ \ \approx 1.21\ ns$$

10 Stage Pipeline:

$$CPI = \frac{total\ cycles}{total\ \#\ of\ instructions} = \frac{999\ cycles}{(6x99)} = 1.681$$
$$\approx 1.68\ cycles/instructions$$
$$Average\ Instruction\ Execution\ Time = (CPI)(Cycle\ Time)$$
$$= (1.68)(0.5) = 0.84\ ns$$

C.2

a)

Jumps and calls 1% +1 cycles

1	2	3	4		
	1				
		1	2	3	4

Conditional Taken (60%)(15%) = 9% +2 cycles

1	2	3	4			
	1					
			1	2	3	4

Conditional Not Taken (40%)(15%) = 6% +1 cycles

1	2	3	4		
	1				
		1	2	3	4

$$\begin{aligned}
&CPI\ of\ Total\ Pipeline \\
&= Ideal\ CPI + Jump\ Call\ CPI + Conditional\ Taken\ CPI + Conditional\ Not\ Taken\ CPI \\
&= 1 + (0.01)(1) + (0.09)(2) + (0.06)(1) = 1.25
\end{aligned}$$

$$Speedup = \frac{1}{1.25} = 0.8$$

C.2

b)

Jumps and calls 1%

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15					
	1																		
					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

+4 cycles

Conditional Taken (60%)(15%) = 9%

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15										
	1																							
										1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

+9 cycles

Conditional Not Taken (40%)(15%) = 6%

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15					
	1																		
					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

+4 cycles

$$\begin{aligned}&CPI\ of\ Total\ Pipeline\\&= Ideal\ CPI + Jump\ Call\ CPI + Conditional\ Taken\ CPI + Conditional\ Not\ Taken\ CPI\\&= 1 + (0.01)(4) + (0.09)(9) + (0.06)(4) = 2.09\end{aligned}$$

$$Speedup = \frac{1}{2.09} = 0.4785 \approx 0.48$$

C.7

a)

$$Speedup = \frac{Execution\ Time\ old}{Execution\ Time\ new}$$

$$Execution\ Time = CPI\ * Cycle\ Time\ * Instructions$$

$$\frac{Execution\ Time\ 5\ stage}{Execution\ Time\ 12\ stage} = \frac{\frac{6}{5} * 1 * Instr}{\frac{11}{8} * 0.6 * Instr} = 1.45$$

b)

$$\text{CPI } 12\ Stage = (0.2 * 0.05 * 5) + \frac{11}{8} = 1.425 \approx 1.43$$

$$\text{CPI } 5\ Stage = (0.2 * 0.05 * 2) + \frac{6}{5} = 1.22$$

$$Speedup = \frac{1.22 * 1 * Instr}{1.43 * 0.6 * Instr} = 1.421911 \approx 1.42$$

5 PART E

5.1 Binary translation

The following is x86 assembly. Write equivalent code in C and RISC-V assembly. (It will probably be easier for you to write the C code first, and then the RISC-V assembly equivalent.) For C code, include a mapping between the variables you use and the x86 registers; for RISC-V code, include a mapping between the RISC-V registers you use and the x86 registers. Your code should run on the BRISC-V simulator. Remember to only use the RV32I instruction subset.

5.2 Solution

```
    .text
    .globl main
main:
    movl $0x0, %ebx
    movl $0x0, %ecx
    jmp  here
tloop:
    mov  %ecx, %eax
    add  %eax, %ebx
    addl $0x1, %ecx
here:
    cmpl $0x63, %ecx
    jle  tloop
ret
```

Algorithm 2: x86 assembly code

Below are my translations between the x86 to C code and x86 to RISC-V code. The register maps are in comments within each respective code.

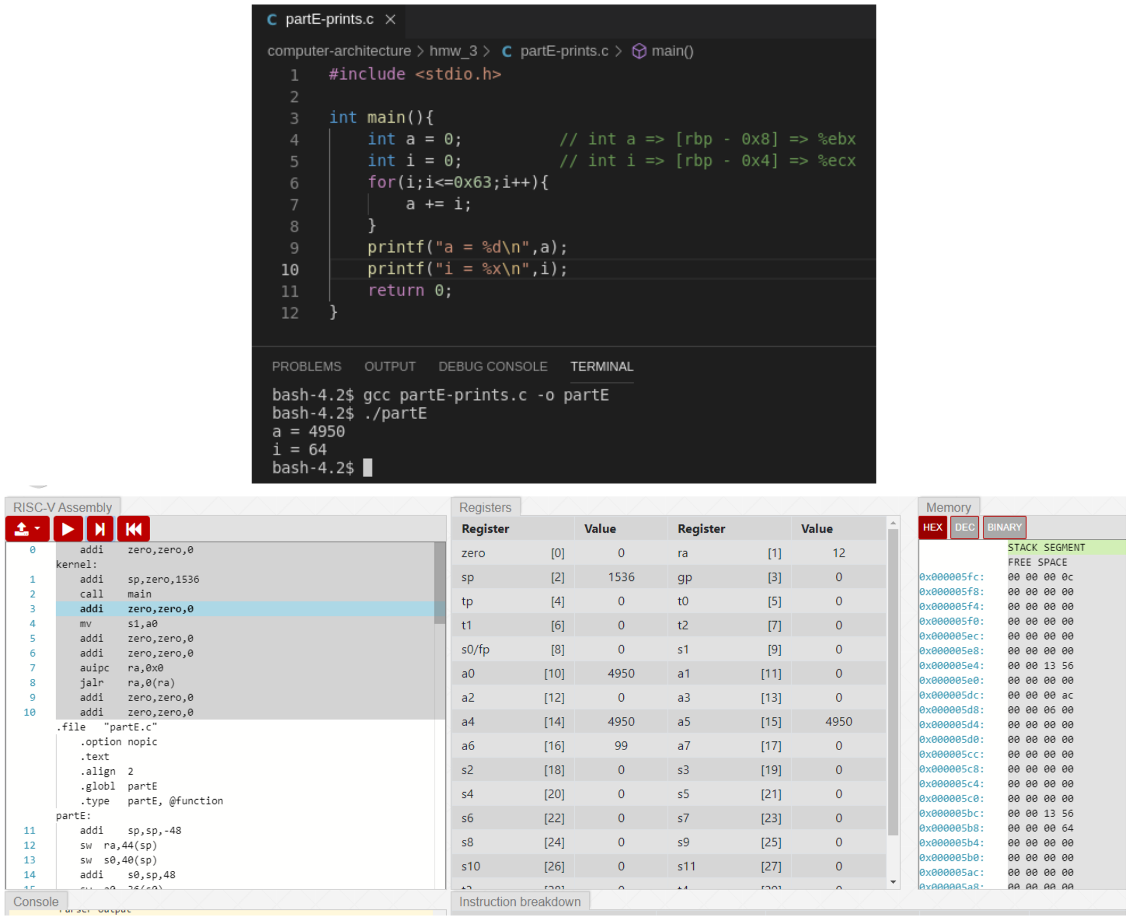
```
int main(){        // Register Mappings
    int a = 0;          // int a => [rbp - 0x8] => %ebx
    int i = 0;          // int i => [rbp - 0x4] => %ecx
    for(i;i<=0x63;i++){
        a += i;
    }
    return 0;
}
```

Algorithm 3: C code

```
.file "partE.c"
  .option nopic
  .text
  .align  2
  .globl  partE
  .type partE, @function
partE:
  addi  sp,sp,-48
  sw  ra,44(sp)
  sw  s0,40(sp)
  addi  s0,sp,48
  sw  a0,-36(s0)
  sw  a1,-40(s0)
.L2:
  lw   a4,-36(s0)
  lw   a5,-40(s0)
  li   a6,99
  add  a4,a5,a4
  addi  a5,a5,1         ;  Register  Mappings
  sw   a4,-36(s0)         ; a4 => -36(s0) => %ebx
  sw   a5,-40(s0)         ; a5 => -40(s0) => %ecx
  ble  a5,a6,.L2
  j  .L3
.L3:
  lw   a5,-36(s0)
  mv   a0,a5
  lw   ra,44(sp)
  lw   s0,40(sp)
  addi  sp,sp,48
  jr   ra
  .size  partE, .-partE
  .align   2
  .globl   main
  .type  main, @function
main:
  addi  sp,sp,-32
  sw   ra,28(sp)
  sw   s0,24(sp)
  addi  s0,sp,32
  li   a5,0
  sw   a5,-20(s0)
  li   a5,0
  sw   a5,-24(s0)
  lw   a1,-24(s0)
  lw   a0,-20(s0)
  call  partE
  sw   a0,-28(s0)
  lw   a5,-28(s0)
  mv   a0,a5
  lw   ra,28(sp)
  lw   s0,24(sp)
  addi  sp,sp,32
  jr   ra
  .size  main, .-main
  .ident  "GCC: (GNU) 7.2.0"
```

Algorithm 4: RISC-V code

Both codes are verified and demonstrated below in Figure 4.



```c
#include <stdio.h>

int main(){
    int a = 0;          // int a => [rbp - 0x8] => %ebx
    int i = 0;          // int i => [rbp - 0x4] => %ecx
    for(i;i<=0x63;i++){
        a += i;
    }
    printf("a = %d\n",a);
    printf("i = %x\n",i);
    return 0;
}
```

```
bash-4.2$ gcc partE-prints.c -o partE
bash-4.2$ ./partE
a = 4950
i = 64
bash-4.2$
```

```
    addi    zero,zero,0
kernel:
    addi    sp,zero,1536
    call    main
    addi    zero,zero,0
    mv      s1,a0
    addi    zero,zero,0
    addi    zero,zero,0
    auipc   ra,0x0
    jalr    ra,0(ra)
    addi    zero,zero,0
    addi    zero,zero,0
.file   "partE.c"
    .option nopic
    .text
    .align  2
    .globl  partE
    .type   partE, @function
partE:
    addi    sp,sp,-48
    sw  ra,44(sp)
    sw  s0,40(sp)
    addi    s0,sp,48
```

Register		Value	Register		Value
zero	[0]	0	ra	[1]	12
sp	[2]	1536	gp	[3]	0
tp	[4]	0	t0	[5]	0
t1	[6]	0	t2	[7]	0
s0/fp	[8]	0	s1	[9]	0
a0	[10]	4950	a1	[11]	0
a2	[12]	0	a3	[13]	0
a4	[14]	4950	a5	[15]	4950
a6	[16]	99	a7	[17]	0
s2	[18]	0	s3	[19]	0
s4	[20]	0	s5	[21]	0
s6	[22]	0	s7	[23]	0
s8	[24]	0	s9	[25]	0
s10	[26]	0	s11	[27]	0

```
             STACK SEGMENT
             FREE SPACE
0x000005fc:  00 00 00 0c
0x000005f8:  00 00 00 00
0x000005f4:  00 00 00 00
0x000005f0:  00 00 00 00
0x000005ec:  00 00 00 00
0x000005e8:  00 00 00 00
0x000005e4:  00 00 13 56
0x000005e0:  00 00 00 00
0x000005dc:  00 00 00 ac
0x000005d8:  00 00 06 00
0x000005d4:  00 00 00 00
0x000005d0:  00 00 00 00
0x000005cc:  00 00 00 00
0x000005c8:  00 00 00 00
0x000005c4:  00 00 00 00
0x000005c0:  00 00 00 00
0x000005bc:  00 00 13 56
0x000005b8:  00 00 00 64
0x000005b4:  00 00 00 00
0x000005b0:  00 00 00 00
0x000005ac:  00 00 00 00
```

Figure 4: C code (with additional print statements) demonstration on x86-64 platform [A], RISC-V demonstration on BRISC-V Simulator [B].

6 PART F

6.1 Dynamic scheduling

The CDC6600 implemented a Scoreboard to enable dynamic scheduling in the datapath. An alternative mechanism, called the Tomasulo Algorithm was implemented by IBM on the System/360 Model 91. Provide detailed descriptions of these two mechanisms and discuss how they resolved different kinds of hazards. Argue for which mechanism you would choose to implement in practice and defend your choice. Identify a design that uses at least one of these mechanisms today.

6.2 Solution

Scoreboard is a centralized hardware mechanism for dynamic scheduling introduced by CDC 6600 [4]. It issues instructions in-order and executes instructions out-of-order [5]. Scoreboard focuses on maintaining a one instruction per clock cycle execution rate by issuing/executing instructions as soon as possible [4]. For full performance, Scoreboard depends on sufficient resources and data independence. Scoreboard consists of four stages - issue, read operands, execution, write result - and begins the next instruction whenever source operands and functional units are available [1]. The typical structure of Scoreboard is in Figure 5, created by Dr. Hong Jiang [5].

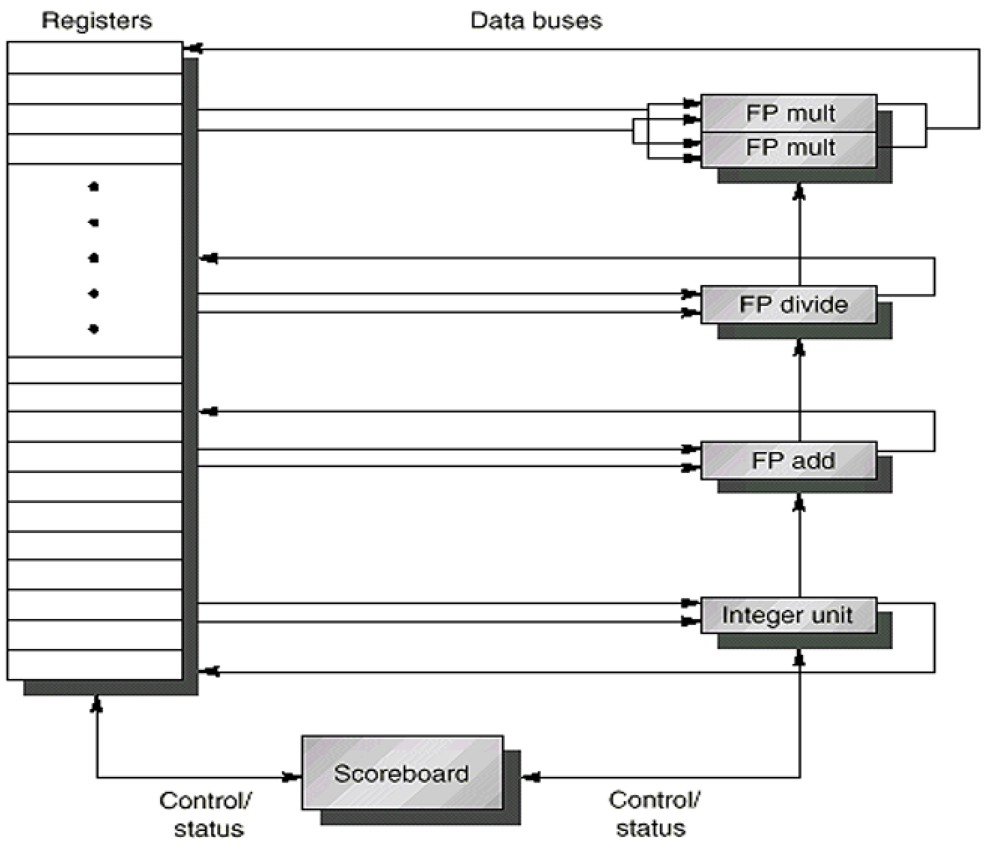


Figure 5: Typical Scoreboard Structure.

Scoreboard handles hazard detection through hardware. First, it utilizes hazard detection hardware to stall pipeline during hazards [5]. Second, it utilizes hardware to create dynamic dependency graphs for rearranging instruction execution order at run-time, shown in Figure 6 provided by Dr. Hong Jiang [5]. At the issue stage, Scoreboard checks and stalls for WAW hazards [1]. At the read operands stage, Scoreboard checks for available source operands and resolves RAW hazards by rearranging instructions [1]. At the write result stage, Scoreboard checks and stalls for WAR hazards [1].

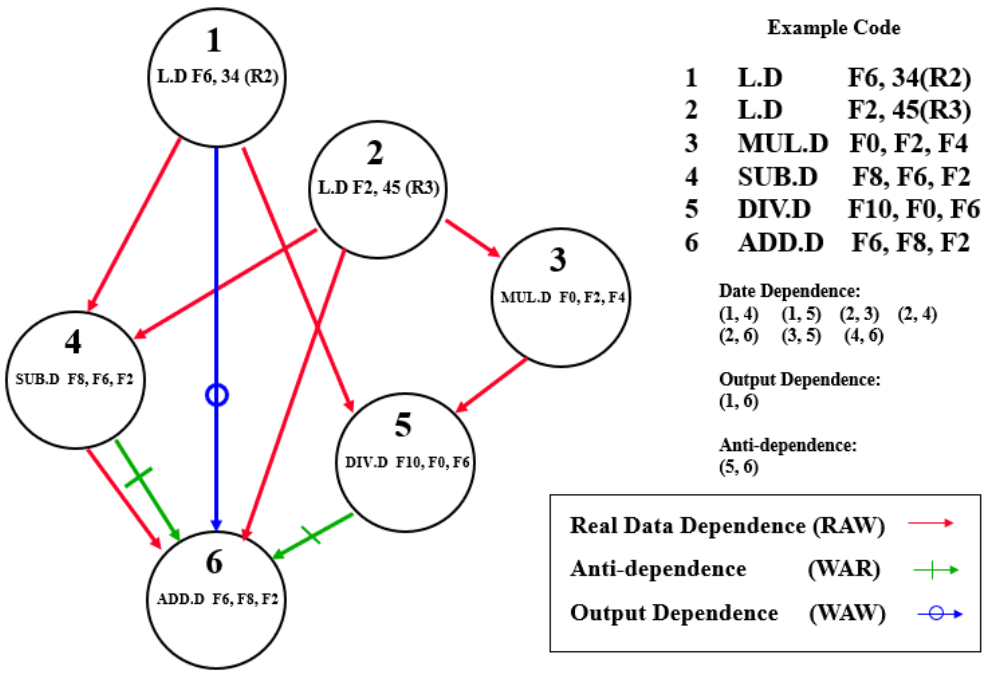


Figure 6: Scoreboard Dependency Graph.

Tomasulo's algorithm is a more advanced method for dynamic scheduling than Scoreboarding. Tomasulo's algorithm dynamically renames registers and handles speculation [7]. Rather than stalling for WAW or WAR hazards, renaming registers allows Tomasulo's algorithm to eliminate these output and anti-dependencies. It renames registers with reservation stations (buffers that fetch and store available instruction operands) [1]. Results may be buffered after previous instruction execution, allowing the next instruction to execute. The reservation station then sends results directly to the next reservation station or stores it in the register file, eliminating WAR or WAW stalls [1]. Tomasulo's organization is shown in Figure 7, provided by Dr. Dean Tullsen [3].

I would choose to implement Tomasulo's Algorithm rather than Scoreboarding. First, Scoreboard is limited directly by hardware. Scoreboard requires open functional units to execute instructions out-of-order; whereas Tomasulo allows speculative branch prediction, which removes limitations from basic functional units. Second, Scoreboard resolves RAW hazards but stalls to prevent WAW and WAR hazards. Tomasulo outperforms Scoreboard by avoiding WAR and WAW hazards with register renaming, lowering amount of stalls. Scoreboard may be adequate for simple processors; however, Tomasulo prevents all hazards, required for high performance on processors. In processors today, Intel's x86-64 chips utilize reservation stations for register renaming and advanced branch prediction, showing a modern design for Tomasulo's algorithm [9].

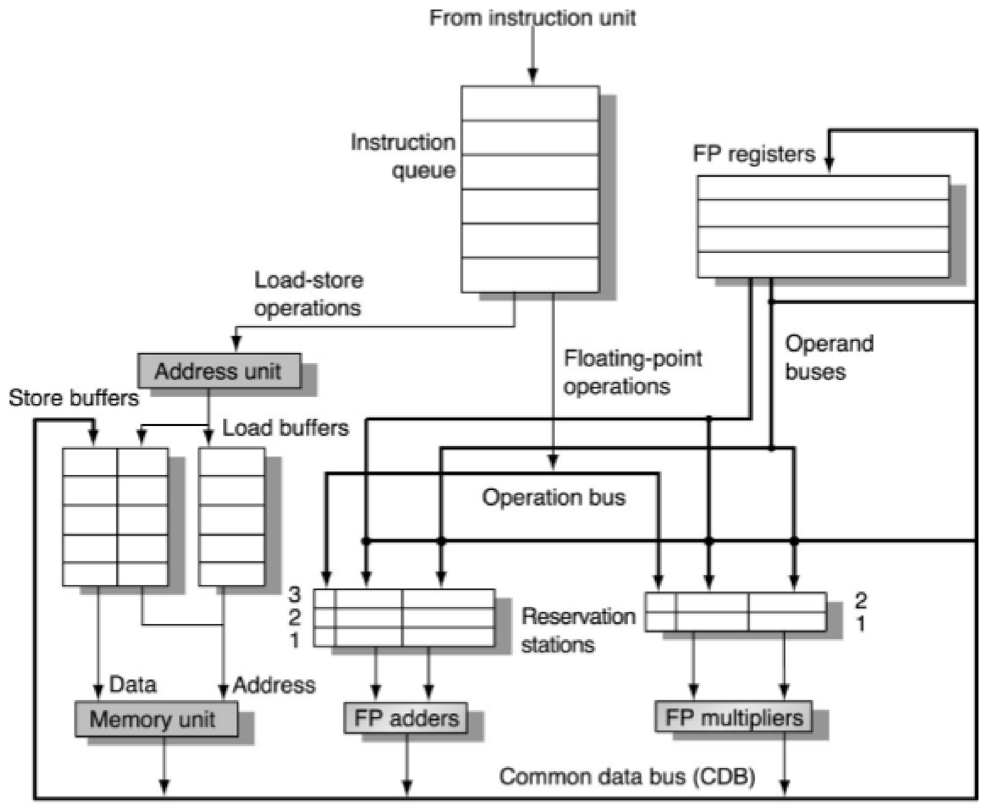


Figure 7: Tomasulo Organization.

7 EXTRA CREDIT (25 POINTS)

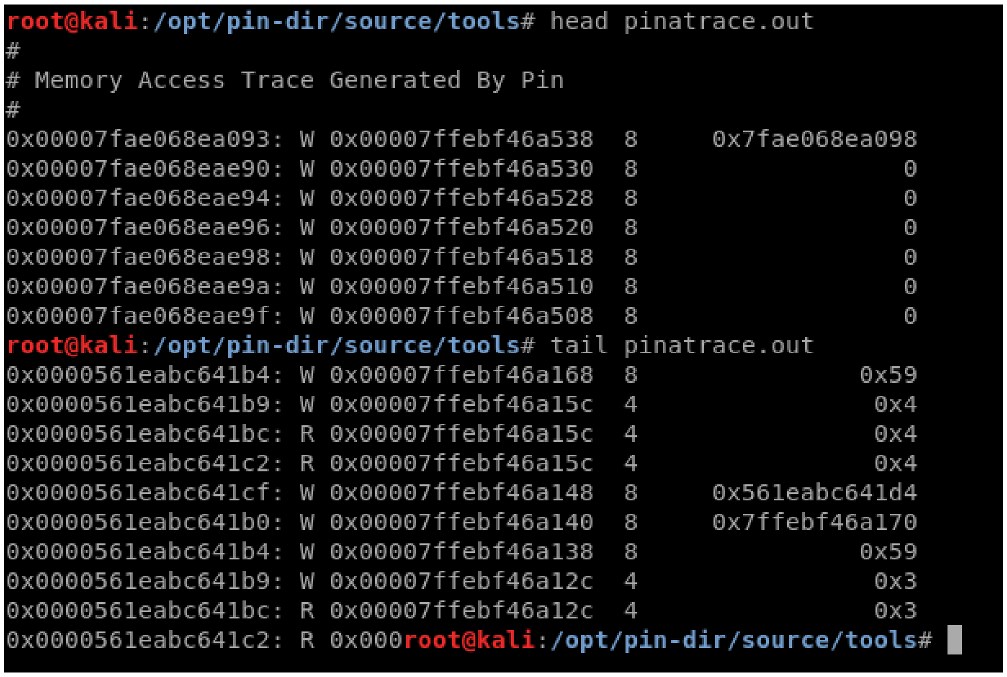
7.1 Data address trace

Find a tool to generate a data address trace for the fibo.c program run on X86. Only collect data addresses execution. Identify how many unique pages of memory (assume 4KB pages) are touched by data references in your code. How are you going to get this done? You could modify a simulator that can save the trace, or you can find a customized tool that provides you with the ability to generate the trace and records addresses for loads and stores. This is your choice. Provide details on what tool you used.

7.2 Solution

I utilized Intel's pin tool to collect data addresses execution. Specifically, I used the pinatrace instrumentation tool with pin; the command was ../../pin -t /opt/pin-dir/source/tools/SimpleExamples/obj-intel64/pinatrace.so – fibo. This tool provides a memory reference trace through the program. The output of the trace gives all memory addresses referenced by a program, the columns in the result are as follows: instruction pointer address, read/write, memory addressed to be read/written, and size of memory. An example is provided in Figure 8 which shows the head and tail of the output file.

The number of unique pages of memory is 674,741 pages. This was calculated by the following equation: (memory address - last memory address) / page size = (0x00007fae068ea093 - 0x0000561eabc641c2) / (4K * 1024) = 0x298f5ac85ed1 / (4K *1024) = 0xa4bb5 or 674,741 pages.



```
root@kali:/opt/pin-dir/source/tools# head pinatrace.out
#
# Memory Access Trace Generated By Pin
#
0x00007fae068ea093: W 0x00007ffebf46a538  8     0x7fae068ea098
0x00007fae068eae90: W 0x00007ffebf46a530  8                  0
0x00007fae068eae94: W 0x00007ffebf46a528  8                  0
0x00007fae068eae96: W 0x00007ffebf46a520  8                  0
0x00007fae068eae98: W 0x00007ffebf46a518  8                  0
0x00007fae068eae9a: W 0x00007ffebf46a510  8                  0
0x00007fae068eae9f: W 0x00007ffebf46a508  8                  0
root@kali:/opt/pin-dir/source/tools# tail pinatrace.out
0x0000561eabc641b4: W 0x00007ffebf46a168  8               0x59
0x0000561eabc641b9: W 0x00007ffebf46a15c  4                0x4
0x0000561eabc641bc: R 0x00007ffebf46a15c  4                0x4
0x0000561eabc641c2: R 0x00007ffebf46a15c  4                0x4
0x0000561eabc641cf: W 0x00007ffebf46a148  8     0x561eabc641d4
0x0000561eabc641b0: W 0x00007ffebf46a140  8     0x7ffebf46a170
0x0000561eabc641b4: W 0x00007ffebf46a138  8               0x59
0x0000561eabc641b9: W 0x00007ffebf46a12c  4                0x3
0x0000561eabc641bc: R 0x00007ffebf46a12c  4                0x3
0x0000561eabc641c2: R 0x000root@kali:/opt/pin-dir/source/tools#
```

Figure 8: Address trace results.

REFERENCES

[1] Tong A. Dynamic scheduling. URL https://www.cs.umd.edu/~meesh/cmsc411/website/projects/dynamic/scoreboard.html. [Online; accessed 16-October-2019].

[2] ARM. Arm7tdmi (rev 3) core processor product overview. URL infocenter.arm.com/help/topic/com.arm.doc.dvi0027b/DVI_0027A_ARM7TDMI_PO.pdf. [Online; accessed 12-October-2019].

[3] Tullsen D. Instruction level parallelism. URL http://cseweb.ucsd.edu/classes/wi05/cse240a/ilp2.pdf. [Online; accessed 16-October-2019].

[4] Prabhu G. Dynamic scheduling techniques. URL http://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/dynamSchedTech.html. [Online; accessed 16-October-2019].

[5] Jiang H. Instruction-level parallelism: Scoreboard. URL https://www.cs.umd.edu/website/projects/dynamic/scoreboard. [Online; accessed 16-October-2019].

[6] A. Hartstein and T. R. Puzak. *The Optimum Pipeline Depth for a Microprocessor*. 2002.

[7] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[8] Texas Instruments. Msp430f15x, msp430f16x, msp430f161x mixed signal microcontroller. URL https://www.ti.com/lit/ds/symlink/msp430f1611.pdf. [Online; accessed 12-October-2019].

[9] Intel. Intel 64 and ia-32 architectures software developer's manual volume 1 basic architecture. URL https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf#zoom=100. [Online; accessed 16-October-2019].

[10] N. P. Jouppi and D. W. Wall. *Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines*. Western Research Laboratory, 1989.