Graehme Blair, Drew Hasse
ghb130, ajh348
EECS 361 – Single Cycle Processor

Design

-ALU

The ALU described in the first lab was used as the ALU in our processor.

-IFU

The instruction fetch unit was made following the design from lecture eight. The IFU takes in a clock signal, initialization control signal, 16-bit immediate, and two signals to check if the branch should be taken. The IFU outputs a 32-bit address that the instruction memory uses to get the current instruction. A 30-bit PC register that writes on each clock cycle was made, as well as an n-bit extender. The n-bit extender was chosen so that it could be used for the 30-bit immediate extension in the IFU and the immediate extension for i-type instructions as well as loads and stores. A 30-bit pc was used because the bottom bits of the address out are 00 in all cases. The upper 30 bits of the address out are PC+4 when the two branch signals (zero and branch in our implementation) are high, which is what should happen when a branch is not taken. When the branch is taken, the zero and branch signals are high, and the upper 30 bits of the target address are calculated by sign extending the immediate to 30 bits and, and adding it to PC+4. The main issues faced with the IFU was initializing the value in the PC. At first, we implemented the IFU as a closed loop, only taking in a clock signal, and there was no way to set a starting value. This was fixed by adding an initial value constant and an init signal, that is high for the first cycle and then low at all other times. This init signal is used as the select to a mux placed before the PC, which selects either the initial value or the new value of PC.

-Register File

The register file takes in a clock; registers a, b and w; Bus W; and write enable. It outputs Busses A and B. Internally it maintains 31 32-bit registers. It determines the write enable bit for each internal register by decoding RW and ANDing the write enable input bitwise with the resulting vector. Each internal register is then fed the correct write enable bit from the vector. The outputs for bus A and bus B are determined by using 32 32 to 1 muxes with bits from each register as input and RA and RB as select signals. Since the zero register always contains zero in MIPS register zero is just statically allocated a value of zero. The main problem for this component was making sure that the zero register couldn't be written to without complicated logic. This was solved by removing the zero 'register' and replacing it with a constant signal.

-Datapath

The datapath component takes in all necessary control signals as input and outputs the current instruction. The datapath contains muxes, a register file, an extender, an ALU, an IFU, instruction memory and data memory. These are hooked together in almost exactly the same way as shown on slide 15 of lecture 8. The key differences are the lack of a jump signal and the addition of a few muxes. The first additional mux chooses RA to be Rs or Rt based on whether or not the current instruction is a shift. The next mux chooses the input of the extender to be either the last 16 bits of the instruction if it is not a shift or bits 10 – 6 if it is a shift. The next mux chooses the input of the ALU to be either the output of the ALUsrc mux for any instruction but shift or the output of the extender if it is a shift. This was needed because shift is considered to be an r-type instruction but needed input from the immediate field. The next mux chooses either the output of the previously described mux or 0 as the

input for the ALU based on whether the instruction is BGTZ since the bus b input to the ALU needs to be zero in this case. The final additional mux picks the zero signal input to the IFU to be either the ALU zero signal, NOT ALU zero, or NOT ALU zero AND NOT ALU msb depending on whether the instruction is beq, bne or bgtz, respectively. The main issue encountered in the datapath component was finding out that certain instructions needed different inputs than what we originally thought (looking at you SLL). This was solved through the use of the muxes described above.

-Control (See end of document for table)

The control was divided into two components, the ALU control and the main control. Both were implemented using a PLA, because it would be simple to find the necessary output signals and would not require complicated logic. A PLA inverter and six-to-one AND gates were made to implement the PLA. The inverter works by taking in the six bits of data, as well as six bits that correspond to whether that bit of data needs to be inverted. If inv(i) is 1, then data(i) will be inverted before it is put into the AND gate. One note about this is that they are currently separate components (the PLA inverter, and the AND6to1), however, it would have been simpler to combine these two into one component for the PLA, rather than having one of each for each minterm, but I realized it too late. The current implementation still functions, properly but is not the most efficient in terms of the number of lines of code required. For the ALU control, the inputs are the opcode and the function code and the output is the necessary function for the ALU. Espresso was used to minimize the logic of the PLA's. The R-type instructions looked at the function codes and found the output based on these, and the non R-type instructions looked at the opcode. Muxes were used to select either the output from the R-type PLA, or the non R-type PLA, with the select bit being all the opcode bits OR'd with each other, and the result inverted. This is because the opcode for R-type instructions is 000000, so the select will only be 1, when the instruction is R-type. The main issues we encountered with the ALU control, was typos in the .pla file used by Espresso. For the SLT and SLTU, we initially had the output as the subtraction ALU opcode, instead of the SLT or SLTU opcode. This generated the wrong PLA and had to be corrected.

The main control was implemented the same way as the ALU control for simplicity. The main control takes in the opcode and generates the nine control signals using a PLA. The output signals are: RegDst, RegWr, Branch, ExtOp, AluSrc, MemWr ,MemtoReg, BrSel1, BrSel0. Most of the control signals are the same as in the lecture; however, we needed to add a 2-bit branch select signal, that would feed in the appropriate value to the zero input of the IFU, depending on the type of branch. BNE, BEQ, and BGTZ all have different conditions in which they should branch, so they were fed into a 3-to-1 mux with BrSel as the two select bits, and the appropriate signal was passed to the IFU. We encountered problems similar to those from the ALU control when creating the main control. The main issues were typos or writing down the wrong control signal that needed to be generated before running Espresso. However, once the skeleton of the main control was made, these errors were not too difficult to fix. Both the controls, were able to be exhaustively tested due to the small number of inputs and outputs.

-Single Cell Processor

The SSP component takes in a clock and a pcInit signal. The pc init signal writes 00400020 into the program counter. This component was very simple since we created the control and the datapath as separate components. It consists of a main control and ALU control and the datapath component. The control units take in the pieces of the current instruction they need and output the control signals which are then fed into the datapth component, which outputs the current instruction. There were no major issues in the construction of this component.

Program Traces

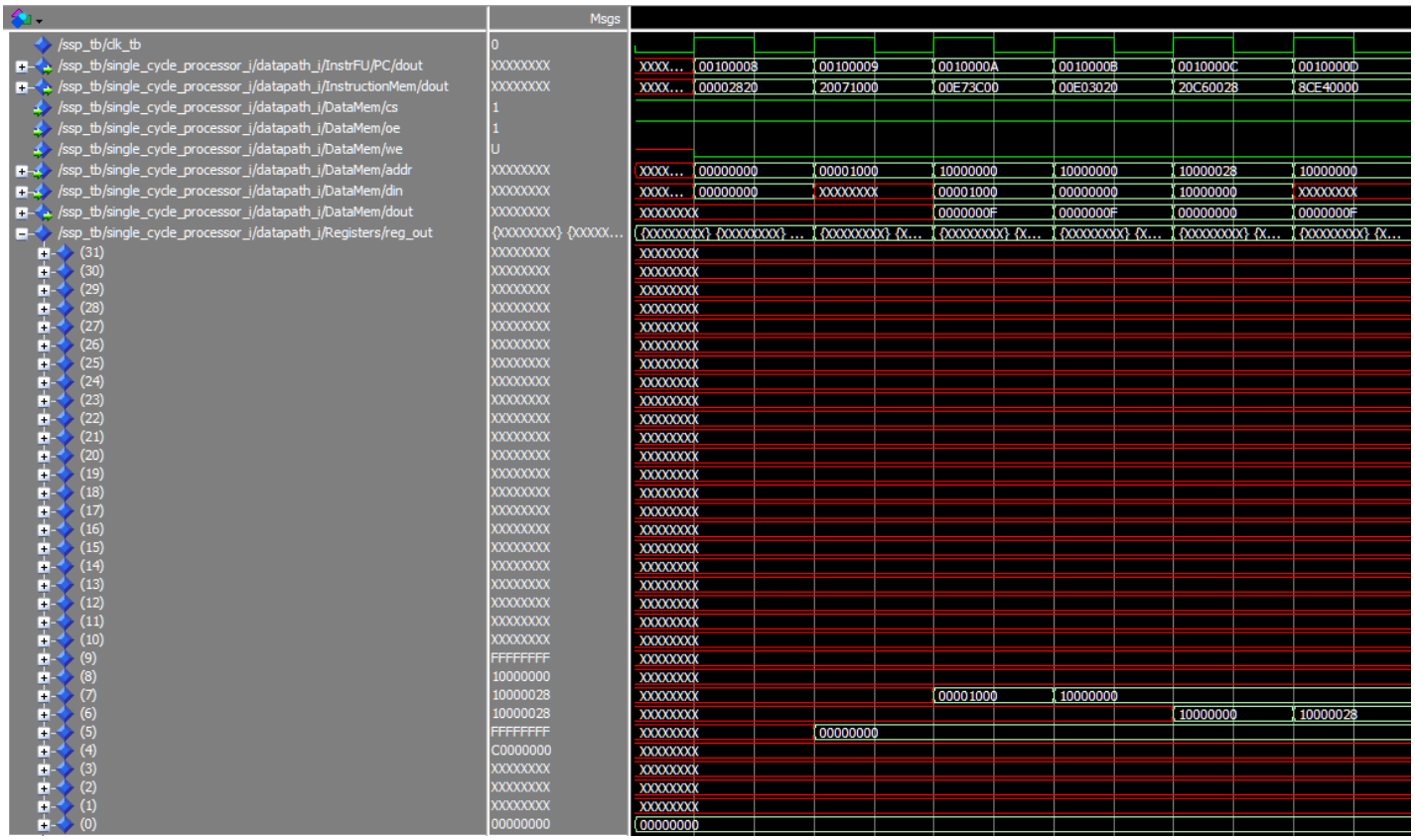
We determined the correct output of each program by converting it to assembly and running through them by hand. The hand 'run' programs compared to their actual traces are shown below. The programs were modified to load the results from memory to registers for easy access.

Unsigned Sum

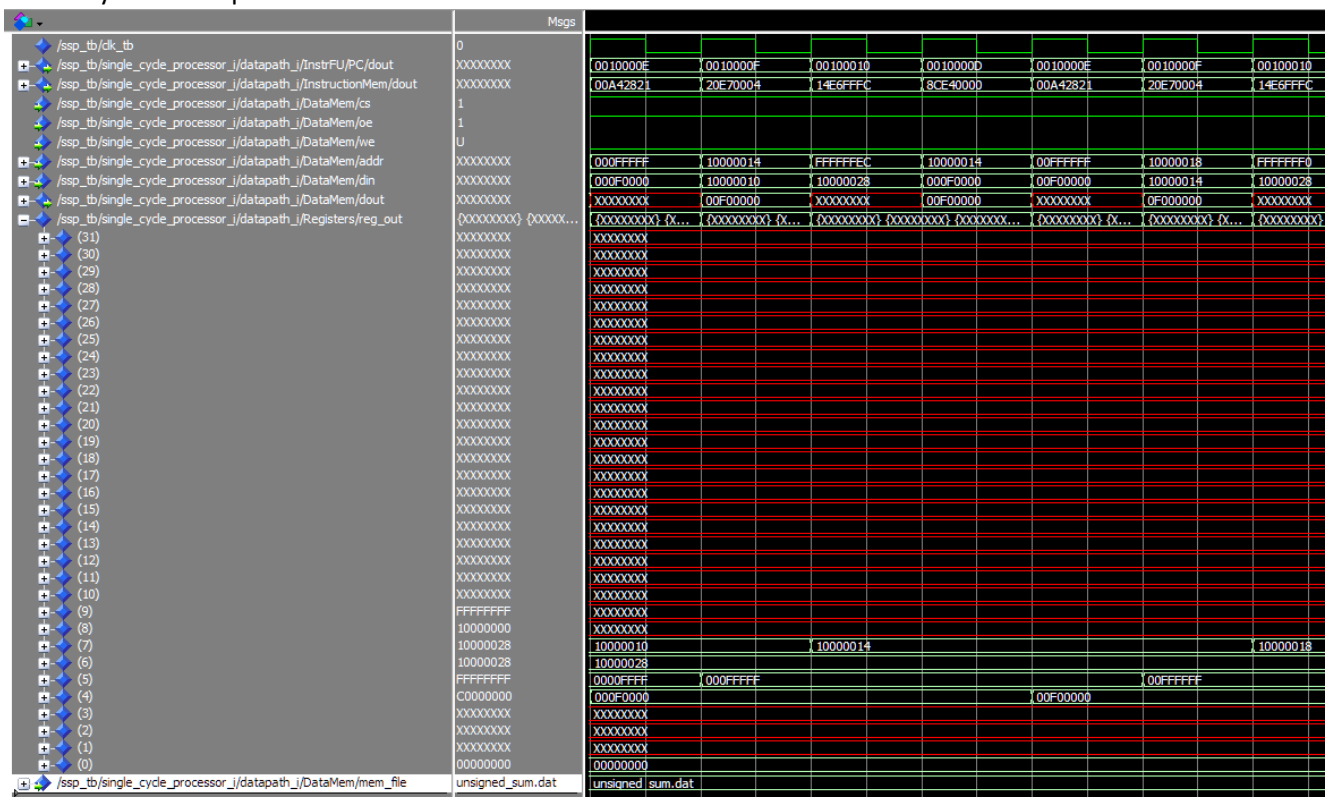
ADD $a1 $zero $zero a1 = 0
ADDI $a3 $zero 0x1000 a3 = 0x1000
SLL $a3 $a3 0x10000 a3 = 0x10000000
ADD $a2 $a3 $zero a2 = a3
ADDI $a2 $a2 0x0028 a2 = 0x10000028
LW $a0 0x0000 $a3 a0 = 0000000f
ADDU $a1 $a1 $a0 a1 = 0000000f
ADDI $a3 $a3 0x0004 a3 = 0x10000004
BNE $a3 $a2 0xFFFC
SW $a1 0x0000 $a3

--Sums over elements 10000000 to 100000024 stores in 10000028
--Output is correct if 10000028 constains ffffffff

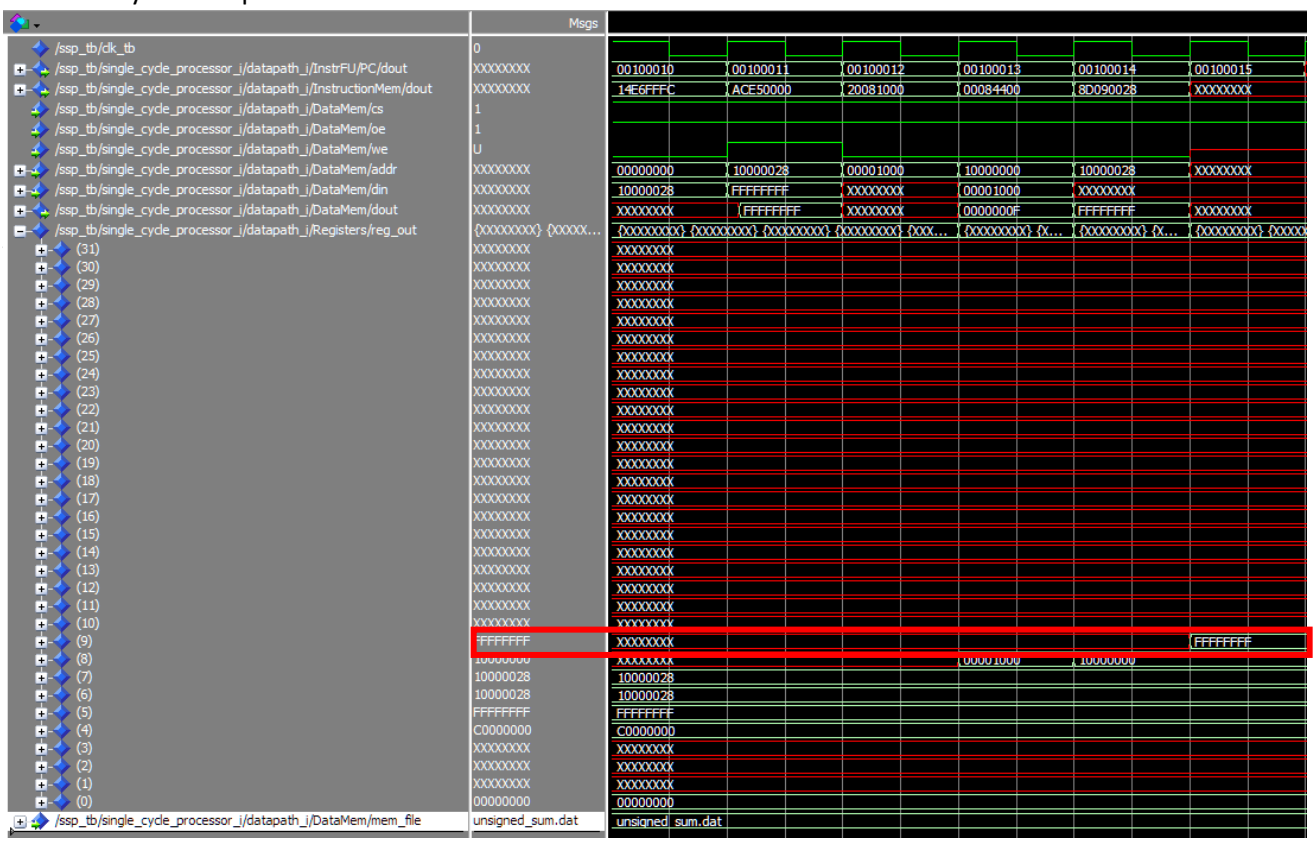
Initial few cycles of operation:



A few cycles mid operation:



Final few cycles of operation:



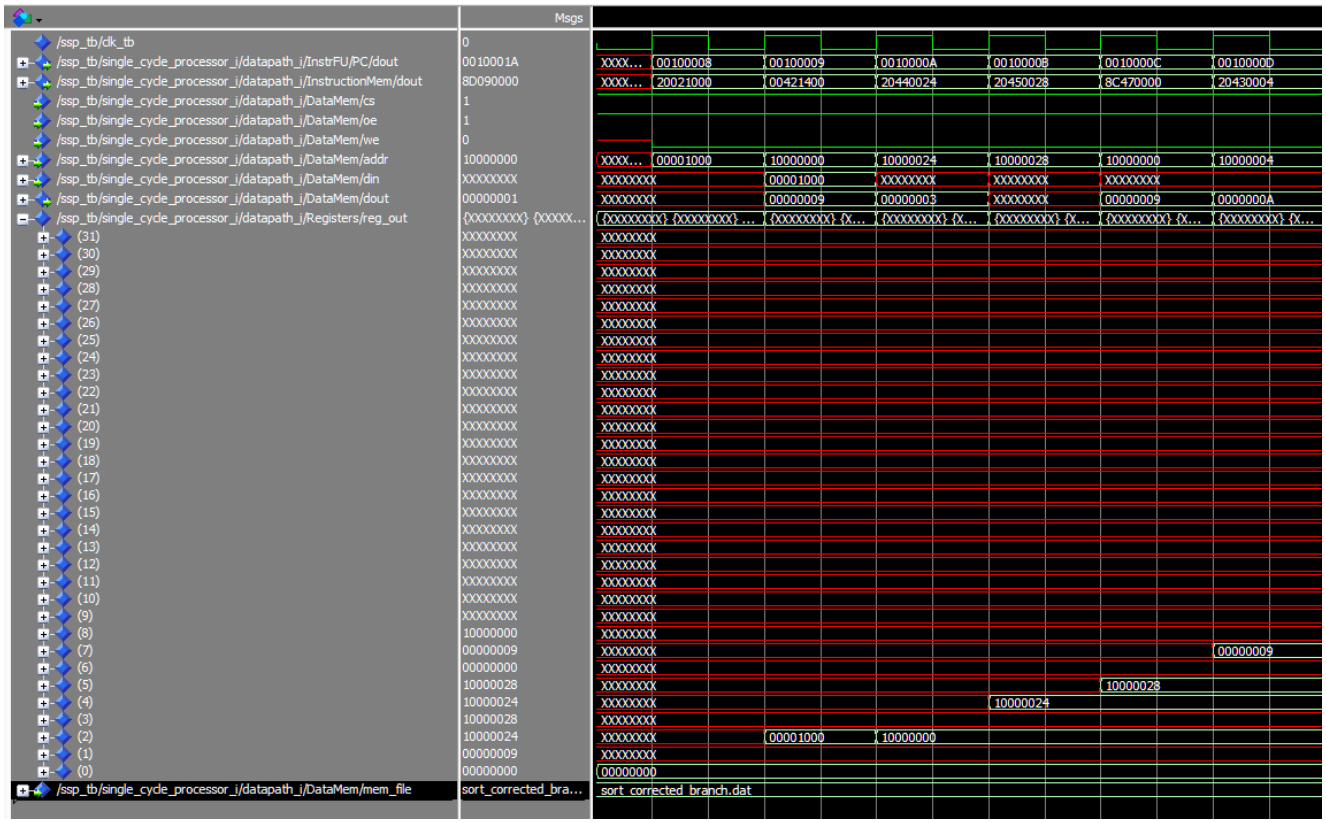
The red box above hold the value stored in location 10000028. FFFFFFFF checks out!

Sort Corrected Branch

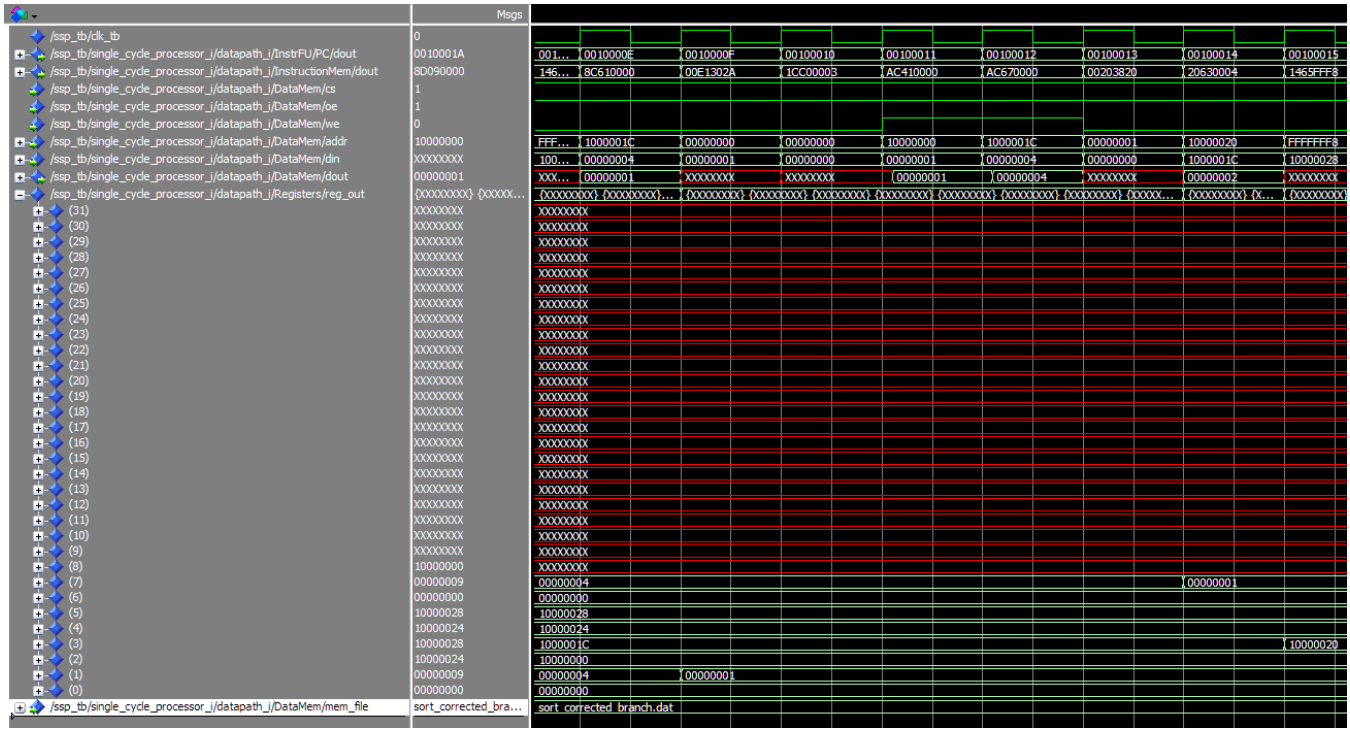
ADDI $v0 $zero 0x1000	v0 = 1000	
SLL $v0 $v0 10000	v0 = 10000000	
ADDI $a0 $v0 0x0024	a0 = 10000024	
ADDI $a1 $v0 0x0028	a1 = 10000028	
LW $a3 0x0000 $v0	a3 = 00000009	
ADDI $v1 $v0 0x0004	v1 = 10000004	
LW $at 0x0000 $v1	at = 0000000a	at = 00000008
SLT $a2 $a3 $at	a2 = 1	a2 = 0
BGTZ $a2 0x0003	branch taken	branch not taken
SW $at 0x0000 $v0		10000000 <= 00000008
SW $a3 0x0000 $v1		10000008 <= 00000009
ADD $a3 $at $zero		a3 = 00000008
ADDI $v1 $v1 0x0004	v1 = 10000008	v1 = 1000000c
BNE $v1 $a1 0xFFF8	branch taken	
ADDI $v0 $v0 0x0004		
BNE $v0 $a0 0xFFF4		

--ITS BUBBLE SORT - ish
--Loops through 10000000 to 10000024 swapping data when it finds a larger item at a higher address.
--Completed process should have memory: 1,2,3,4,5,6,7,8,9,a

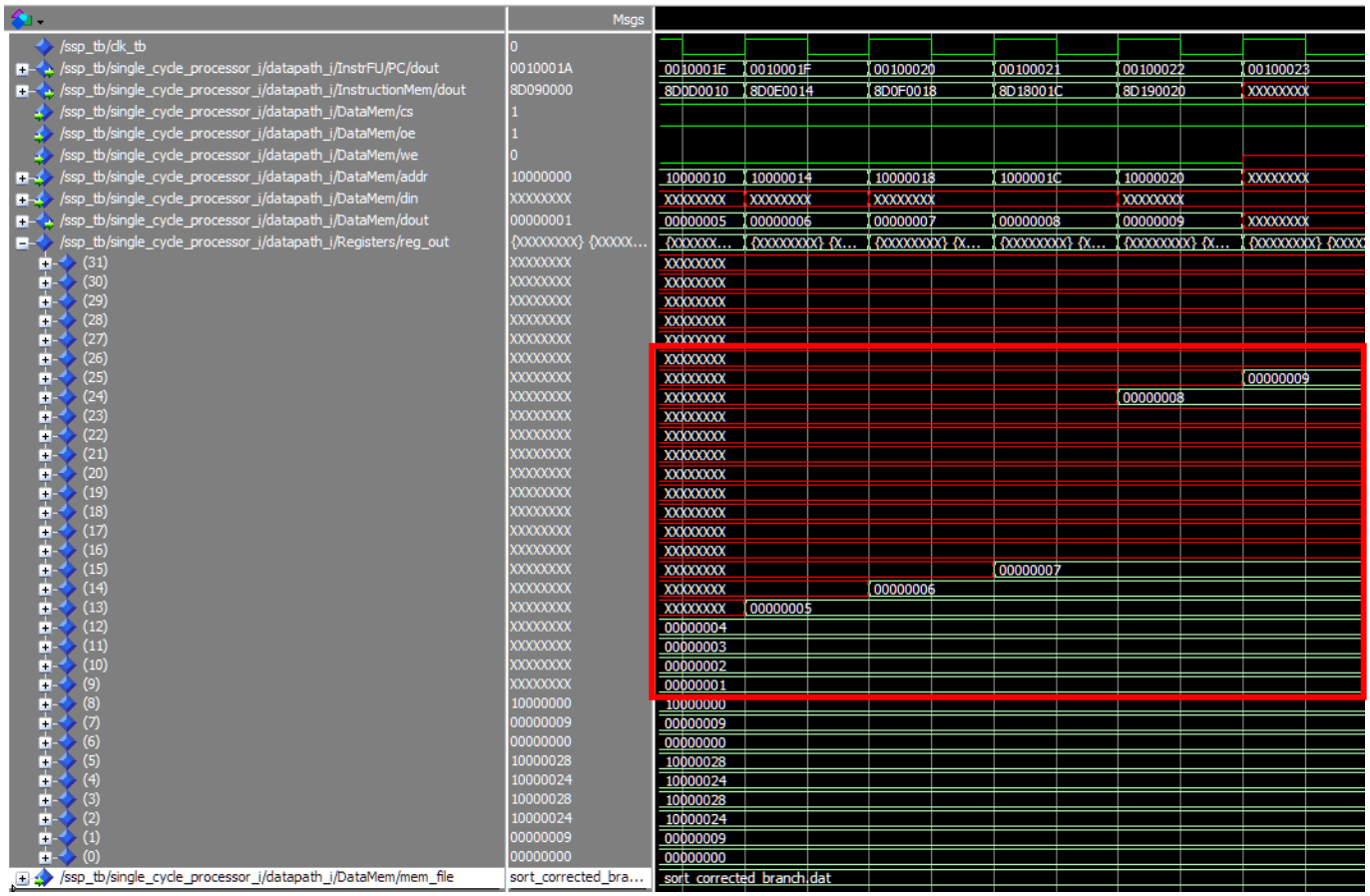
First few cycles of operation:



A few cycles mid operation:



The final few cycles:



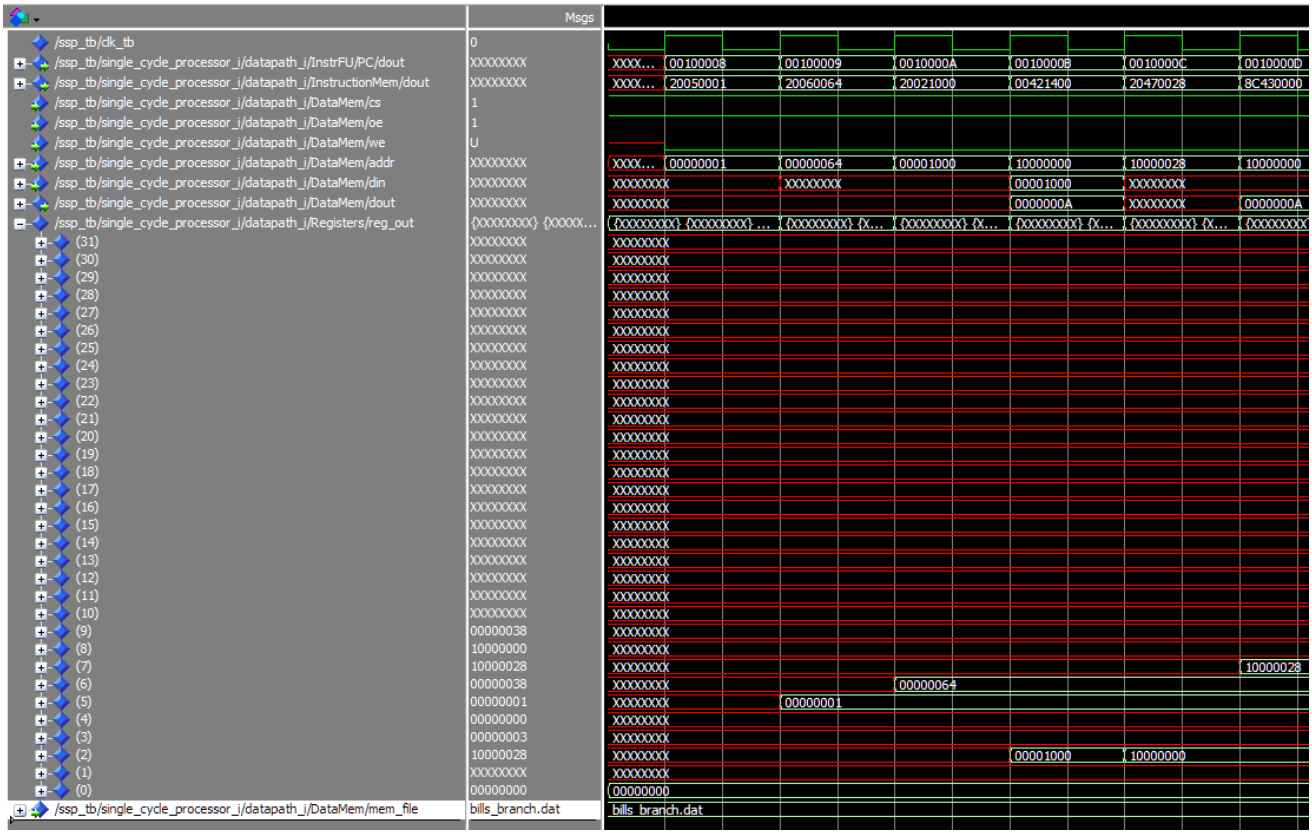
The red box above contains (from bottom up) the memory locations 10000000 to 10000020. 1,2,3,4,5,6,7,8,9 checks out!

Bills Branch

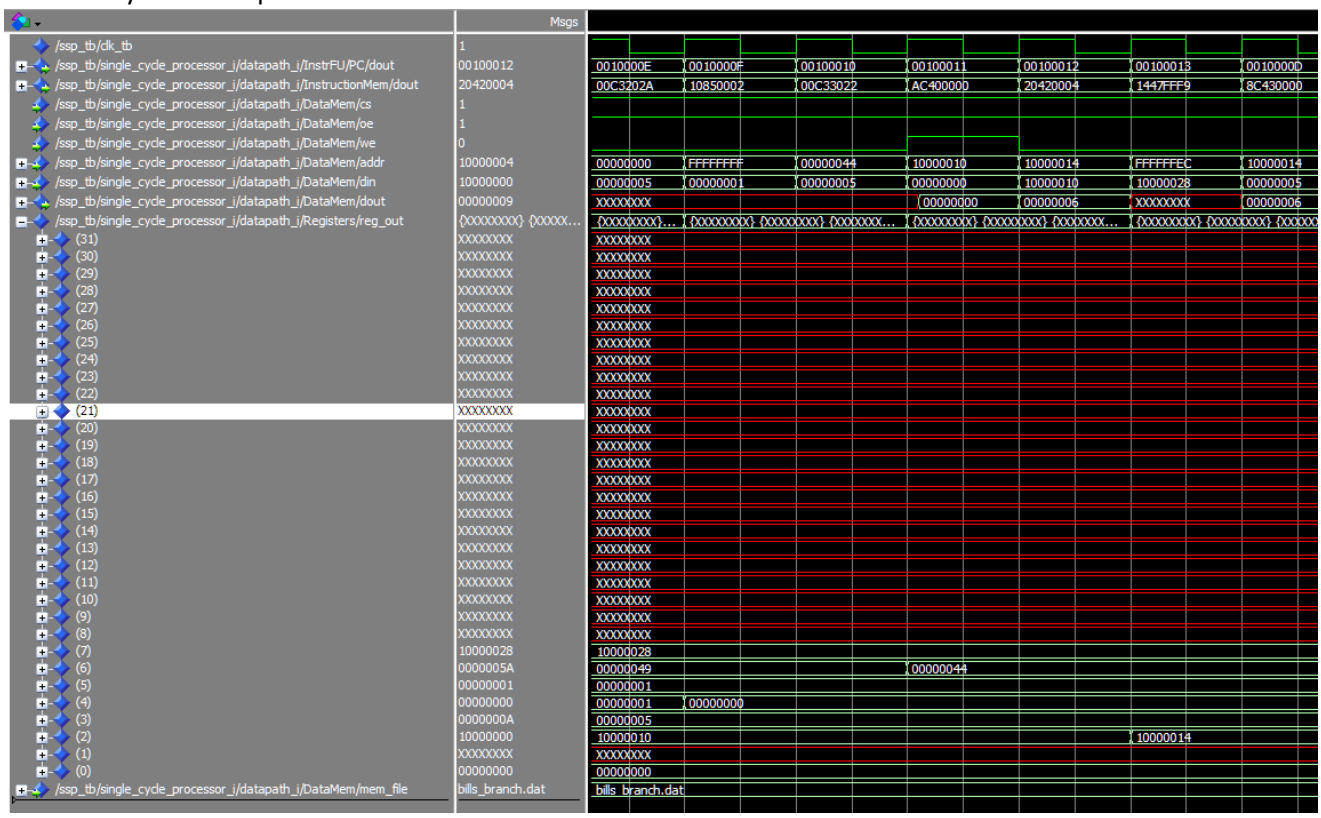
ADDI $a1 $zero 0x0001	a1 = 0001	
ADDI $a2 $zero 0x0064	a2 = 0064	
ADDI $v0 $zero 0x1000	v0 = 1000	
SLL $v0 $v0 0x10000	v0 = 10000000	
ADDI $a3 $v0 0x0028	a3 = 10000028	
LW $v1 0x0000 $v0	v1 = 0000000a	v1 = 00000009
SLT $a0 $a2 $v1	a0 = 0 (64 > a)	a0 = 0 (5a > 9)
BEQ $a0 $a1 0x0002	branch not taken	branch not taken
SUB $a2 $a2 $v1	a2 = 5a	a2 = 51
SW $zero 0x0000 $v0	10000000 <= 0	10000004 <= 0
ADDI $v0 $v0 0x0004	v0 = 10000004	v0 = 100000008
BNE $v0 $a3 0xFFF9	branch taken	branch taken
SW $a2 0x0000 $a3		

-- Iterates through 10000000 to 10000024 subtracting value found there from
-- a running total starting at 100, afterwards it writes 0 to the mem location
--If value is too large it skips it
--aftwerwards it stores the amount leftover into 10000028
-- Completed process should have memory: 0,0,0,2bc,0,0,190,0,0,0,38

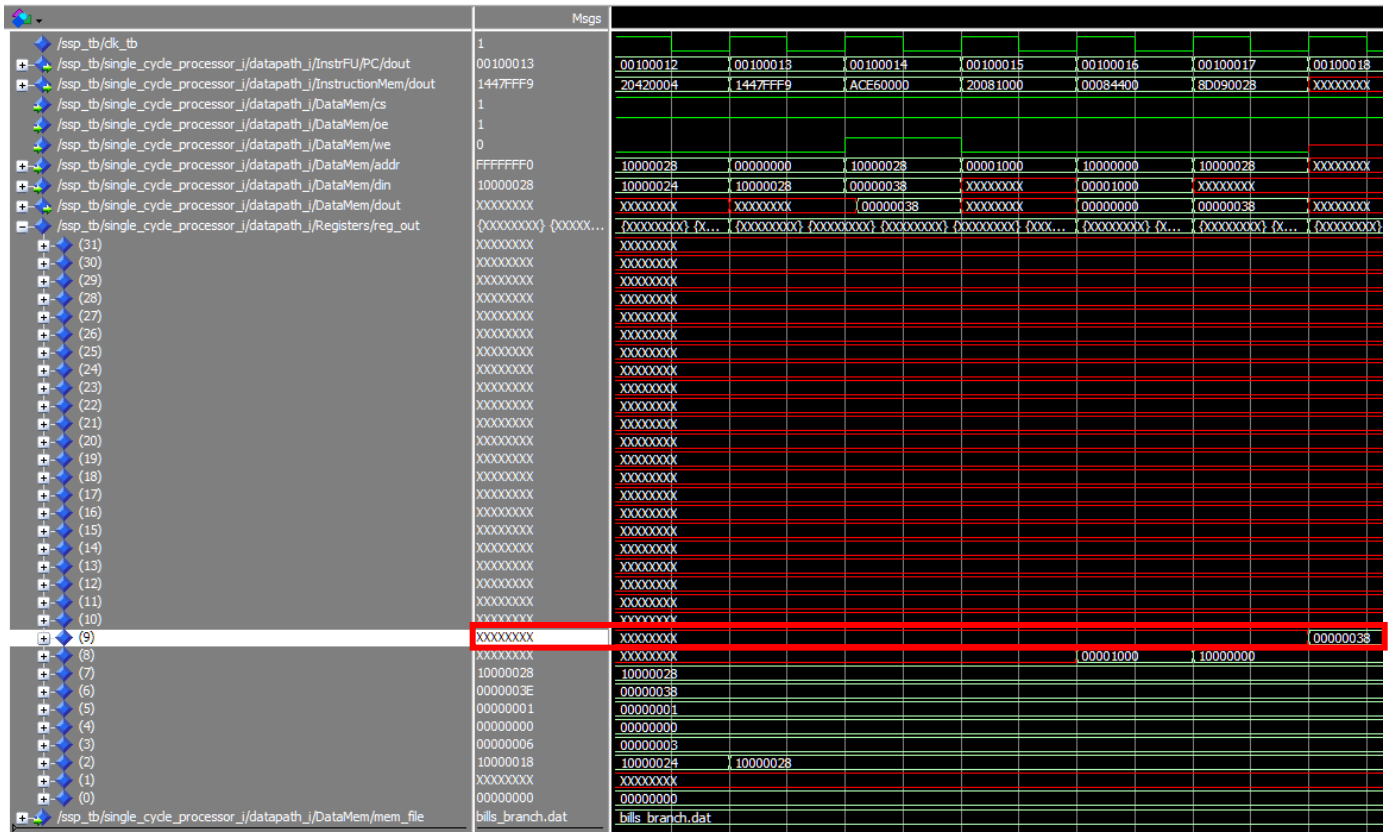
The first few cycle of operation:



A few cycles mid operation:



The final few cycles:



The red box above contains memory location 10000028. 00000038 checks out!

Control Signal Tables

Main Control

Name\Type	R-Type	LW	SW	BEQ	BNE	BGTZ	ADDI
RegDst	1	0	X	X	X	X	0
RegWr	1	1	0	0	0	0	1
Branch	0	0	0	1	1	1	0
ExtOp	X	1	1	X	X	X	1
ALUsrc	0	1	1	0	0	0	1
MemWr	0	0	1	0	0	0	0
MemtoReg	0	1	X	X	X	X	0
BrSel(1)	X	X	X	0	0	1	X
BrSel(0)	X	X	X	0	1	X	X

ALU Control

R-type Instructions:

Function Code	ALUop
100000	0000
100001	0000
100010	0001
100011	0001
101010	0101
101011	0110
100100	0010
100101	0100
000000	1000

Non-R-Type Instructions

OP Code	ALUop
100011	0000
101011	0000
000100	0001
000101	0001
000111	0001
001000	0000