CS 51 Homework 3

Jessie Li

October 7, 2024

1.

The design of my 16-byte RAM is based on the 8-nibble RAM in `ram.circ` from the Fun with Flip Flops lecture. It takes three inputs, `data in` (1 byte wide), `address` (4 bits wide) and a `write` clock (rising-edge triggered), and outputs `data out`.

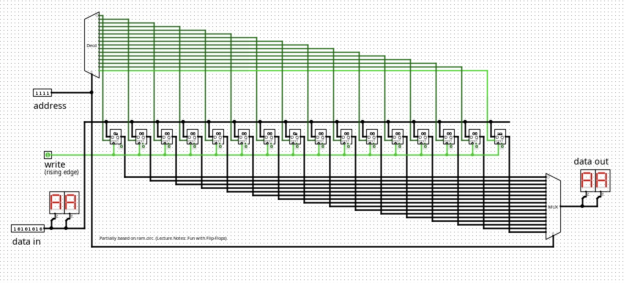

Figure 1: Writing `0xaa` to address `0x0`.

Testing

I manually verified that my circuit demonstrates the following functionalities:

1. Stores `data in` at the input `address` on the rising edge of the `write` clock.
2. `data out` matches the value stored at `address`.

Here's an example of a sequence I tested:

`address`	`data in`	`write`	`data out`	description
0x0	0x01	0	0x00	set `data in` to 0x01 while clock is deasserted
0x0	0x01	1	0x01	write 0x01 to 0x0 on clock rise and show in `data out`
0x0	0x02	1	0x01	changing `data in` while clock asserted should not update value at 0x0
0x0	0x02	0	0x01	deassert clock
0x0	0x02	1	0x02	update data at 0x0 on clock rise to 0x02
0x0	0x02	0	0x02	deassert clock
0xf	0x02	0	0x00	change `address` to 0xf, `data out` should be 0x00
0xf	0xaa	0	0x00	set `data in` to 0xaa
0xf	0xaa	1	0xaa	write 0xaa to 0xf
0x0	0xaa	1	0x02	check that 0x02 is still stored at 0x0
0xf	0xaa	1	0xaa	check that 0xaa is still stored at 0xf

2.

For Q2, I created a 16 slot × 8 bit block of RAM at addresses `0xY0` through `0xYf` for a given constant `Y`. I used a buffer to ensure that `data out` floats if the most significant nibble `MSN` of the address is not equal to `Y`.

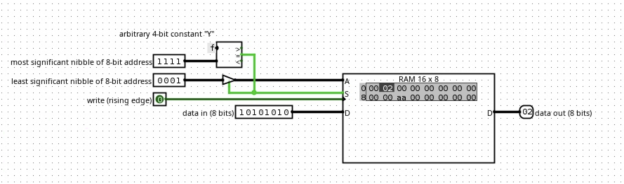

Figure 2: RAM chunk for addresses ranging from `0xf0` to `0xff`.

Testing

Manual testing confirms the following functionality:

1. `data out` shows the value stored in RAM at the input address when the `MSN` of the address is equal to `Y`, floating otherwise.
2. `data in` is stored only when the clock rises and the `MSN` = `Y`.

One sequence I tested with constant `Y` = `0xf`:

`MSN`	`LSN`	`data in`	`write`	`data out`	description
0x0	0x0	0x01	0	float	change `data in` while clock deasserted
0x0	0x0	0x01	1	float	write with `MSN` ≠ `Y`
0x0	0x1	0x01	1	float	set `LSN` to 0x1
0xf	0x1	0x01	1	0x00	`MSN` = `Y`
0xf	0x1	0x01	1	0x00	set `data in` to 0x02
0xf	0x1	0x01	0	0x00	deassert clock
0xf	0x1	0x02	1	0x02	write `MSN` = `Y`
0xf	0xa	0x02	1	0x00	set `LSN` = 0xa
0xf	0xa	0xaa	1	0x00	set `data in` to 0xaa
0xf	0xa	0xaa	0	0x00	deassert clock
0xf	0xa	0xaa	0	0xaa	write 0xaa with `MSN` = `Y`
0xe	0xa	0xaa	1	float	set `MSN` ≠ `Y`
0xf	0xa	0xaa	1	0xaa	check that 0xaa is still stored at `0xfa`
0xf	0x1	0xaa	1	0x02	check that 0x02 is still stored at `0xf1`

3.

For Q3, I completed the skeletons for `regwriter` and `regreader` to allow:

1. Writing to up to two different registers simultaneously on the rising edge on the clock. **If `reqE` and `reqM` are both asserted and `dstE` == `dstM`, `valM` is loaded into the register.**
2. Reading two different registers simultaneously, independent of the clock.

Here's a preview of my designs:

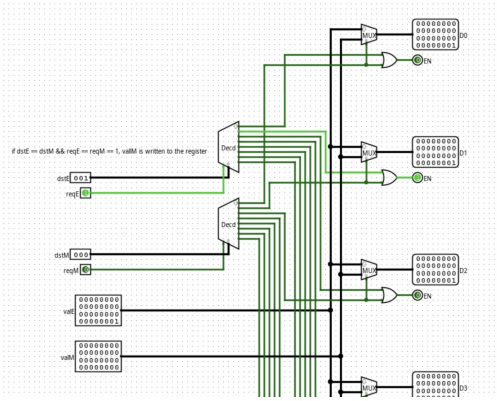

Figure 3: `regwriter`. Decoders determine which register to write to based on `dstE` or `dstM`. `valE` and `valM` are accessible to each output via a MUX, which outputs `valE` if the selection bit (the corresponding output line of `M`'s decoder) is 0, otherwise `valM`. At least one of `reqE` or `reqM` must be asserted for writing to be enabled for a specific register.

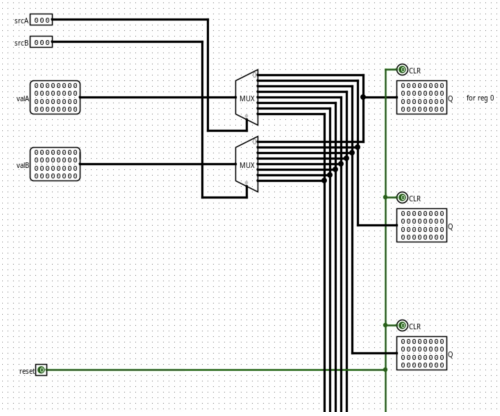

Figure 4: `regreader`. `srcA` and `srcB` select which registers to read and show the values in `valA` and `valB`.

Testing

To verify correct functionality, these were some of the test cases I covered:

1. **Write to two different registers.** Assert `reqE` and `reqM` with `dstE` $\neq$ `dstM`.

   Expect `valE` to be loaded into the register at `dstE` and `valM` to be loaded into the register at `dstM`.

2. **Write to the same register.** Assert `reqE` and `reqM` with `dstE` = `dstM`.

   Expect `valM` to be loaded into the register at `dstM` = `dstE`.

3. **Overwrite a previously stored register value.** Pick a register with a previously loaded value for `dstE` and load a new value `valE` into it.

   Expect `valE` to replace whatever value was previously stored in the register at `dstE`.

4. **No write to `dstM` when `reqM` is not asserted.** Similarly when `reqE` is not asserted.

   Expect the value of the register at `dstM` to stay constant on the rising edge of the clock if `reqM` = 0 and `dstE` $\neq$ `dstM` or `reqE` = 0. However, if `dstE` = `dstM` and `reqE` = 1, expect `valE` to be loaded into the register at `dstE` = `dstM`.

5. **Read from two different registers.** Set `srcA` $\neq$ `srcB`.

   Expect `valA` to show the value currently stored in the register at `srcA` and `valB` to show the value currently stored in the register at `srcB`.

6. **Read the same register.** Set `srcA` = `srcB`.

   Expect `valA` = `valB` = the value stored in the register at `srcA` = `srcB`.

7. **Clear registers when `clr` is asserted.** Assert `clr` when multiple registers have non-zero values.

   Expect all registers to be asynchronously cleared to zero. While `clr` is asserted, registers should be pinned to zero regardless of write attempts.

4.

Q4 asks us to design an ALU for a Y86 processor with 4 operations: `addl` (0x0), `subl` (0x1), `andl` (0x2), and `xorl` (0x3). The ALU takes as input a 4-bit operator `aluOP` and two 32-bit signed two's complement integers `aluA` and `aluB`, and outputs the arithmetic result of `aluB aluOP aluB` in `valE`.

I created two subcircuits to handle addition and subtraction: `addl` and `subl`. Both are responsible for setting the overflow flag `O`.

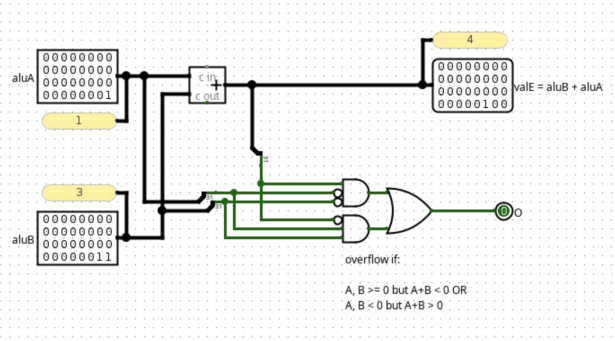

Figure 5: `addl` subcircuit. Overflow `O` is set to 1 if `A` and `B` have the same sign, but the result `valE` = `A` + `B` has the opposite. `O` is always zero when `A` and `B` have opposite signs.

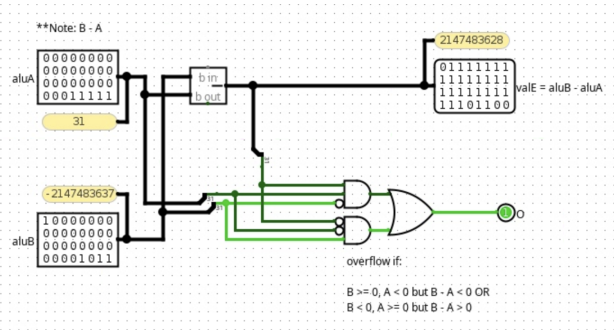

Figure 6: `subl` subcircuit. Note that this circuit computes `B - A`. Overflow `O` is set to 1 if `B` < 0 and `A` > 0 but `valE` = `B - A` > 0 or if `B` > 0 and `A` < 0 but `valE` < 0. `O` is always zero when `A` and `B` have the same sign.

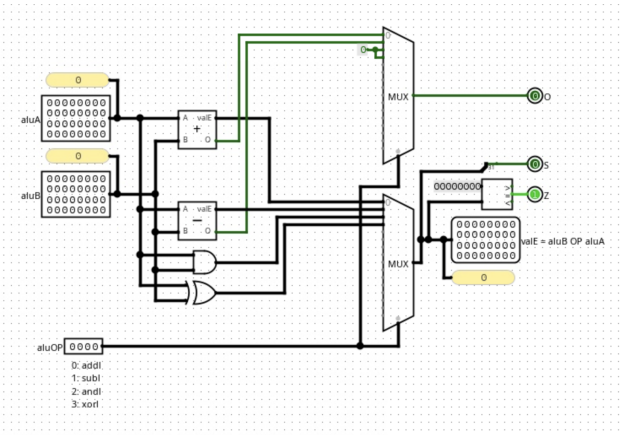

Figure 7: Main circuit with four operations: addition, subtraction, AND, and XOR.

Testing

Some of my test cases are listed on the next page.

aluA	aluB	aluOP	valE	Z	S	O	
0x00000000	0x00000000	0x0 (+)	0x0000000	1	0	0	add
0xffffffff	0x00000001	0x0 (+)	0x0000000	1	0	0	add, zero
0x00000001	0x70000000	0x0 (+)	0x7000001	0	0	0	add two +, no overflow
0xf0000000	0xa0000000	0x0 (+)	0x9000000	0	1	0	add two −, no overflow
0x00000001	0xa0000000	0x0 (+)	0xa000001	0	1	0	add +/−, no overflow
0x7fffffff	0x00000001	0x0 (+)	0x8000000	0	1	1	add, overflow max
0xffffffff	0x80000000	0x0 (+)	0x7fffffff	0	0	1	add, overflow min
0x00000000	0x00000000	0x1 (−)	0x0000000	1	0	0	sub
0x00000001	0x00000001	0x1 (−)	0x0000000	1	0	0	sub, zero
0x00000001	0x00000002	0x1 (−)	0x0000001	0	0	0	sub two +, no overflow
0xfffffffa	0xfffffffc	0x1 (−)	0x0000002	0	0	0	sub two −, no overflow
0xfffffffe	0x00000001	0x1 (−)	0x0000003	0	0	0	sub B > 0, A < 0, no overflow
0x00000001	0xfffffffe	0x1 (−)	0xfffffd	0	1	0	sub B < 0, A > 0, no overflow
0x80000001	0x00000001	0x1 (−)	0x8000000	0	1	1	sub, overflow max
0x00000001	0x80000000	0x1 (−)	0x7fffffff	0	0	1	sub, overflow min
0x80808080	0x10101010	0x2 (AND)	0x00000000	1	0	0	and, zero
0x00000001	0x0000000f	0x2 (AND)	0x00000001	0	0	0	and, nonzero
0x80000001	0x8000000f	0x2 (AND)	0x80000001	0	1	0	and, sign
0xffff0000	0xffff0000	0x3 (XOR)	0x00000000	1	0	0	xor, zero
0x00000001	0x00000002	0x3 (XOR)	0x00000003	0	0	0	xor, nonzero
0x40000000	0x80000000	0x3 (XOR)	0xc0000000	0	1	0	xor, sign
0x00000000	0x00000000	0x4 (?)	xxxxxxxx	x	x	x	unknown operator