ECE 2300 Digital Logic and Computer Organization
Topic 11: Multi-Cycle Processors

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

revision: 2025-12-11-02-16

List of Problems

Problem 1. Performance Evaluation

In this problem, we will estimate the execution time of multiple programs when running on our 63-state multi-cycle processor from lecture.

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

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

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

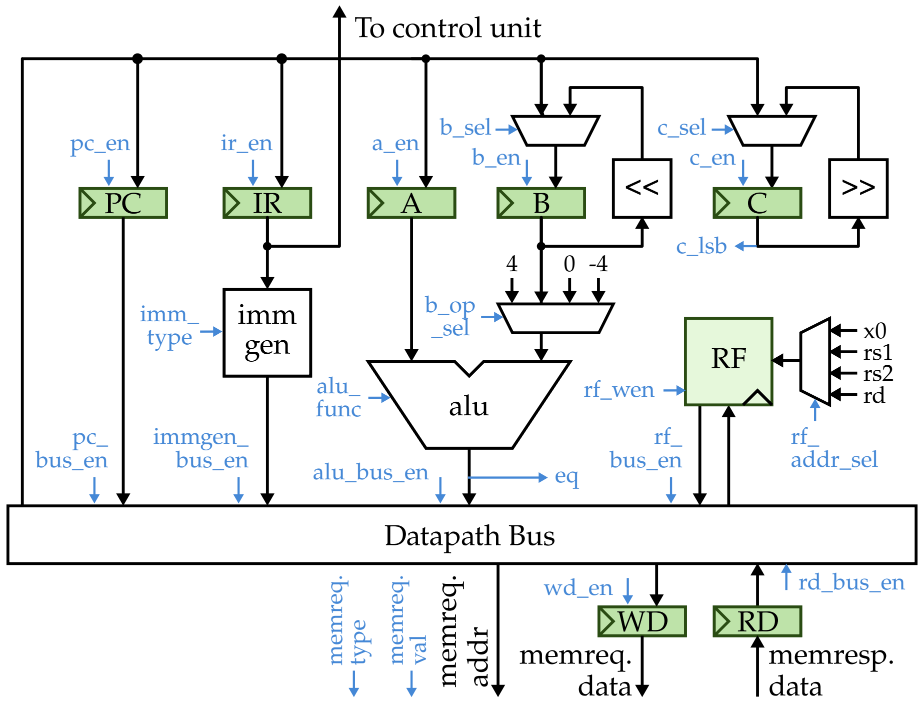
First, we will determine the number of cycles needed per instruction. This metric generally depends on the ISA and microarchitecture of the processor that executes the code. For the multi-cycle processor, each instruction requires a different number of cycles. Thus, depending on the distribution of instructions used in the application, the average number of cycles per instruction changes. Therefore, we will first determine the number of cycles needed for each of our eight instructions (later, when inspecting each application in detail, we will determine the average number of cycles needed per instruction for each program). 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 Cycles per Instruction

How many cycles are needed per instruction for each of our eight instructions when executed on our 63-state multi-cycle processor? *Note: Remember that the number of cycles for the "bne" instruction depends on whether the branch is taken or not.*

Part 1.B Estimate Clock Period

First, we need to figure out the clock period of our 63-state multi-cycle processor. **Determine the critical path (remember to ignore false paths). Highlight this path in the datapath diagram and describe the state which triggers this path. Compute the minimum clock period T_C that would still ensure correct operation of the 63-state multi-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
Datapath Bus	20τ
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) and the number of cycles needed for multiple 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 multi-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).

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

```
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    # wait sqrt comp
lw   x5, 512(x0)
sw   x5, 256(x0)
```

Complete the transaction diagram below for the multi-cycle processor. Insert instructions in the left column and add the respective cycle numbers above each column. *Note: For the multiply operation, you may abbreviate the intermediate `MXX` states in your diagram using '...' notation rather than writing out all states. Show enough states to make the pattern clear (e.g., the first few states, '...', and the final state). Remember to write out all states in the exam unless otherwise stated.*

Determine the number of dynamic instructions executed and the number of cycles needed for executing them. Compute with this data the average number of cycles needed per instruction. *Note: Include the nop/addi instruction, which takes multiple cycles, 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.

Compute the number of dynamic instructions, the number of cycles, and average number of cycles needed per instruction 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 2. *Note: For the multiply operation, you may abbreviate the intermediate `MXX` states in your diagram using '...' notation rather than writing out all states. Show enough states to make the pattern clear (e.g., the first few states, '...', and the final state). Remember to write out all states in the exam unless otherwise stated.*

Compute the number of dynamic instructions, the number of cycles, and average number of cycles needed per instruction when computing the factorial of 2.

Compute the run time in units of τ when computing the factorial of 2 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 $F_n = 1$ and $F_{n-1} = 0$.

Compute the number of dynamic instructions, the number of cycles, and average number of cycles needed for $F_n = 1$ and $F_{n-1} = 0$.

Compute the run time in units of τ for $F_n = 1$ and $F_{n-1} = 0$ using equation 1.

Compute the number of dynamic instructions, the number of cycles, and average number of cycles needed 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 3 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, 268   # 268 = 256+3*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 a newline character is at position 3 (for an index starting at 1).

Compute the number of dynamic instructions, the number of cycles, and average number of cycles.

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 264.

Compute the number of dynamic instructions, the number of cycles, and average number of cycles.

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, 2       # func arg: array len
 jal  x1,  mask_array
 addi x10, x0, 0       # return success
```

Complete the transaction diagram; assume only the last element in the two element array to be non-zero. *Note: For the multiply operation, you may abbreviate the intermediate `MXX` states in your diagram using '...' notation rather than writing out all states. Show enough states to make the pattern clear (e.g., the first few states, '...', and the final state). Remember to write out all states in the exam unless otherwise stated.*

Compute the number of dynamic instructions, the number of cycles, and average number of cycles.

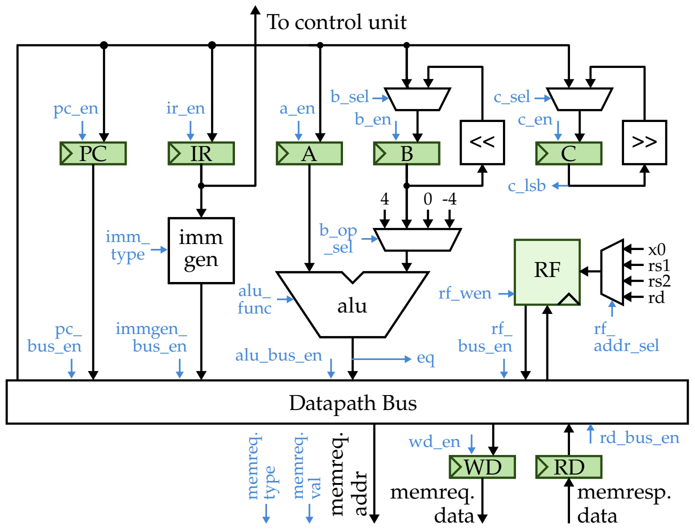
Compute the run time in units of τ using equation 1.

Problem 2. 65-State Multi-Cycle Processor

The minimum clock period of our 63-state multi-cycle processor is defined by the memory instructions `lw` and `sw` (see Problem 1.B). More specifically, the states `L2` and `S3` perform both the address computation (offset added to the value stored in a register) and memory accesses, leading to a situation in which two modules (ALU, memory) with the highest propagation delays in our datapath are within the same (real) path. This results in a minimum clock period that is much higher than most states within the 63-state multi-cycle processor require.

A potential optimization is to split the `L2` and `S3` states into two states each (`L2A`, `L2B`, `S3A`, and `S3B`, respectively). In the `A` states, we perform the address computation and store the result in a new temporary register `TMP` (analogous to the `PC` register). In the subsequent `B` states, we perform the actual memory request. This renders the path from the `B` register through the ALU into memory a false path, reducing the overall minimum clock period. However, `lw` and `sw` instructions will each require an additional cycle in the new **65-state multi-cycle processor**.

Part 2.A Design



	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
Datapath Bus	20τ
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

Add the `TMP` register to the block diagram of the multi-cycle processor above.

Furthermore, determine the new critical path. Highlight the path in the datapath diagram and describe the state or states that trigger this path. Compute the minimum clock period T_C that would still ensure correct operation of the 65-state multi-cycle processor in units of τ.

Next, draw the resulting FSM diagram of the 65-state multi-cycle processor. *Feel free to omit most of the multiply states, as their repetitive structure is understood.*

Furthermore, write the micro-ops for the newly created states `L2A`**,** `L2B`**,** `S3A`**,** `S3B`**.**

Part 2.B Performance Measurement

We are going to evaluate the performance using the array reversal program from before. However, in this evaluation we will reverse a much longer array containing 513 elements. Thus, we will only execute a single iteration of the loop in a transaction diagram and extrapolate the total runtime from this.

```
 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
```

2.B.1 63-State Multi-Cycle Processor

First, we will execute the array reversal on the **63-state multi-cycle processor** for reference. **Complete the transaction diagram below for the 63-state multi-cycle processor.**

How many cycles are required per loop when executing on the 63-state multi-cycle processor? What is the average number of cycles per instruction?

Determine the run time for reversing the array with 513 elements when executing on the 63-state multi-cycle processor.

2.B.2 65-State Multi-Cycle Processor

Next, we will execute the array reversal on the **65-state multi-cycle processor**. **Complete the transaction diagram below for the 65-state multi-cycle processor.**

How many cycles are required per loop when executing on the 65-state multi-cycle processor? What is the average number of cycles per instruction?

Determine the run time for reversing the array with 513 elements when executing on the 65-state multi-cycle processor.

Part 2.C Comparative Analysis

Perform a quantitative and qualitative comparative analysis between the 63-state and 65-state multi-cycle processors. Does the potential performance gain justify the increase in area? Can you generalize the potential performance gain to other algorithms with different instruction compositions and could we see a performance degradation for some applications? What is the maximum possible performance improvement?