BTSB-ISA

9-bit single-cycle CPU

Ezequiel Herrera-Ortiz
Richard Bull
Cody Heiner

Overview

Operations Description

Code	*Name*	*Description*
aro	Arithmetic Operation	Using the value in register **$S**[3:2], we run indicated instruction on REG1 and REG2. We then store the result in REG1, except in the case of Compare.
hi	Set High Bits	Sets the bits of register **$I**[7:5] to the Immediate Value
lo	Set Low Bits	Sets the bits of register **$I**[4:0] to the Immediate Value
SPC	Specialized Pattern Counting	parameter: **$r** read memory at location given by **$r** -> w[0..7] **$t5** += (w[0]&w[1]&w[2]&w[3] \| w[1]&w[2]&w[3]&w[4] \| w[2]&w[3]&w[4]&w[5] \| w[3]&w[4]$w[5]&w[6] | w[4]&w[5]$w[6]&w[7])
mov	Move	Dest Reg = Src Reg
INC	Increment Register	parameter: **$r** **$r** += 1
MFL	Multiply Function Low	**$s5** += **$t5**[0] & (**$t4** << **$s6**)
MFH	Multiply Function High	**$s4** += **$t5**[0] & (**$t4** >> (8-**$s6**)) + **#CV**
ISC	Increment Shift Counter	**$s6** += 1 **#C Z** = [(**$t5** >> **$s6**) == 0]
SM	Right Shift Multiplicand	**$t5** = **$t5** >> 1
ldst	Load/Store	If **$S[4]** = 0, then memory location in Dest Reg = Src Reg. If **$S[4]** = 1, then Src Reg = memory location in Dest Reg
ssb	Set Setting Bits	Used to set the **$S** register which is used as a selector for loaded operation.
be	Branch Equal (Direct)	If **#C**[Z] = 1, then PC = Register
bne	Branch Not Equal (Direct)	If **#C**[Z] = 0, then PC = Register
obe	Offset Branch Equal	If **#C**[Z] = 1, then PC = PC + immediate
obne	Offset Branch Not Equal	If **#C**[Z] = 0, then PC = PC + immediate
tally	Tally Counter	parameter: **$r** read memory at location **$r** → i tally[i] += 1 **if** tally[i] > tally_count **or** tally[i] >= tally_count **and** i < max tally_count = tally[i] max = i **$t1** = max
count	Count of Tally	parameter: **$r** **$r** = tally_count
halt	Halt	Used to signal the end of the program
tba	TBA	Coming soon!

Register Overview

Register Bank A		Register Bank B	
0000 - **$0**	← Constant 0 register	1000 - **$S**	← Setting Register
0001 - **$I**	← Immediate Register	1001 - **$s0**	← General use register
0010 - **$t0**	← General use register	1010 - **$s1**	← General use register
0011 - **$t1**	← General use register	1011 - **$s2**	← General use register
0100 - **$t2**	← General use register	1100 - **$s3**	← General use register
0101 - **$t3**	← General use register	1101 - **$s4**	← General use register
0110 - **$t4**	← General use register	1110 - **$s5**	← General use register
0111 - **$t5**	← General use register	1111 - **$s6**	← General use register

Setting Registers

***M** = Register Bank MSB

***A** = Arithmetic Function

***LS** = Load / Store Selection

These will be stored in our Setting Register **$S**

Register **$S** is described as follows:

Unspecified - 3 bits	***LS** - 1 Bit	***A** - 2 Bits	***M** - 2 Bits

Setting Codes for ***M**:

A	Bank A → Bank A
B	Bank B → Bank B
AB	Bank A → Bank B
BA	Bank B → Bank A

Questions

1. Have you made any changes to your ISA from lab 1? What were they? Why did you make them? [Keep in mind you need approval to do so]

A: We noticed that we had to rewrite an instructions for program 1 and program 3 as they did not run properly. For program 3 the old instruction did not take into account that we need the smallest value that occurs the most. In program 2 the shifting was done opposite to the correct way and the implicit registers were changed. However the changes to program 1's MFL and MFH instruction were partially for convenience. We also needed to do immense error checking and corrections to almost all instructions implemented and as a side-effect of these changes some extra functionality is now available, however it is unused. We also have realized one of our instructions, SM which was intended for Program 1 is unneeded. If we really needed a new instruction we could remove this one.

2. What are your dynamic instruction counts for program 1? program 2? program 3?

A: Based on worst case scenarios.
Program 1: 57
Program 2: 269
Program 3: 270

3. What could you have done differently to better optimize for dynamic instruction count?

A:The bulk of our dynamic instruction count comes from our looping. If we were to combine the compare function with our branching then our loops would improve by around 33%, taking into account that our loops are only 3-4 lines long. This would require that we make our compare take in 2 implicit registers. And we would only use the obne branching since its the only one we used in our program.

4. How successful were you at optimizing for ease of hardware design? Give examples.

A: We did a horrible job at optimizing for ease of hardware design. In focusing on lowering instruction count the hardware became much more complex. Our register file alone has 13 inputs and 4 outputs. The complexity added to our modules required a complex control signal structure. I'm not sure if God hates me, but the fact that all wires are purple by default in Verilog has caused some serious reflections on my life's choices.

5. What could you have done differently to better optimize for ease of hardware design?

A: To optimize for ease of hardware we should have omitted the special functions, as well as the different branching options. The special functions caused our ALU and Reg file to have excessive inputs and outputs. This also lead to us needing a much more complex control unit and datapath. As for branching, we have direct and indirect, however, we only use not equal offset, and equal to offset. However, these struggles came with a relatively low Dynamic Instruction Count.

6. How easy/difficult would it be to extend your design to a multicycle implementation? Give examples.

A: Much of the initial difficulty would be the immense amount of time spent on rewiring our schematic and painstakingly trying to figure out which one of the 100+ purple wires goes to what. However, after that most of our datapath implementation has kept to making sure each instruction can be split up into a multicycle implementation rather easily. We only have small exceptions for our special functions which would require “special consideration” because some call out to memory on their own.

7. What do you expect to be the biggest challenge in creating a pipelined design? Please give an answer that is specific to your ISA.

A: Keeping track of all the controls. In this lab assignment, we had great difficulty testing our program due to all the control signals that we had to follow. I know that going into a pipeline machine will greatly increase the complexity of the control unit, thus leading to a much more complex machine to follow the data flow.

8. What might you have done differently if a priority was ease of programming? Give examples.

A: Our ISA contains functions that we never use. This is because we wanted our ISA to be flexible and able to run different types of programs. To make our ISA fast, we created machine- and program-specific instructions. To combine flexibility and speed, we had to overload instructions using dedicated condition code registers. In eliminating the special functions, we would have the opt codes to eliminate the need of condition codes.

9. What instruction takes longest on your machine (and thus would set the cycle time)? (Use rough estimates).

A: The largest instruction in our ISA is the SPC (Special Pattern Counter). This operation goes through registers, data memory, ALU, and back to registers.

10. What might you have done differently if a priority was short cycle time? (again, use rough estimates). Give examples.

A: To lower our cycle time we would have to reduce the SPC instruction, since it sets the cycle time. The only way we would be able to do this is split the instruction up into two parts where one part loads an address and another instruction checks whether the value loaded is a pattern or not. With the way it is setup now I believe it actually has the longest datapath required of all instructions. When we have to implement a pipelined machine we will have to split it up into it’s individual parts.

Program 1: Timing Diagram, assembly code, and machine code per program

Assembly	Machine Code
ssb Ld,Add,AA	011000000
hi 0	100000000
lo 1	100100001
ldst $I, $t5	010001111
lo 2	100100010
ldst $I, $t4	010001110
ssb Ld,Add,AB	011000001
mov $0 $s4	001000101
mov $0, $s5	001000110
mov $0, $s6	001000111
ssb Ld, Slt, AA	011001100
mov $t5, $t1	001111011
slt $t1 $t4	000011110
ssb Ld,Cmp, AA	011001000
cmp $t1, $0	000011000
obe 4	101100100
mov $t4 $t3	001110101
mov $t5 $t4	001111110
mov $t3, $t5	001101111
mfl	110000000
mfh	110010000
isc	110011000
obne -3	101111101
ssb St, Add, AB	011010001
lo 3	100100011
st $I, $s4	010001101
lo 4	100100100
st $I, $s5	010001110
halt	110111000

Program 2: Timing Diagram, assembly code, and machine code per program

Assembly:	**Machine Code:**
lo 0	100100000
hi 32	100000001
ssb Ld, Add, AA	011000000
mov $I, $t0	001001010
lo 0	100100000
hi 96	100000011
mov $I, $t1	001001011
mov $0, $t5	001000111
ssb St, Cmp, AA	011011000
SPC $t0	110001010
INC $t0	110100010
cmp $t1, $t0	000011010
obne -3	101111101
lo 5	100100101
hi 0	100000000
st $I, $t5	010001111
halt	110111000

Program 3: Timing Diagram, assembly code, and machine code per program

Assembly:	**Machine Code:**
ssb Ld, Cmp, AA	011001000
lo 0	100100000
hi 128	100000100
mov $I, $t0	001001010
hi 192	100000110
mov $I, $t2	001001100
tally $t0	110110010
inc $t0	110100010
cmp $t0, $t2	000010100
obne -3	101111101
count $t3	110111101
ssb St, Add, AA	011010000
lo 126	100111110
hi 126	100000011
st $I, $t1	010001011
lo 127	100111111
st $I, $t3	010001101
halt	110111000