FEMTO-MC: Finicky, Elementary Machine TO MicroCompute

McCoy R. Becker

Friday, April 24, 2020

FEMTO-MC

A 16 bit microcomputer for the 21st century!

Overview:

1. 16 bit architecture with 15 GPRs.
2. Word size is 16 bits.
3. Single *mem* instruction can access memory to load and store. This instruction allows loading and storing of individual bytes.
4. PC is not accessible as a GPR.
5. Special feature: barrel shifter, and explicit instructions which take advantage of it.

Special (software) bonuses:

1. A nice, friendly emulator!
2. Assembler builds a parse tree - debugging not complete, but much potential!

Instruction set

Figure 3: 2 register RR-type instructions ($X_3X_2X_1X_0$ is specified by a constant $immed$)

instruction	4 bits	4 bits	4 bits	4 bits
ADD	1000	reg A	reg B	$X_3X_2X_1X_0$
LOG	1001	reg A	reg B	$X_3X_2X_1X_0$
MEM	0001	reg A	reg B	$X_3X_2X_1X_0$

Figure 4: 2 register RRI-type instructions

instruction	4 bits	4 bits	4 bits	4 bits
BEQ	0010	reg A	reg B	immed
JUMP	0011	reg A	reg B	unused
SHIFTL	0100	reg A	reg B	immed
SHIFTR	0101	reg A	reg B	immed

Figure 5: Single register RI-type instructions

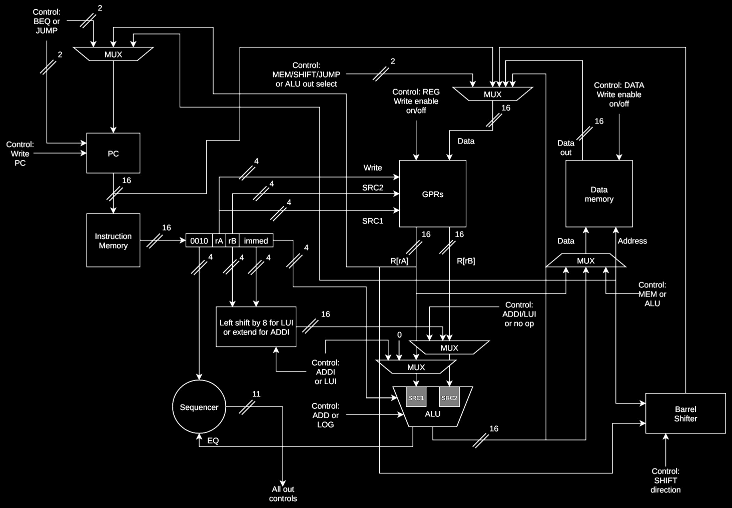
instruction	4 bits	4 bits	8 bits
LUI	0000	reg A	immed
ADDI	1010	reg A	immed

More specifics

The architecture possesses the following abilities:

1. Subroutine call and return (via a *jump* instruction which stores the current PC).
2. Ability to load immediate values (via *lui* and *addi* instructions). Combined, these instructions allow addressing of up to 64 kB of word addressable memory (128 kB byte addressable).
3. Ability to set and test bits with a *log* instruction.
4. Ability to conditionally branch with a *beq* instruction.
5. Ability to logically bit-wise complement a value (multiple ways, through function bits on an *add* instructions or directly with function bits on the *log* instruction).
6. Immediates are zero extended.

Block diagram



Despite what diagrams indicates - instruction memory and data memory are the same!

Implementation specifics

1. All paths in the dataflow are used by instructions.
2. PC is a separate adder from the ALU.
3. Barrel shifter is designed into the implementation architecture - used by *shiftr* and *shiftl* instructions.

Nice to haves:

1. ALU constantly tests for equality, used by *beq* path.
2. Control switches allow output from ALU to write directly to register array for the correct instructions, no need to go through memory (in some weird way).

Special feature

A 16-bit barrel shifter - allows arbitrary shifts in either direction.

1. Follows the scheme for logarithmic-time multiple bit shifting outlined in the *Shifters* slides.
2. 2nd operand to *shiftr* and *shiftl* determines shift magnitude. This can be used to iteratively shift the first operand by accumulating values in the 2nd operand.
3. Makes multiplication a snap! (as we will see later)

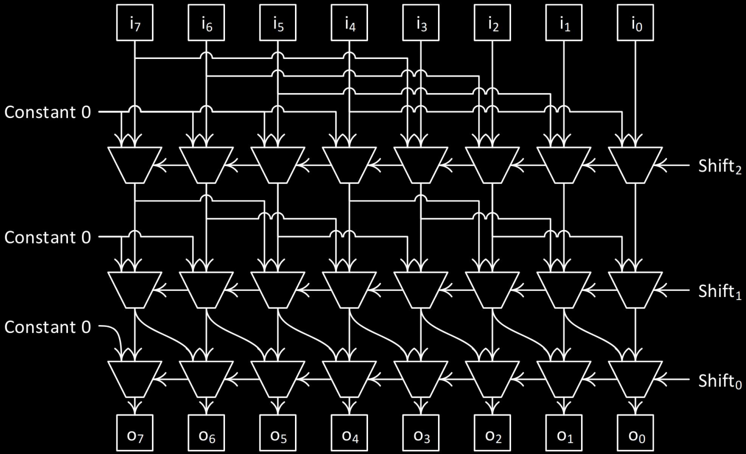


Figure 1: One half of a barrel shifter - from Dr. Frankel's slides.

Clocking scheme

This is standard (I expect):

1. Transitioning the sequencer FSM is performed on the falling edge.
2. Dataflow propagation occurs when the clock is low.
3. Destructive updates (i.e. writing to the register array) is performed on the rising edge.
4. When the clock is high, nothing happens - the FSM waits until the falling edge to transition to the next state.

On to the assembler!