HW #4: 2.4, 2.12, 2.20, 2.26 — Matt Krueger

2.4 [E] Registers R4 and R5 contain the decimal numbers 2000 and 3000 before each of the following addressing modes is used to access a memory operand. What is the effective address (EA) in each case? R4 = 2000 R5 : 3000

(*a*) 12(R4) EA = R4 + 12 = 2000 + 12 = 2012

(*b*) (R4,R5) EA = R4 + R5 = 2000 + 3000 = 5000

(*c*) 28(R4,R5) EA = R4 + R5 + 28 = 5028

(*d*) (R4)+ EA = R4 + 4 = 2004

(*e*) −(R4) EA = R4 − 4 = 1996

2.12 [E] Both of the following statement segments cause the value 300 to be stored in location 1000, but at different times.

```
ORIGIN      1000
DATAWORD    300
```

Assembly directives; initialize location 1000 with 300. @ compile time (also assembly time)

and

```
Move   R2, #1000
Move   R3, #300
Store  R3, (R2)
```

load R2 with val @ 1000 — already contains 300

load R3 with val @ 300

Point memory of R2 @ R3

Explain the difference.

Difference:

- first segment uses directives to initialize memory at assembly-time
- second segment dynamically performs the storage at run-time

2.20 [M] Show the processor stack contents and the contents of the stack pointer, SP, immediately after each of the following instructions in the program in Figure 2.18 is executed. Assume that [SP] = 1000 at Level 1, before execution of the calling program begins.

(*a*) The second Store instruction in the subroutine SP : 972 ; stack : [..., R5, R4, R3, R2, 100, N, NUM1]

(*b*) The last Load instruction in the subroutine SP : 972 ; stack : [..., R5, R4, R3, R2, 100, SUM, NUM1]

(*c*) The last Store instruction in the calling program SP : 992 ; stack : [... SUM, NUM1]

★ START: SP : 1000 ★

2.26 [M] The dot-product computation is discussed in Section 2.12.1. This type of computation can be used in the following signal-processing task. An input signal time sequence IN(0), IN(1), IN(2), IN(3), . . . , is processed by a 3-element weight vector (WT(0), WT(1), WT(2)) = (1/8, 1/4, 1/2) to produce an output signal time sequence OUT(0), OUT(1), OUT(2), OUT(3), . . . , as follows:

$$\begin{aligned}
\mathrm{OUT}(0) &= \mathrm{WT}(0) \times \mathrm{IN}(0) + \mathrm{WT}(1) \times \mathrm{IN}(1) + \mathrm{WT}(2) \times \mathrm{IN}(2)\\
\mathrm{OUT}(1) &= \mathrm{WT}(0) \times \mathrm{IN}(1) + \mathrm{WT}(1) \times \mathrm{IN}(2) + \mathrm{WT}(2) \times \mathrm{IN}(3)\\
\mathrm{OUT}(2) &= \mathrm{WT}(0) \times \mathrm{IN}(2) + \mathrm{WT}(1) \times \mathrm{IN}(3) + \mathrm{WT}(2) \times \mathrm{IN}(4)\\
\mathrm{OUT}(3) &= \mathrm{WT}(0) \times \mathrm{IN}(3) + \mathrm{WT}(1) \times \mathrm{IN}(4) + \mathrm{WT}(2) \times \mathrm{IN}(5)\\
&\vdots
\end{aligned}$$

All signal and weight values are 32-bit signed numbers. The weights, inputs, and outputs, are stored in the memory starting at locations WT, IN, and OUT, respectively. Write a RISC-style program to calculate and store the output values for the first *n* outputs, where *n* is stored at location N.

Hint: Arithmetic right shifts can be used to do the multiplications.

$\frac{1}{8} = 2^{-3}$ (3 right shifts)

$\frac{1}{4} = 2^{-2}$ (2 right shifts)

$\frac{1}{2} = 2^{-1}$ (1 right shift)

Assembly:

number 'n' as loop counter

```
load R1, N
load R2, IN
load R3, WT
load R4, OUT
move R5, #0

Loop:
    load R6, (R2)
    load R7, 4(R2)
    load R8, 8(R2)
    move R9, #0
    move R10, #0
    move R11, #0
    move R12, R6
    ShiftRightArithmetic R12, #3
    move R13, R7
    ShiftRightArithmetic R13, #2
    move R14, R8
    ShiftRightArithmetic R14, #1
    Add R15, R12, R13
    Add R15, R15, R14
    Store R15, (R4)
    Add R2, R2, #4
    Add R4, R4, #4
    Add R5, R5, #1
    subtract R1, R1 #1
    BranchNotZero R1, Loop
```

load inputs

empty R

shifts

storage

increment counter

BNRE; return to subroutine "loop"