CSE 141L

Milestone 4 - DLB processor design demo

Architectural Highlights

- Single cycle machine
- 6 general purpose registers, bit storage, and counter
- Supported operations: lw, sw, inc, luv, and, cpy, sb, gb, flip, xor, loop, shr, goto, beq, rb

Philosophy

Don’t Look Back!

Goal is to minimize the number of reads and writes in the program, especially reads and writes of the same data.

Machine type

General Purpose, load-store machine

Not all registers are created equal, though

Instruction format

3 types of instructions in the ISA: 2R-type, I-type, LUT-type

2R	OPCODE	rd	rs	h
1R	OPCODE	rd	off	h
I	OPCODE	rs	imm	
LUT	OPCODE	lut		

Instruction List

LW/SW - load and save, built in offset calculation

INC - increments counter

LUV - loads a value from LUT <3

AND - bitwise and of two registers

CPY - Copies contents of one register to another

SB - Sets a bit inside a register

RB - resets stored bit to 0

GB - Saves a bit from a register

FLIP - Flips a bit inside a register

XOR - a reductive XOR

LOOP - Helps with looping through the counter

LSH - a left shift

GOTO - jump around the program

BEQ - Branch if two regisiters are equal

ADD - Adds an immediate to the register

Branching logic

With only 9 bit in each instruction, how do we deal with branching logic?

Implemented in software with the support of two instructions: goto and beq.

- For each branch that is **taken**, it does nothing. As a programmer, we put a goto instruction right below that, pointing to an address wherever this branch needs to go.
- For each branch that is **not-taken**, it increments the pc such that the next instruction to be executed is the next after next. This way, the processor avoid executing the goto instruction.

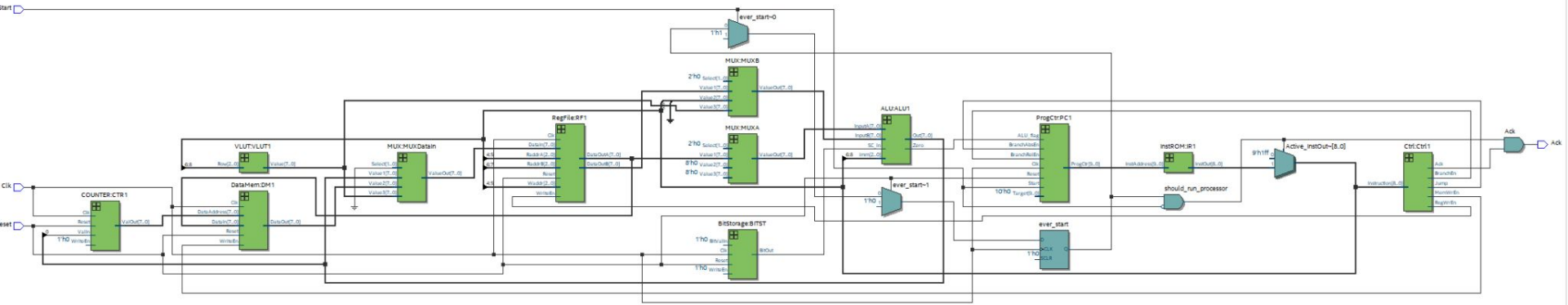
Branching Logic(cont’d)

- Addresses are calculated indirectly, by using the counter and one of the hardcoded offsets.

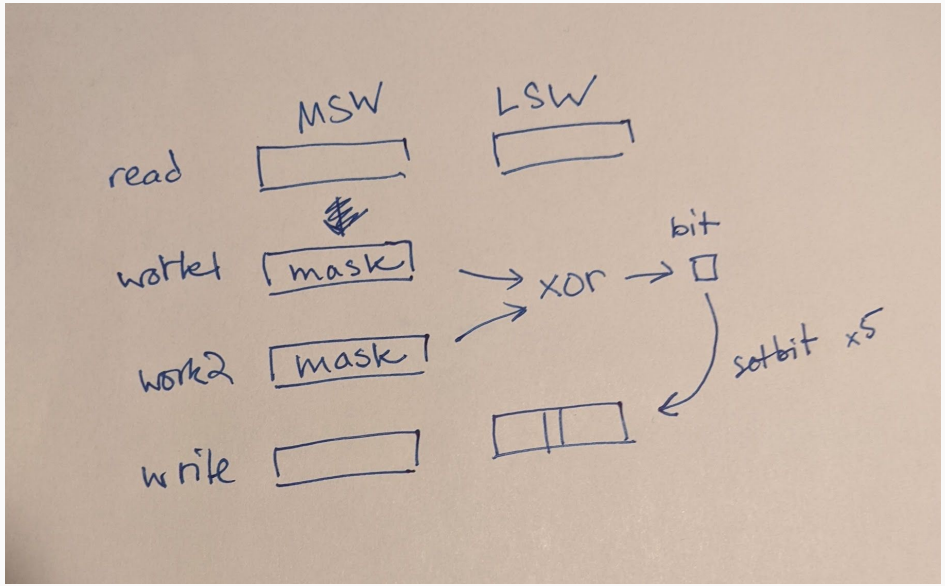
Programmer’s Model

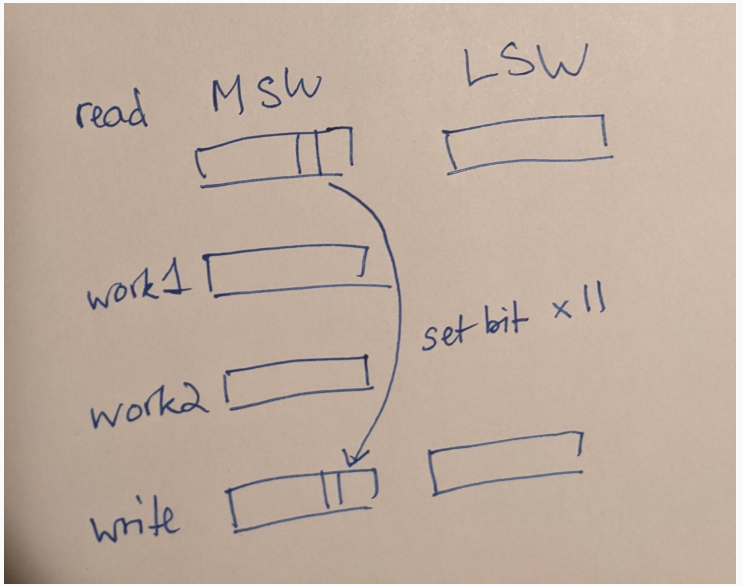
- Typical Load Store machine, but with some quirks!
- Not MIPS/ARM compatible (but similar!)
- ALU is used for some non-arithmetic instructions, such as copy and calculating equality of registers for branches.

Block diagram & datapath

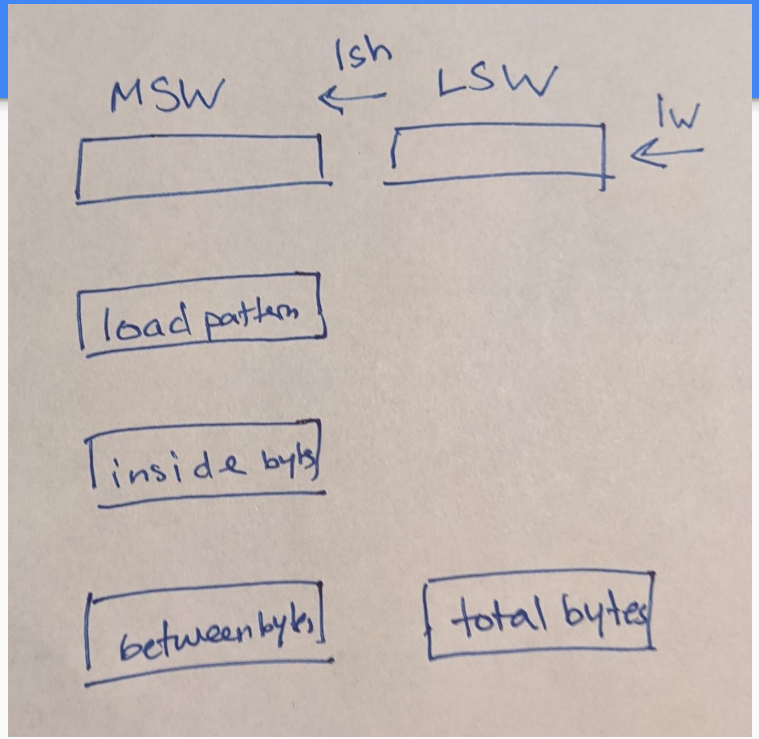


Program 1+2 Implementation





Program 3


MSW
lsh
LSW
lw
load pattern
inside byts
between byts
total bytes

Testbench - how it works?

The testbench gives multiple sets of input values, calls the testing function, compute the expected outputs in testbench, then compare the calculated outputs to the outputs produced by the functions.

- Top-level TB
- Instruction TB
- ALU TB
- ProgCtr TB

Testbench - executing result of our TB

The testbench executes well, except for a bug which causes the testbench to exit early. However, if you continue execution manually with run 1000, it works perfectly well, showcasing perfect performance for lw, sw, and cpy.

Assembler

The assembler is written in Python. It takes a file name, where the file containing an assembly program. It will then convert the assembly code to machine code that is valid for our processor, and last, generate a new file call [file name]-machine-code, where it writes all the machine codes.

It also recognizes comments and automatically ignores empty lines.

One special feature of the assembler is that, it recognizes the tags in the assembly language and can save the tag and address of that corresponding instruction into a table, so that when the tag is referenced in a instruction in the future, the assembler will automatically generate the address code.

Performance Evaluation

The team did not get the three programs to execute correctly, having been able to debug only a limited number of instructions.

However, the team is also very confident in the correctness of the pseudocode.

The team also believes that the greatest area of performance would be in time management. Despite turning in Milestone 1 in a timely manner, we fell off the bandwagon later.

Thank you