Single Cycle Datapath with MIPS instruction sw

Instruction sw

The below snippet of code gives you a generic idea of the use of the instruction sw. I will try and step through the datapath in the surrounding columns by breaking down some of the details.

```
.data
numOne: .word 17

.globl main
.text
main:

la $s3, numOne
sw $s1, 0($s3)


li $v0, 10
syscall
.end main
```

code.asm

I-Type instruction classes

opcode	rs	rt	memory address offset
sw	$19	$17	0
	$s3	$s1	
101011	10011	10001	0000 0000 0000 0000

I-Type instruction

Please notice in the SPIM simulation located in the top, second column of this file. You can see where **sw** is executed and that it is a match for the base 16 result shown in the cells below.

Number	Base
10101110011100010000000000000000	in base 2
2926641152	in base 10
0xae710000	in base 16

Register File

rs	rt	memory address offset
10011	10001	0000 0000 0000 0000
Inst[25-21]	Inst[20-16]	Inst[15-0]
Read Register 1	Read Register 2	Sign Extension Unit
base address in memory		32 bit extended value
	ALUsrc 0	ALUsrc 1
1st Operand	Write Data	2nd Operand

I used 0 as an offset so even though the 1st Operand and 2nd Operand are being added the address is not changed. Regardless the MemWrite should be 1, meaning write to memory and the write data "word = 17" to the summed rs and memory address offset: 10011.

ALU control lines

sw has the ALUOp 00 and the desired ALU action is add or 0010.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

SPIM

This is just an example of the addresses being used for each instruction including **sw** $17, 0(\mathbf{19})$ as you can see, with the exception of the **jal** instruction, the address is incremented by four each instruction. I believe this would mean **PCSrc** recieves a branch target control signal or **1**, for the **jal** instruction, while all the other instructions in this code including **sw** would instead be **0** and therefore **PC + 4**. The line you are looking for below is:
[0x00400028] 0xae710000 sw $17, 0(\mathbf{19})$ **; 9: sw** $s1, 0(\mathbf{s3})$
Here is the printout of stepping through the aforementioned code using SPIM:

```
[0x00400000]  0x8fa40000  lw $4, 0($29)              ; 183: lw $a0 0($sp)   # argc
(spim) s
[0x00400004]  0x27a50004  addiu $5, $29, 4           ; 184: addiu $a1 $sp 4   #
    argv
(spim) s
[0x00400008]  0x24a60004  addiu $6, $5, 4            ; 185: addiu $a2 $a1 4   #
    envp
(spim) s
[0x0040000c]  0x00041080  sll $2, $4, 2              ; 186: sll $v0 $a0 2
(spim) s
[0x00400010]  0x00c23021  addu $6, $6, $2            ; 187: addu $a2 $a2 $v0
(spim) s
[0x00400014]  0x0c100009  jal 0x00400024 [main]      ; 188: jal main
(spim) s
[0x00400024]  0x3c131001  lui $19, 4097 [numOne]     ; 8: la $s3, numOne
(spim) s
[0x00400028]  0xae710000  sw $17, 0($19)             ; 9: sw $s1, 0($s3)
(spim) s
[0x0040002c]  0x3402000a  ori $2, $0, 10             ; 12: li $v0, 10
(spim) s
[0x00400030]  0x0000000c  syscall                    ; 13: syscall
```

4 bytes long

The following python code shows how each hexidecimal location listed in the above SPIM simulation is incremented by **PC + 4** in decimal numbers, with the exception of the branched instruction **jal** which branched **16 bits** or **(4 * 4bit)** increments. I'm assuming the **jal** instruction is used when **main** begins in SPIM.

```
#!/usr/bin/env python
a = int('0x00400000',16)
print a
b = int('0x00400004',16)
print b
c = int('0x00400008',16)
print c
d = int('0x0040000c',16)
print d
e = int('0x00400010',16)
print e
f = int('0x00400014',16)
print f
#jal command increments by 16 or 4 * 4
la = int('0x00400024',16)
print "la $s3, numOne:"
print la
sw = int('0x00400028',16)
print "sw $s1, 0($s3):"
print sw
li = int('0x0040002c',16)
print li
syscall = int('0x00400030',16)
print syscall
```

4bytes.py

4194304
4194308
4194312
4194316
4194320
4194324
la $s3, numOne:
4194340
sw $s1, 0($s3):
4194344
4194348
4194352

Single Cycle Datapath