The American University in Cairo

FemtoRV32ic

Project 1 – Computer Architecture

Hunter CPU: Milestone 3 report

Habiba Gamal	*900151007*
Ali El-Said	*900150264*
Ahmed Wael Fahmy	*900160127*

Introduction

In this project, we were required to design and implement a pipelined CPU that supports all the RISCV32IC instruction set, with the exception of some instructions such as FENCE and CSR. Up till this milestone (Milestone 3), All RISC-V32IC instructions are supported. The main difference between this milestone and the previous one is the added support for compressed instructions. We were required to come up with a Datapath block diagram, as well as implement the CPU in Verilog RTL. Some restrictions were imposed on our design, specifically that there would be a single memory for both data and instructions, unlike the familiar design of separate data and instruction "caches" in the Datapath. This memory also had to be single ported, unlike the register rile, which had to be dual ported. We were provided with a recommended pipeline implementation; with a 3-stage pipeline. The first stage included fetching and decoding, the second executing and memory, while the final stage included writing back. This design reduced the potential hazards as it avoids having 2 conflicting stages in the same clock cycle.

This milestone, we started off by understanding how the compressed instructions functioned. The main objective behind compressed instructions is reducing space when the instruction can be encoded in only 16-bits. We added a decompression unit that transformed compressed instruction into their equivalent

32-bit instructions. We also added a Multiplexor that chose between the output of the normal Instruction register and the decompression unit. This was mainly controlled by the least significant two bits of a given instruction (only “11” is a 32-bit word opcode, any other combination is a 16-bit instruction opcode). An adder that increments by 2 was also required in case of compressed instructions, and a multiplexor selecting between the two PC increments was added, controlled by the same signal discussed previously. Figure (1) shows this the updated Datapath diagram.

Updated Datapath Block Diagram

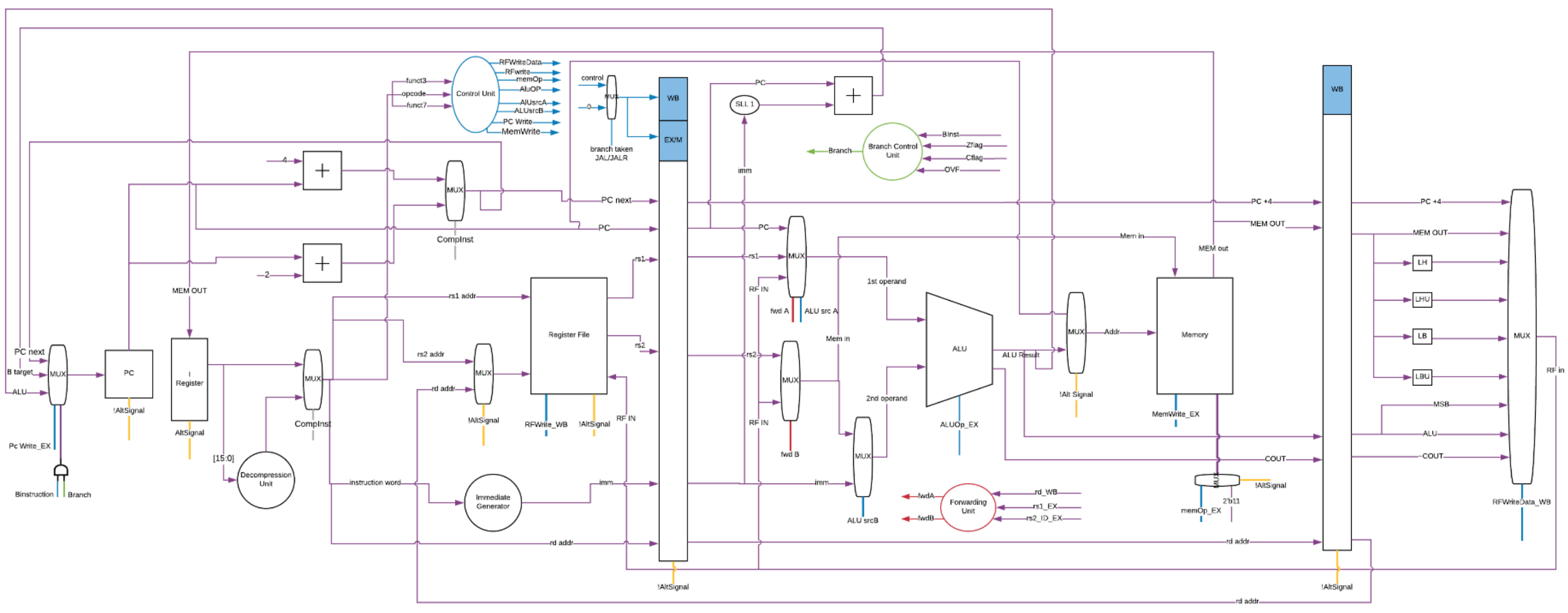


Figure (1)

This final Datapath block diagram is what we were able to create after incrementally finding issues with, and fixing mistakes in our initial Datapath designs. This is the final Datapath for milestone 3, which supports compressed instructions as previously discussed. The Datapath is controlled by three discrete modules: The control unit, the branching unit, and the forwarding unit. The control unit receives the opcode and the function codes and produces most of the control signals; some of which are passed along in the pipeline registers. The branching unit is responsible for determining whether a branch should be taken or not. The forwarding unit handles dependency cases by forwarding the needed values to the locations where they are needed. This was made simpler by only having two pipeline registers, unlike the previous five-stage design which included four registers.

We saw no need to add extra pipeline registers for the two sub-stages, as this would have little to no benefits. Handling the single memory proved to be one of the projects difficulties, which we resolved by having an alternating signal that acted as a control for many modules, effectively switching the memory from one role to another. Another addition was that of the Instruction register, which holds the output from the memory to be later distributed to the Immediate generator, the control unit, the pipeline register, and the register file. To handle branch instructions, a multiplexer was needed to select the appropriate input to the program counter depending on both the control unit and branching unit signals. More instructions meant that the write back multiplexer was much larger, as we also opted to extend the values of the load instructions in parallel.

The only changes to the Datapath made this milestone were to support compressed instructions, as the RV32i instructions were functioning correctly.

Memory Design

This memory design proved helpful when supporting compressed instructions as it allowed reading unaligned words from the memory. This was crucial as compressed instructions are often half word aligned. Handling the store instructions, with the alignment issues of RISC V proved to be challenging, as we read in the official RISC V instruction manual that unaligned reading and writing was supported, although not recommended. To address this challenge, we broke down the memory into four banks of four below is a diagram to help explain our memory design. Note that in the case of unaligned words and half-words (both reading and writing), In_adder is zero if the byte will be read from the input address, and will be one if the byte is read from the following word (next address). Multiplexers allocate bytes into their correct banks, controlled by selection lines coming from the control unit, that depend on the instruction (SW, SH, SB) and the least significant bits of the address. How the output is concatenated also depends on the two least significant bits of the address.

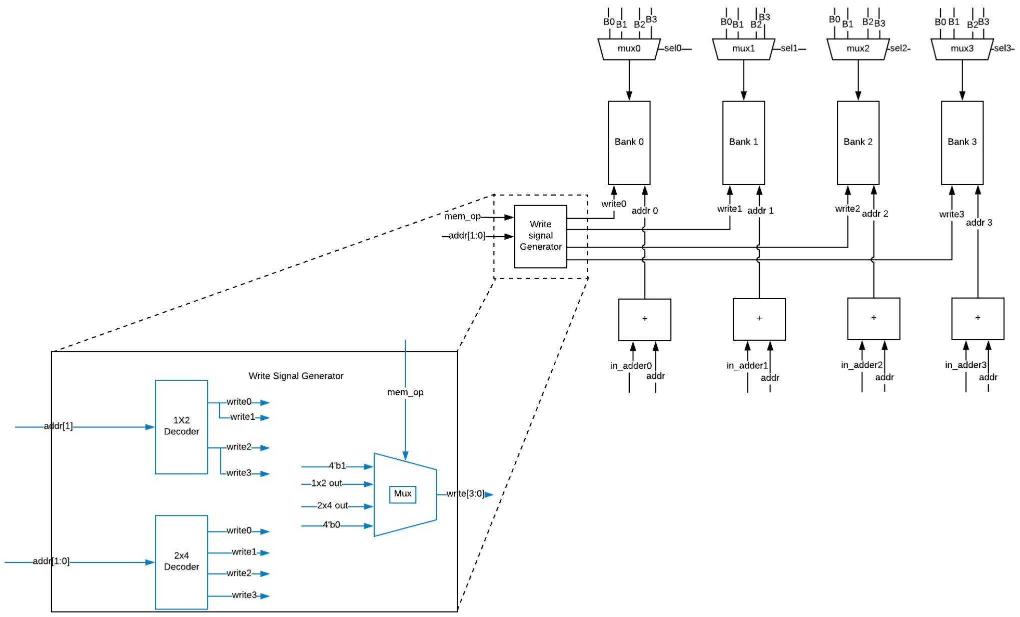
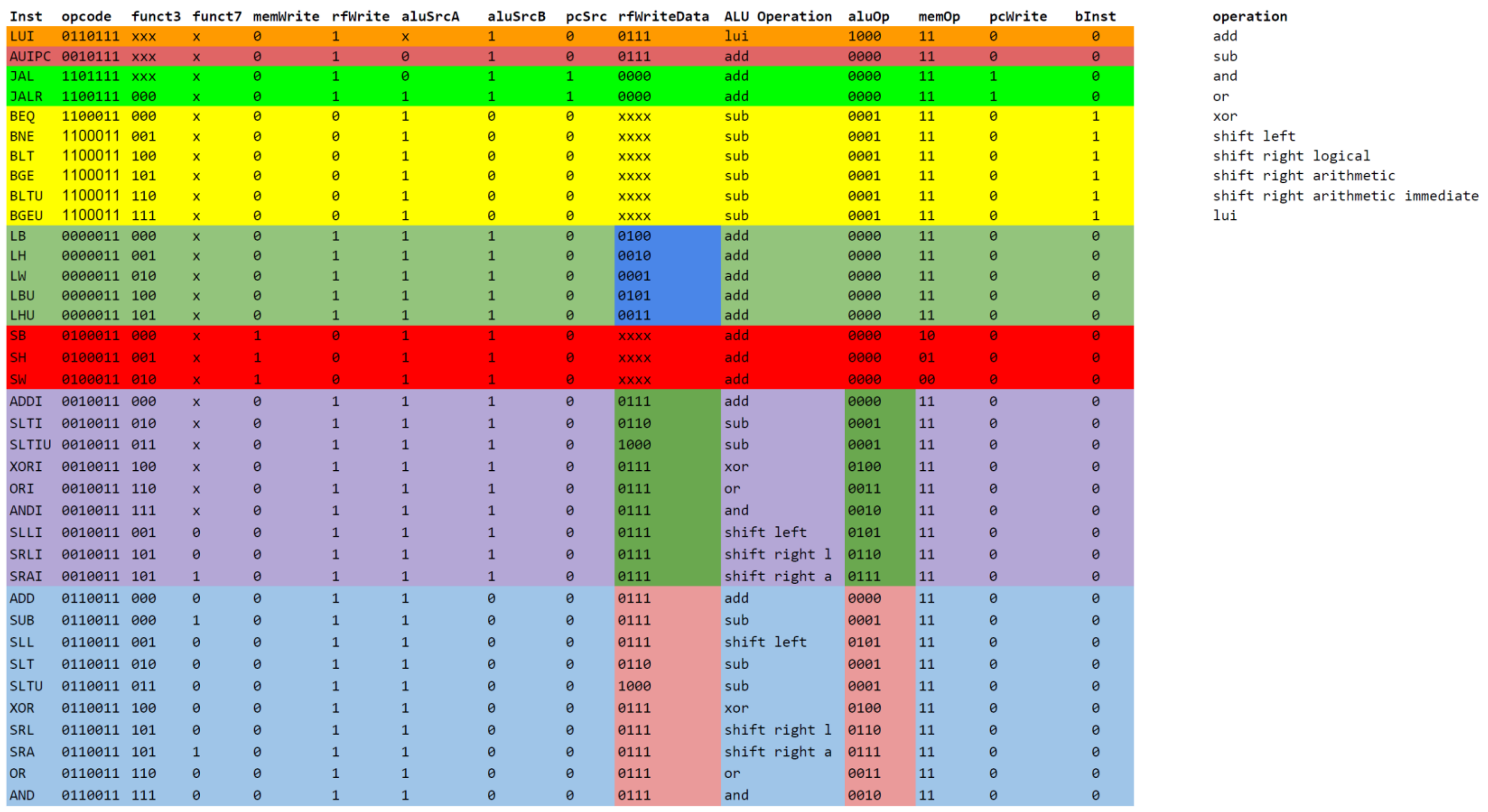


Figure (2)

Control Unit Analysis

As previously discussed, the control unit was initially designed, then iteratively changed as issues came to surface. This was our final implementation of the control unit signals. Some essential signals such as aluSrc were clear, but others such as memOp were added as we tinkered with our Datapath and module design. Note that the alternating signal is independent of the control unit. No additions to the CU were needed to support compressed instructions.



Inst	opcode	funct3	funct7	memWrite	rfWrite	aluSrcA	aluSrcB	pcSrc	rfWriteData	ALU Operation	aluOp	memOp	pcWrite	bInst
LUI	0110111	xxx	x	0	1	x	1	0	0111	lui	1000	11	0	0
AUIPC	0010111	xxx	x	0	1	0	1	0	0111	add	0000	11	0	0
JAL	1101111	xxx	x	0	1	0	1	1	0000	add	0000	11	1	0
JALR	1100111	000	x	0	1	1	1	1	0000	add	0000	11	1	0
BEQ	1100011	000	x	0	0	1	0	0	xxxx	sub	0001	11	0	1
BNE	1100011	001	x	0	0	1	0	0	xxxx	sub	0001	11	0	1
BLT	1100011	100	x	0	0	1	0	0	xxxx	sub	0001	11	0	1
BGE	1100011	101	x	0	0	1	0	0	xxxx	sub	0001	11	0	1
BLTU	1100011	110	x	0	0	1	0	0	xxxx	sub	0001	11	0	1
BGEU	1100011	111	x	0	0	1	0	0	xxxx	sub	0001	11	0	1
LB	0000011	000	x	0	1	1	1	0	0100	add	0000	11	0	0
LH	0000011	001	x	0	1	1	1	0	0010	add	0000	11	0	0
LW	0000011	010	x	0	1	1	1	0	0001	add	0000	11	0	0
LBU	0000011	100	x	0	1	1	1	0	0101	add	0000	11	0	0
LHU	0000011	101	x	0	1	1	1	0	0011	add	0000	11	0	0
SB	0100011	000	x	1	0	1	1	0	xxxx	add	0000	10	0	0
SH	0100011	001	x	1	0	1	1	0	xxxx	add	0000	01	0	0
SW	0100011	010	x	1	0	1	1	0	xxxx	add	0000	00	0	0
ADDI	0010011	000	x	0	1	1	1	0	0111	add	0000	11	0	0
SLTI	0010011	010	x	0	1	1	1	0	0110	sub	0001	11	0	0
SLTIU	0010011	011	x	0	1	1	1	0	1000	sub	0001	11	0	0
XORI	0010011	100	x	0	1	1	1	0	0111	xor	0100	11	0	0
ORI	0010011	110	x	0	1	1	1	0	0111	or	0011	11	0	0
ANDI	0010011	111	x	0	1	1	1	0	0111	and	0010	11	0	0
SLLI	0010011	001	0	0	1	1	1	0	0111	shift left	0101	11	0	0
SRLI	0010011	101	0	0	1	1	1	0	0111	shift right l	0110	11	0	0
SRAI	0010011	101	1	0	1	1	1	0	0111	shift right a	0111	11	0	0
ADD	0110011	000	0	0	1	1	0	0	0111	add	0000	11	0	0
SUB	0110011	000	1	0	1	1	0	0	0111	sub	0001	11	0	0
SLL	0110011	001	0	0	1	1	0	0	0111	shift left	0101	11	0	0
SLT	0110011	010	0	0	1	1	0	0	0110	sub	0001	11	0	0
SLTU	0110011	011	0	0	1	1	0	0	1000	sub	0001	11	0	0
XOR	0110011	100	0	0	1	1	0	0	0111	xor	0100	11	0	0
SRL	0110011	101	0	0	1	1	0	0	0111	shift right l	0110	11	0	0
SRA	0110011	101	1	0	1	1	0	0	0111	shift right a	0111	11	0	0
OR	0110011	110	0	0	1	1	0	0	0111	or	0011	11	0	0
AND	0110011	111	0	0	1	1	0	0	0111	and	0010	11	0	0

operation
add
sub
and
or
xor
shift left
shift right logical
shift right arithmetic
shift right arithmetic immediate
lui

Figure (3)

Limitations

Final milestone functions such as EBREAK and ECALL were not supported as they are not yet required in this milestone. we were not able to find any limitations or issues with our final Datapath block diagram or out Verilog HDL code through our testing.