ECE 2300 Digital Logic and Computer Organization

Topic 10: Single-Cycle Processors

`http://www.csl.cornell.edu/courses/ece2300`
School of Electrical and Computer Engineering
Cornell University

revision: 2025-12-09-12-46

List of Problems

Problem 1. Performance Evaluation

In this problem, we will estimate the execution time of multiple programs when running on our single-cycle processor from lecture.

We will be using the following equation from lecture to estimate the performance of each program:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Avg Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}} \tag{1}$$

In this practice problem, we will determine each of these product terms individually.

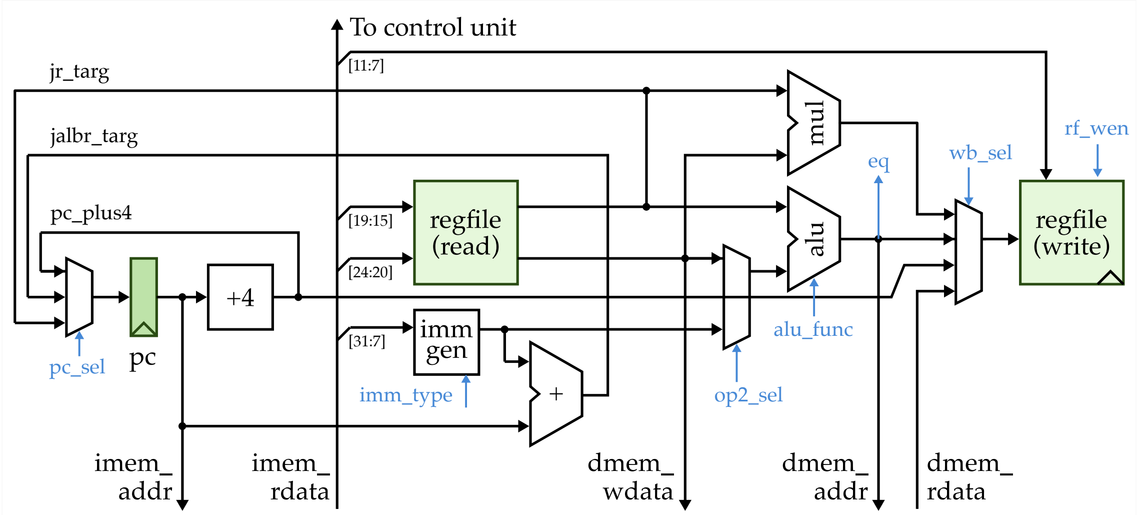
First, we will determine the number of cycles needed per instruction. This metric depends on the ISA and microarchitecture of the processor that executes the code. Next, we will determine the time needed for each cycle (clock period), which depends on the microarchitecture and implementation of the processor core. Lastly, the number of instructions depends on the specific program to execute (and the compiler that converts it from higher-level programming languages to assembly). We will determine it for each program individually in later steps.

Part 1.A Average Cycles per Instruction

How many cycles are needed per instruction for our single-cycle processor?

Part 1.B Estimate Clock Period

First, we need to figure out the clock period of our single-cycle processor. **Determine the critical path (remember to ignore false paths). Highlight this path in the datapath diagram and describe the instruction which triggers this path. Compute the minimum clock T_C period that would still ensure correct operation of the single-cycle processor in units of τ.**



	t_{pd}
32-bit 2-to-1 Mux	4τ
32-bit 4-to-1 Mux	8τ
32-bit Adder	60τ
32-bit ALU	64τ
32-bit Multiplier	100τ
32-bit +4 Unit	30τ
ImmGen Unit	12τ
32-bit Reg (t_{cq})	9τ
Register File Read	25τ
Memory Read	120τ

32-bit Reg (t_{setup})	10τ
Register File (t_{setup})	20τ
Memory (t_{setup})	120τ

Part 1.C Estimate Number of Instructions per Program

Next, we will determine the number of instructions executed (dynamic instruction count) for a number of programs.

1.C.1 Program 1: Pythagorean Theorem

Program 1 computes the length of the hypotenuse of a right triangle (using integers, not floating-point numbers) via the Pythagorean theorem (see equation below). An IO-mapped accelerator for computing the square root function is connected to the single-cycle processor. The accelerator reads from `out0` (address 528), requires one cycle to compute (during which the processor must busy-wait), and then writes the integer square root (rounded down) to `in0` (address 512). **Execute the TinyRV1 assembly code using the provided sheets.**

$$C = \sqrt{A^2 + B^2}$$

```
□□□ □□□ 00 sw   x0, 528(x0)
□□□ □□□ 04 lw   x5, 256(x0)
□□□ □□□ 08 lw   x6, 260(x0)
□□□ □□□ 12 mul  x5, x5, x5
□□□ □□□ 16 mul  x6, x6, x6
□□□ □□□ 20 add  x5, x5, x6
□□□ □□□ 24 sw   x5, 528(x0)
□□□ □□□ 28 addi x0, x0, 0   # wait for accelerator
□□□ □□□ 32 lw   x5, 512(x0)
□□□ □□□ 36 sw   x5, 256(x0)
```

External I/O

528	
...	
512	

Memory

508	
...	
264	
260	7
256	5
...	
40	
36	
32	
28	
24	
20	
16	
12	
8	
4	
0	

Registers

x31	
x30	
x29	
x28	
...	
x11	
x10	
...	
x7	
x6	
x5	
...	
x0	

From now on, we will mainly use transaction diagrams for analyzing the code execution. **Complete the transaction diagram below.**

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
`sw   x0, 528(x0)`																					
`lw   x5, 256(x0)`																					
`lw   x6, 260(x0)`																					
`mul  x5, x5, x5`																					
`mul  x6, x6, x6`																					
`add  x5, x5, x6`																					
`sw   x5, 528(x0)`																					
`addi x0, x0, 0`																					
`lw   x5, 512(x0)`																					
`sw   x5, 256(x0)`																					

Compute the number of dynamic instructions when executing this program. *Note: Include the nop/addi instruction while the processor is busy-waiting.*

Compute the run time of program 1 using equation 1 in units of τ.

1.C.2 Program 2: Factorial Function

Next, we will compute the factorials (see equation below) for the input stored in `in0` at address 512. When complete, this program will store the result in `out0` at address 528. **Inspect and understand the TinyRV1 code below.**

$$n! = \prod_{k=1}^{n} k = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

```
 addi x5, x0, 1
 lw   x6, 512(x0)   # read in0
 addi x6, x6, 1     # stop = in0+1
 addi x7, x0, 1     # fact = 1
 bne  x6, x5, loop
 jal  x0, end       # catch 0!
loop:
 mul  x7, x5, x7    # fact = fact*i
 addi x5, x5, 1
 bne  x5, x6, loop
end:
 sw   x7, 528(x0)   # out0 = fact
```

Complete the transaction diagram for in0 being set to 0.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Compute the number of dynamic instructions when computing the factorial of 0.

Compute the run time in units of τ when computing the factorial of 0 using equation 1.

Complete the transaction diagram for in0 to be set to 4.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Compute the number of dynamic instructions when computing the factorial of 4.

Compute the run time in units of τ when computing the factorial of 4 using equation 1.

1.C.3 Program 3: Reverse Fibonacci

The following program computes the element that came just before (F_{n-2}) from the current two Fibonacci sequence elements (F_n and F_{n-1}), as shown in the equation below. The two current Fibonacci sequence elements are provided as function arguments and are thus stored in x10 and x11 according to the TinyRV1 ISA. F_{n-2} will be returned as a return value and will thus also be stored in x10. We return an error code (-1) when F_{n-1} is 0, since no element before 0 exists in the sequence.

$$F_{n-2} = F_n - F_{n-1}$$

```
 addi x5,  x10, 0     # x5=Fn
 addi x6,  x11, 0     # x6=Fn-1
 addi x10, x0,  -1    # return value
 bne  x6,  x0,  comp  # check Fn-1!=0
 jal  x0,  end        # return error
comp:
 mul  x6,  x6,  x10   # x6=-(Fn-1)
 add  x10, x5,  x6    # x10=Fn-(Fn-1)
end:
 addi x0,  x0,  0     # nop
```

Complete the transaction diagram for $F_n = 89$ and $F_{n-1} = 55$.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Compute the number of dynamic instructions for $F_n = 89$ and $F_{n-1} = 55$.

Compute the run time in units of τ for $F_n = 89$ and $F_{n-1} = 55$ using equation 1.

1.C.4 Program 4: Determine Line Count

Usually, text strings (e.g., “hello, world!”) are encoded with the ASCII encoding scheme. ASCII defines a number for each character. For instance, “h” is represented by the decimal value 104 and “e” by 101, resulting in an integer array in which each element corresponds to a character.

In this program, we will determine the number of lines required to print a text string. To do so, we will count the number of newline characters (“\n”) within our string. ASCII encodes the newline character as the decimal value 10. Our text string starts in memory at address 256 and has a length of 8 elements.

Note: Usually, ASCII characters are stored in 8-bit memory units. However, for this assignment, we will assume that they are stored in 32-bit words. Furthermore, we will assume that our text string ends with a newline character, so the number of newline characters corresponds to the number of lines required.

Inspect and understand the TinyRV1 code below.

```
  addi x5,  x0, 256   # ptr addr
  addi x6,  x0, 288   # 288 = 256+8*4
  addi x7,  x0, 0     # newline count
  addi x10, x0, 10    # newline character
loop:
  lw   x11, 0(x5)
  bne  x11, x10, jump # check if newline char
  addi x7,  x7, 1     # increment newline count
jump:
  addi x5,  x5, 4
  bne  x5,  x6, loop
  addi x10, x7, 0     # done
```

Complete the transaction diagram; assume the newline characters are at position 4 and 8 (for an index starting at 1). *Note: Do not forget to add the cycle number above each column.*

Transaction diagram continued on next page...

Compute the number of dynamic instructions.

Compute the run time in units of τ using equation 1.

Create a function for computing the run time in units of τ for sequence length L containing P percent newline characters.

1.C.5 Program 5: Array Reversal

In this program, we reverse an array of 32-bit integers. We receive as function arguments the pointer to its front in register x10 and the pointer to its back in register x11. In each loop iteration, we swap the values stored at the addresses to which the front and back pointers point. Afterwards, we increment the front pointer and check if it equals the back pointer. If so, we are done, as we have reached the middle of an array with an even number of elements. Otherwise, we decrement the back pointer and again check if the pointers point to the same address. If they do, we are done, as we have reached the middle element of an array with an odd number of elements. When we are done, we return zero in register x10 (according to the TinyRV1 ISA) to indicate successful completion of the function.

Note: Assume the array to have at least two elements.

Inspect and understand the TinyRV1 code below.

```
 addi x5,  x10, 0      # ptr front
 addi x6,  x11, 0      # ptr back
loop:
 lw   x10, 0(x5)       # swap values
 lw   x11, 0(x6)
 sw   x11, 0(x5)
 sw   x10, 0(x6)
 addi x5,  x5, 4       # incr ptr front
 bne  x5,  x6, check2  # check if ptrs equal
 jal  x0,  done
check2:
 addi x6,  x6, -4      # decr ptr back
 bne  x5,  x6, loop
done:
 addi x10, x0, 0       # return success
```

Complete the transaction diagram; assume the front pointer to be 256 and the back to be 268.

Compute the number of dynamic instructions.

Compute the run time in units of τ using equation 1.

1.C.6 Program 6: Mask Array

The following program masks all non-zero elements within an array. It iterates over each element (in the `mask_array` segment), checks if its value is non-zero, and if so, overwrites the respective element with the hexadecimal value `0xFF` (using the `mask_elmnt` segment).

This program is slightly more realistic than previous examples, as it includes two function calls (`mask_array` and `mask_elmnt`) and utilizes the stack to store the return address register when making the nested function call to `mask_elmnt`. The function arguments (array pointer and array length) are passed in registers `x10` and `x11` according to the TinyRV1 ISA. The return address is stored in register `x1` and the stack pointer in `x2`.

Note: To limit instruction count, we only save the minimum required registers to the stack. A fully compliant implementation would save all needed caller-saved registers (e.g., `x5`, `x6`, `x31`) before calling `mask_elmnt`.

Inspect and understand the TinyRV1 code below.

```
 jal  x0,  main
# ...
mask_elmnt:
 sw   x31, 0(x10)      # overwrite memory to mask
 jr   x1               # return to hide
mask_array:
 addi x5,  x10, 0      # ptr front
 addi x6,  x0,  4
 mul  x6,  x6,  x11
 addi x6,  x5,  x6     # ptr end = base + offset
loop:
 lw   x7,  0(x5)       # load element
 bne  x7,  x0, hide
 jal  x0,  incr_ptr
hide:
 addi x2,  x2, -4      # allocate space on stack
 sw   x1,  0(x2)       # put return address on stack
 addi x10, x5, 0       # set func argument: mem addr to mask
 jal  x1,  mask_elmnt
 lw   x1,  0(x2)       # restore return address
 addi x2,  x2, 4       # deallocate space on stack
incr_ptr:
 addi x5,  x5, 4       # increment array ptr
 bne  x5,  x6, loop
 jr   x1               # done; return to main
# ...
main:
 addi x2,  x0, 512     # set stack ptr
 addi x31, x0, 0xFF    # MASK value
 addi x10, x0, 256     # func arg: array ptr
 addi x11, x0, 3       # func arg: array len
 jal  x1,  mask_array
 addi x10, x0, 0       # return success
```

Complete the transaction diagram; assume only the first element in the three element array to be non-zero.

Transaction diagram continued on next page...

Compute the number of dynamic instructions.

Compute the run time in units of τ using equation 1.

1.C.7 Program 7: Pin Check with Timing Vulnerability

The following program performs a simple pin check. The *correct* pin is stored in the data segment of memory starting at address 256. The user input is obtained through memory-mapped I/O starting at address 512. The pin consists of four 32-bit integers.

```
 jal  x0,  main
# ...
chck_loop:
 lw   x5,  0(x10)            # load input pin char
 lw   x6,  0(x11)            # load correct pin char
 bne  x5,  x6,  ret_mismatch  # check for pin char match
 addi x10, x10, 4
 addi x11, x11, 4
 bne  x10, x12, chck_loop     # check if all chars checked
 addi x10, x0,  0             # pin match: return 0
 jr   x1
ret_mismatch:
 addi x10, x0,  1             # pin mismatch: return 1
 jr   x1
# ...
main:
 addi x10, x0,  512           # ptr input pin
 addi x11, x0,  256           # ptr correct pin
 addi x12, x10, 16            # done ptr
 jal  x1,  chck_loop          # check pin
 addi x0,  x0, 0
```

Complete the transaction diagram. Assume the correct pin to be 7, 4, 9, 1. However, the user input is 7, 1, 2, 3.

How many cycles are needed for executing this program?

Create a formula for determining the run time of this program when the first N characters are correct, where N ranges from 0 to 4.

Timing Vulnerability

As you have realized, the run time of this pin check depends on the number of correct elements. Such timing side-channel vulnerabilities are among the most fundamental security issues when implementing secure applications. Attackers can try different inputs and measure the program's run time. If the run time increases, they know they have guessed another element of the PIN correctly. They can then proceed to guess the next element until they have compromised the entire PIN.

When implementing secure systems, you must ensure that you do not leak additional information to potential attackers through side-channels. Timing is just one of many possible side-channels. For instance, different instructions may consume different amounts of power. Therefore, when average power consumption changes, attackers might deduce that the program has started executing dummy instructions that are no longer actually checking the PIN.

Niklas's Solution

`https://vod.video.cornell.edu/id/1_2o8r6rd6`