SSBY Architecture 8 Bit Processor

Yash Sharma, Shalin Patel, Matthew Cavallaro

1. ABSTRACT

Our ssby architecture is an 8-bit processor with a reduced instruction set computing (RISC) design. The architecture is a Harvard Architecture implementation with a 4 byte cache for data memory. Our architecture was designed with the four design principles of MIPS in mind: smaller is faster, make the common case fast, simplicity favors regularity, and good design demands good compromise. The processor is able to carry out basic leaf and nested procedures, signed addition, and memory access. We have an assembler/linker written in C++ that converts a ssby assembly program into 8-bit machine code. The machine code is executed by our processor. The processor was implemented in single cycle, non pipelined format in Verilog.

Instruction	M-format	ALU src	ALU control	Branch	Jump	MemtoReg	MemWrite	RegWrite
add	0	0	00	0	0	0	0	1
addi	0	1	00	0	0	0	0	1
sw	0	1	00	0	0	x	1	0
lw	0	1	00	0	0	1	0	1
beq	0	0	01	1	0	x	0	0
slti	0	1	10	0	0	0	0	1
Li (lui) *	1	1	11	0	0	0	0	1
Li (lli) **	1	1	00	0	0	0	0	1
j	x	x	xx	x	1	x	0	0

* Load upper immediate if li reg contains a zero
**Load lower immediate if li reg contains a one

M-Format Instructions

opcode	R1	immediate

7 5 4 0

I-Format Instructions

opcode	R1	R2	immediate

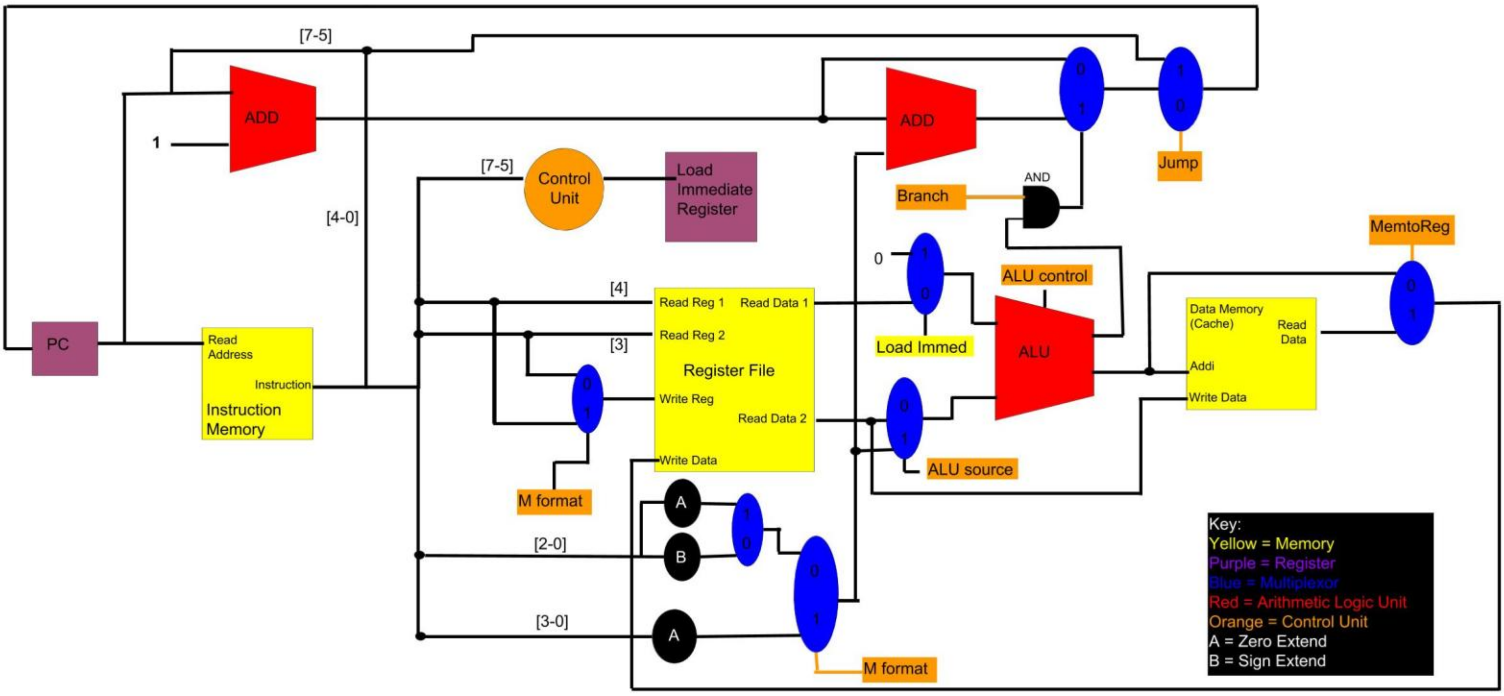
7 5 4 3 0

J-Format Instructions

opcode	Address

7 5 0

2. *DESIGN*



We aimed at achieving a Reduced Instruction Set Computing (RISC) architecture. Accordingly, we only have eight instructions. Based on our requirement for leaf/nested procedures we decided to implement “beq” and “jump”. To handle signed addition we implemented “add”. To handle memory access we implemented “sw” and “lw”. We decided to have two separate commands for memory access to avoid additional complexity. We implemented “li” in order to make it easier to load values, hence making the common case fast. We implemented “addi” in order to avoid constant calls to “li” or “lw,” also making the common case fast. We implemented “slti” in order to improve the completeness of our functionality and have loops that can handle more than equality checking. We decided to have the majority of commands utilize immediates (slti, addi, li) to compensate for the limited number of registers.

We have a RISC architecture, however, one command, load immediate (li), may be interpreted as “complex” because it must be called twice in a row to properly work. The RISC architecture allowed us to keep the datapath smaller, in line with the design principle that smaller is faster.. Also, we’re able to keep the number of formats small (3) which also reduces complexity of the design since there are less muxes.

We decided to have two general purpose registers so that only one bit of our instructions would be used to access the registers. This allows for an extra bit stored in the immediate for every register operand. This makes our processor more efficient because smaller is faster. We have access to a larger range of immediate values and thus can avoid constantly calling li (which is a 2 cycle instruction). Also, the logic for branch-if-equal would have to change since it would only have room for a 1 bit immediate. Suppose we instead have four registers. Then, we could only have two registers (say $00 and $11) that work with beq. This would allow for more room for the immediate, but that comes at the cost of complexity in our datapath.

The addition of li as a two cycle command was not a big hit on our complexity since it only required addition of a mux and 1 bit LI register to operate. Thus it does not require a large hit in complexity/efficiency in our datapath but it allows to us to have a very useful instruction. This resonates with the design principle that good design demands good compromise.

We decided not to have a stack pointer register to meet our design principles. Having one of our two general purpose registers as the stack pointer would be nonsensical. We considered having a non-addressable register for the stack pointer and a special command to push/pop from the stack based on the given input. This instruction would take opcode, push/pop (1 bit), and the register (1 bit) making it a 5 bit command instruction which wouldn’t take full advantage of the 8 bits we have. Also, this command would add a LOT of complexity to our datapath and require a new format. This goes against the design principle smaller is faster. In addition, this instruction would be complex and that would against our goal of creating a RISC architecture. Given this all, we decided to let the programmer include a stack pointer in data memory. By creating a smaller datapath that utilizes a cache, we believe our processor is more efficient than an implementation with a stack pointer register. This is the clearest example within our design choices that good design demands good compromise.

The majority of our instructions are a subset of MIPS, so we decided to stick closely to the MIPS datapath. Notable changes are the LI register, 0 extenders, jump implementation, and beq implementation. We use 0 extenders for immediates in the M format instruction since that format is only for li. The first time li is called on a register, the 4 bit immediate is shifted left 4 bits and the result is stored in the register. The second time li is called, the immediate is added to register. This time, the immediate must be zero extended so that we can properly update the latter four bits of li with addition in the ALU. The LI register is a one bit register that tracks how many times LI has been called. For jump, we decided to use a 5 bit immediate and prepend the first 3 bits of the PC. Thus you can only jump within a 32 instruction block of code. If we used a PC relative jump, we would have had to use a signed immediate which would limit the number of instructions we could jump. If we used absolute address jumping, our programs cannot be longer than 32 lines. This design choice is the same one made by MIPS, however our implementation is slightly different sign MIPS uses byte addressing with 4 byte instructions. Also, we decided to make the 3 bit immediate for beq unsigned. If the immediate would be signed, the range to branch would only be -4 to 3 lines. Branching back only four lines would almost never happen, thus we decided to make the value unsigned, and hence the range is 0 to 7. This is much more useful; however, jumping backwards has to be handled by the jump command now. This is yet again an example of good design demands good compromise.

3. *CODE*

Processor:

The processor was written in Verilog, a hardware description language (HDL). It was written in behavioral verilog. The datapath was written in verilog through modules created for each physical component, as well as the logic for "getting the next PC" (handle jump and branch logic). All physical parts were made to be modules in order to resemble hardware as best as possible. The assembled machine code was read into the processor through the Instruction Memory module, and was sent through the databus due to the address specified by the program counter. This was done to again resemble hardware as possible, as Instruction Memory in a simple buildable processor, given Harvard Architecture, would be implemented as a ROM chip, hence one would not be able to dynamically access it. The components/modules were chained appropriately in "test_bench.v" where a clock was instantiated and fed to the appropriate components. Following are the verilog module and which datapath elements they represent:

Filename (*.v)	Corresponding Datapath Element
ALU	ALU
Ctrl	Control
DM	Data Memory
MM	Main Memory
IM	Instruction Memory
PC	Program Counter
REG	Register File
STwoToOne_1	Two to One Selector (1 bit data)
STwoToOne_8	Two to One Selector (8 bit data)
SignExtend	Sign Extender
ZeroExtend_2	3 bit zero extender
ZeroExtend_3	4 bit zero extender
getNextPC	Branch/Jump logic to control PC
lireg	Load Immediate Register

Assembler:

The assembler was written in C++. We have one file (fileparser.cpp) that has all the parsing functions. The file assembler.cpp is the main program that invokes the functions in fileparser.cpp to assemble a program to machine code. The assembler utilizes three passes to convert the assembly code to machine code. On the first pass, it stores data and indexes all the labels used. On the second pass, it replaces labels on jump and branch-if-equal commands with the appropriate binary values. The third passes actually converts the instructions to machine code. The assembler could have been done with two passes by combining the second and third pass. However, the programs are very small (maximum 256 lines) so the efficiency upgrade would be negligible. The extra complexity of combining the second and third pass was not worth it.

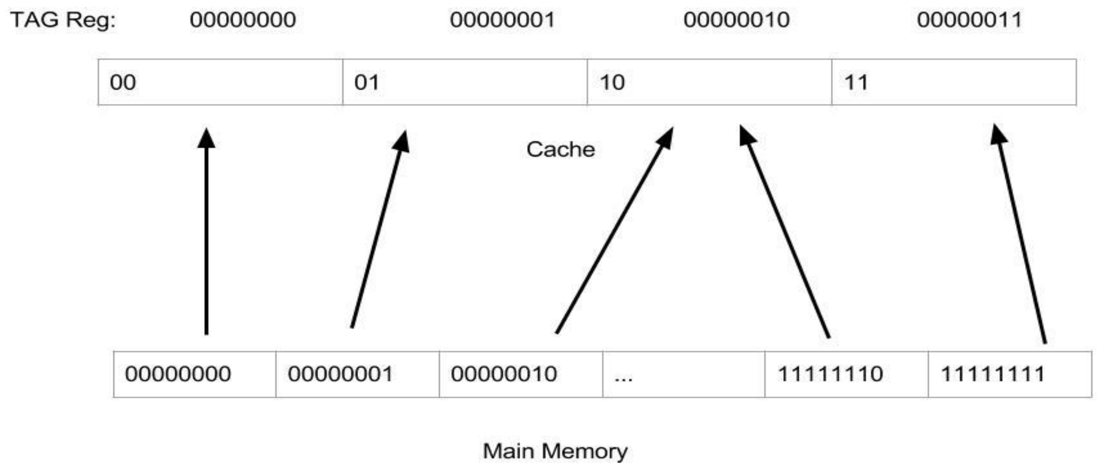
Programs (Fibonacci + Multiplication):

In our programs folder we have fibonacci.asm and mul.asm. Fibonacci is a non-recursive implementation which demonstrates a nested procedure in our code. It seeds the first two numbers as 0 & 1. It finds the nth value after 0 & 1 defined as num1 in the .data section. At the end, register 0 will store the value. Mul is a recursive multiplication program. It finds the product of num1 and num2 in .data section and stores answer in register 0 at end.

Programs are stored in the program/ directory. The assembler will prompt the user for the name of the assembly program file and it will output the machine code in a file named "output.bin" which is also stored in the /program directory. The processor verilog code will then read the machine code from "output.bin" and output the time, program counter, instruction, register values, and any written data to standard output. After you have written a program open a terminal in the root directory and run the command “make run”. This will compile/run the assembler and immediately compile/run the processor.

4. EXTRA CREDIT

Caching:



Cache and Memory Map
Memory Address 2 LSB = Cache Address
Tag = Memory Address. Saved In Tag Reg to specify data.

One of the major pros of Harvard architecture is the ability to physically implement Instruction Memory and Data Memory in different manners. Hence, we can model Instruction Memory as cheap ROM and model Data Memory as expensive RAM, as Data Memory is the one which will be dynamically accessed. Hence, it only makes sense to cache Data Memory.

In our processor implementation, cache was appealing because we do not have a register for the stack pointer. The stack pointer is stored in memory. The issue with this is speed, so cache will significantly speed up our procedures. Cache's function on the principles of spatial and temporal locality. Spatial locality is the concept that if a piece of data is taken from memory, there is a good chance that something in a nearby address will also be called soon. This is especially true if memory is categorized, such as in the example of a library with alphabetized shelves. Temporal locality is the idea that something called recently will be called again soon. This piece of data is good for the cache. In our implementation of cache, temporal locality is demonstrated. Spacial locality is not taken into account because main memory can cache in this implementation are not nearly as large as in computers where spatial locality is necessary.

Cache and memory can be connected in different ways. They must be mapped, otherwise another address will need to be added to the instruction set, specifically “cache address” field. In this implementation, a direct mapping was done using the last two bits of the memory address as the cache address. The cache, therefore, is 4 bytes. This size of cache meets the proportions of cache size to main memory size for common computers. In order to distinguish different pieces of data with the same final bits, a tag is used. This tag is simply a label that is stored in a tag register which is comprised of the same bits of the memory address. This allowed for many problems involving writing back to be solved when the current data address is different from one stored in the cache.

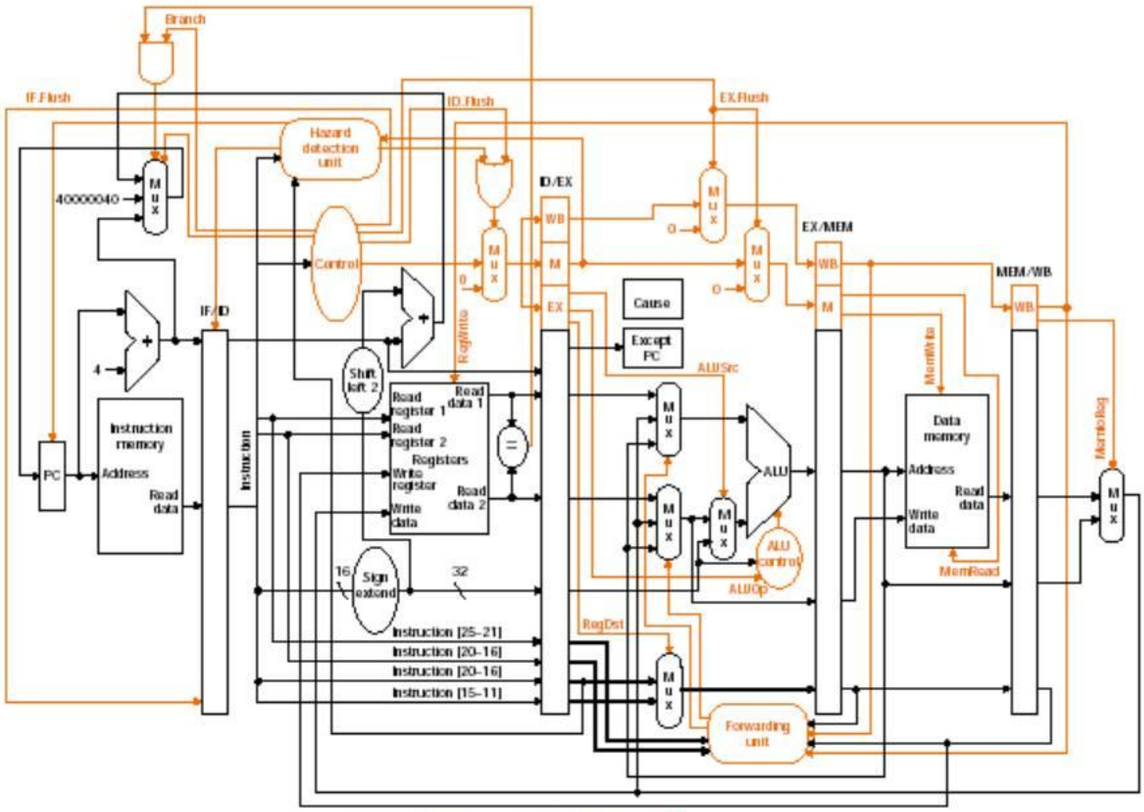
The reason a set associative or fully associative mapping was avoided is their complexity and lack of practicality for a small basic implementation of a computer. These types of cache mappings have benefits in larger scale memory hierarchies.

When it comes to save word and store word instructions, the data path always leads to the cache for data transfer. The conditionals are whether the instruction is a read or a write, and if there is particular data in a particular part of the cache. For writing, the data is always delivered to the address in cache that corresponds to the address in main memory, assuming there is not data there already. If there is data in the address, the cache performs a write back. The data at the address is written to main memory and the new data overwrites the old in the cache. If the cache address was empty, the data will simply be stored there. Write back was chosen in oppose to write through because it is faster not to write to memory every time. In addition, write back was chosen over a writing buffer because it is more direct and involved less components.

For reading, the address in cache that corresponds to the address in memory is checked for the data. First, the cache address is checked to see if there is data present. Then, the tag is checked for a match with the main memory address. If there is a match, this is a hit, and the memory is taken from cache. If however there is a miss, and the cache either had no data present, or the wrong data, then the data is taken from the address in main memory and is placed to the corresponding address in cache. It is then taken from cache and is used in the procedure. One of the more difficult cases to cover is when there is something in the cache, but it is the wrong tag. This item must first be stored into main memory before an item can be placed in the cache address in order to prevent data from being lost in memory. This can cause race conditions if proper timing is not accounted for.

In this Verilog implementation, everything happens instantaneously, so there is no actual time benefit for the procedure. However the function of cache is demonstrated by printing hits and misses of data in the cache. There will be outputs related to the cache on several occasions. For loading words from cache, hits will display, “CACHE HIT,” and misses will first display “CACHE MISS. Waiting for main memory,” then after a pause say, “Continue,” when the main memory responds. For storing words, storing in an empty cache has no output. Storing data with an address identical to another piece of data in cache or main memory leads to an overwrite, which displays “OVERWRITTEN.” Storing data that goes in the same cache address but of a different tag will trigger a write back, which displays, “WRITING BACK.” The hit rate of our cache when applying it to a recursive multiplication program of 5X4 was 85.6%, and when applying it to a fibonacci program of index 7 was 83.9%. These rates indicate an effective cache system and would lead to significant processor speed ups.

Pipelining:



Datapath for Pipelined Processor

The image above is a picture of the MIPS Pipeline. This is a 5-stage pipeline which utilizes both Forwarding (Forwarding Device) as well as Stalling (Hazard Detection Device). This datapath was tailored to our architecture. The stages are Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB), just as in MIPS. We needed to implement pipeline registers in order to implement the stages, where we simply use nonblocking assignments in order to implement the pipeline registers. Furthermore, Control needed to be updated in order to cater to the Pipeline. Lastly, the Forwarding Unit and Hazard Detection Unit needed to be created where depending on the values in the Pipelining registers, the devices activate.

The pipelining modules function when given machine code on their own, however the chaining together induces problems. The error is certainly within chaining together the values appropriately, utilizing which pipeline register value where, as the stages were not defined in its entirety.

Programming into PLD:

This extra-credit was not done for the following reason. In order to program Verilog into a FPGA and demonstrate the processor functionality on a breadboard, like our group did for the Paper Processor, the behavioral verilog must be synthesizable. This requires a plethora of limitations on the behavioral code, causing many upon many changes for functionality. Then the synthesis tool for the proper FPGA needs to be found. Synthesis needs to be done to obtain the appropriate netlist. Then "placing and routing" needs to occur to map to the physical parts on the device appropriately. Alternately the chipmaster could be used, but this option was not explored.

We wanted to focus upon the extra-credits which seemed most pertinent to the course, such as Pipelining and Caching. This extra-credit would require very much research to be done in the material taught in a course such as Digital VLSI, and didn't seem desirable for optimal time allocation.

Compiler:

This extra-credit was not done for the following reason: when writing the assembler for our ISA, we saw how difficult it is to take coded language and break it down into its machine code equivalents, while taking into account whitespace, comments, .text and .data sections, etc. To write a compiler for fibonacci or recursive multiplication written in C, the complexity of the code needed to compile down to appropriate assembly in our ISA increases immensely. We could write a very simple compiler which handles very few instructions, but then we are not satisfying what a compiler should yield, the ability to write higher-level code for our processor, which is a task very difficult to meet.

We wanted to focus upon the extra-credits which seemed most pertinent to the course, such as Pipelining and Caching. This extra-credit would require very much research to be done in the material taught in a course such as Compilers, and didn't seem desirable for optimal time allocation.

5. CONCLUSION

Ssby architecture is a Harvard based RISC processor. The advantages of our processor are that it is RISC based, it is a Harvard architecture, and it utilizes a cache. RISC architecture places emphasis on software as opposed to CISC architecture which emphasizes hardware. RISC is advantageous since it is much easier to program. Harvard architecture is advantageous because it allows us to use different types of memory (ROM vs RAM) which will save costs and optimize the hardware implementation. Furthermore, the implementation is much simpler for caching data memory. Caching is advantageous because it allows for quick memory accessing. This makes having only two general purpose registers a nonissue. As memory access has been optimized, the presumed issue with a lack of a stack pointer register is negligible, hence justifying our design choices.

A pipelined datapath was nearly implemented. Also, research was done to consider the possibility of a compiler being made and creating a physical circuit for our processor. In the end, creating the ssby architecture was a valuable but difficult experience. We appreciate being assigned such a rewarding project.