ECE 2300 Digital Logic and Computer Organization, Fall 2025

Discussion Section 9 – Single-Cycle Processors

revision: 2025-11-07-12-34

In this discussion section, you will estimate the time to execute a vector-vector-add program on the TinyRV1 single-cycle processor we studied in lecture. The vector-vector-add program takes as input two arrays of 4B numbers, adds corresponding array elements, and outputs a third array with the results. Here is the pseudo-code for our vector-vector-add program.

```
for i in range(n):
  dest[i] = src0[i] + src1[i]
```

We will be using the following first-order equation to estimate performance:

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

So our quantitative performance analysis will involve filling in the following table.

	Inst/Prog	Cycles/Inst	Time/Cycle	Time/Prog
vvadd on TinyRV1 Scycle Processor				

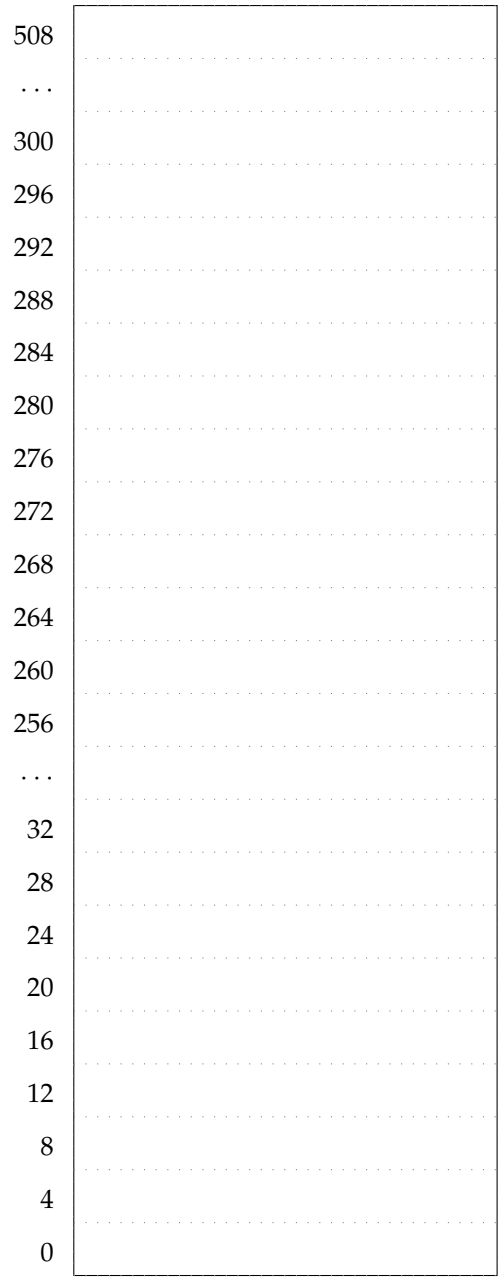
For a single-cycle processor the average cycles per instruction is just one (hopefully no surprises there!). We will be using both a paper worksheet and the instruction-set simulator from the previous discussion section to estimate the number of instructions per program. We will use paper analysis to estimate the time per cycle.

Problem 1. Estimating Instructions per Program with Worksheet

Use the worksheet on the following page to estimate the number of instructions for the assembly program assuming `n` is four (i.e., the length of the input and output arrays is four elements). The number of instructions executed is just the total number of Xs next the assembly code.

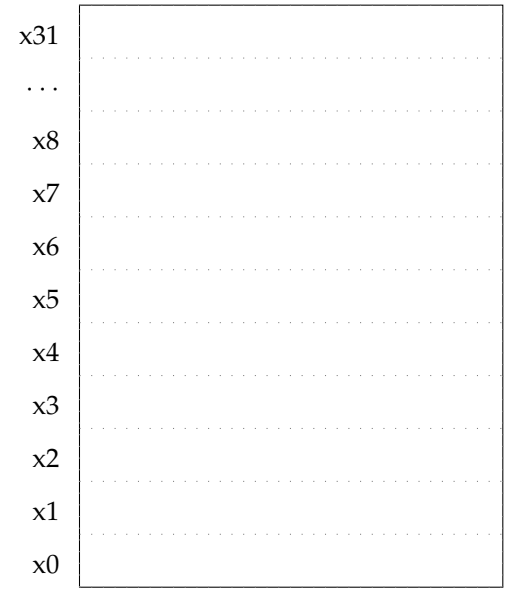
```
□□□ □□□ 00 addi x1, x0, 256   # x1 holds base address of src0
□□□ □□□ 04 addi x2, x0, 272   # x2 holds base address of src1
□□□ □□□ 08 addi x3, x0, 288   # x3 holds base address of dest
□□□ □□□ 12 addi x4, x0, 4     # x4 holds size of arrays
□□□ □□□ 16 lw   x5, 0(x1)     # x5 = src0[i] <--------.
□□□ □□□ 20 lw   x6, 0(x2)     # x6 = src1[i]          |
□□□ □□□ 24 add  x7, x5, x6    # x7 = x5 + x6          |
□□□ □□□ 28 sw   x7, 0(x3)     # dest[i] = x7          |
□□□ □□□ 32 addi x1, x1, 4     # next element of src0  |
□□□ □□□ 36 addi x2, x2, 4     # next element of src1  |
□□□ □□□ 40 addi x3, x3, 4     # next element of dest  |
□□□ □□□ 44 addi x4, x4, -1    # x4 = x4 - 1           |
□□□ □□□ 48 bne  x4, x0, 16    # goto 16 if x4 != 0 ---'
□□□ □□□ 52 addi x0, x0, 0     # nop
□□□ □□□ 56 addi x0, x0, 0     # nop
```

Memory



Program Counter

Registers



Problem 2. Estimating Instructions per Program with ISA Simulator

Let's try executing the same program with the ISA simulator we used in the previous discussion section. Log into `ecelinux` using VS Code and use the following commands to clone the repo, build the ISA simulator, and look at the provided assembly code for our vector-vector-add program

```
% source setup-ece2300.sh
% mkdir -p $HOME/ece2300
% cd $HOME/ece2300
% git clone git@github.com:cornell-ece2300/ece2300-sec09-pbl-scycle-proc sec09
% mkdir sec09/build
% cd sec09/build
% ../configure
% make proc-isa-sim
% code ../lab4/asm/vvadd.asm
```

Notice how we are using a label to specify the branch target and we are initializing the input arrays using the `.data` and `.word` assembler directives. Let's now assemble this assembly program into a machine program.

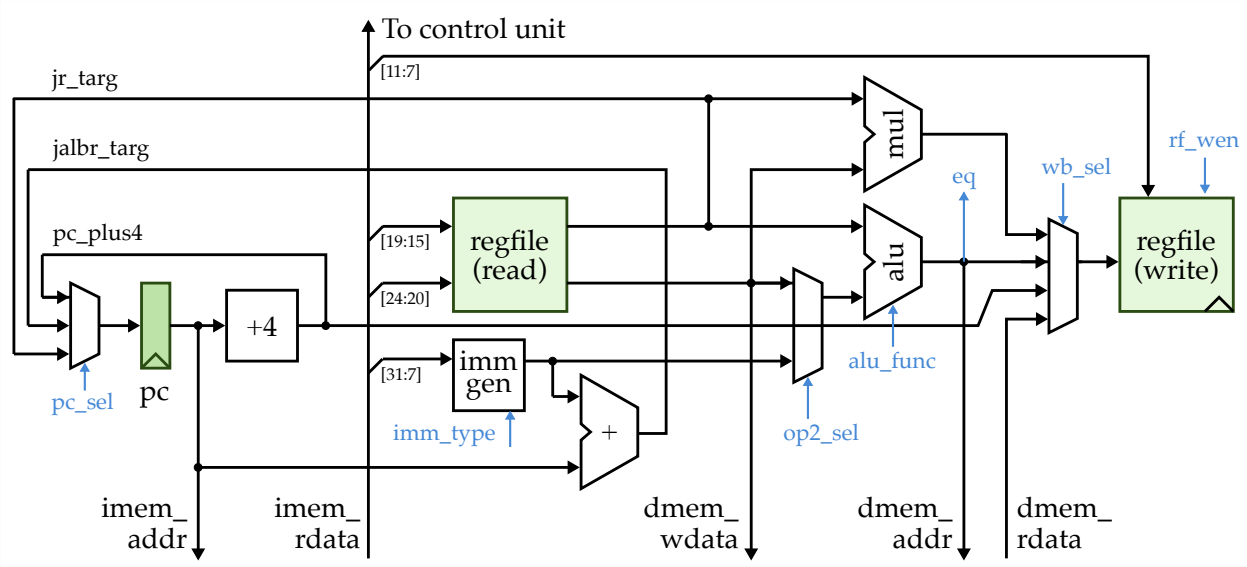
```
% cd $HOME/ece2300/sec09/build
% ../scripts/tinyrv1-assemble -o vvadd.bin ../lab4/asm/vvadd.asm
% cat vvadd.bin
```

Finally, let's run the machine program (i.e., the TinyRV1 binary) on our ISA simulator and visualize its execution.

```
% cd $HOME/ece2300/sec09/build
% ./proc-isa-sim +bin=vvadd.bin +tui
```

Step through the execution and confirm the total number of instructions executed in the program is exactly the same as what you calculated using your worksheet.

Problem 3. Estimating minimum T_C (i.e., clock period, cycle time, or time/cycle)



	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