CSE 141L
Spring, 2017

VERILOG MODULES

May 12, 2017

Author:	Chenxu Jiang
PID:	A92114155
Author:	Dingcheng Hu
PID:	A92090168

1 Introduction

In this lab, we build verilog modules for our CPU, in logic level. We build the following modules:

ALU supports addition/subtraction, 1-bit shifting, and branching decision.

Register File contains 3 sub-module, Accumulator Register Regular Register and Single-Bit Register.

Data Memory is 256 in depth and 8-bit in width, supports reads and writing to an memory address.

Instruction Fetch that contains Program Counter (its increment, branching and halting logic), Instruction Memory that stores read-only intruction memory and Look-up Table that stores branching address.

2 ISA Summarization

There are following types of operations supported:

Data Type:
Load from Memory Address to Register
Store from Register to Memory Address.
Copy Data in one Register to another Register.

Arithmetic Type:
Addition/Subtration of 2 Registers with/without Stored Overflow Bit.
Right Shifting Logically/Arithmetically/use stored Overflow Bit as shift-in.
Left Shifting that always use Overflow Bit as shift-in.

Miscellaneous:
Zero Specified Register.
Decrementing Specified Register by 1.
Add one Register 1 to Register 2 only if Register 3 is not negative.
Flip stored Flip-bit if specified register is negative.
Add 1 to Register 1 if the upper-4-bit of Register 2 and 3 matches.
Add flip bit xor flag bit to specified register.

Branching:
Branch if specified register is not 0.
Branch if upper-4-bit of specified register is not 0.
Branch if the register not equal to 31.

3 ALU Operations

ALU need to support all Arithmetic Type, Miscellaneous and Branching Instruction.

Arithmetic Type:
Addition/Subtration of 2 Registers with/without Stored Overflow Bit.
- `add1, add4, add5, addf0, addf4, sub1, sub3, sub4, sub5, subf2, sub25, add25, subf25`

Right Shifting Logically/Arithmetically/use stored Overflow Bit as shift-in.
- `srl, sra, srf, srl4, sra5, srf5`

Left Shifting that always use Overflow Bit as shift-in.
- `slf0, slf1, slf2`

Miscellaneous:
Zero Specified Register.
- `zero0, zero2, zero3, zero4`

Decrementing Specified Register by 1.
- `dec1`

Add one Register 1 to Register 2 only if Register 3 is not negative.
- `add3_n2, add4_n2`

Flip stored Flip-bit if specified register is negative.
- `atos_2`

Add 1 to Register 1 if the upper-4-bit of Register 2 and 3 matches.
- `add0c25`

Add flip bit xor flag bit to specified register. - `inc_af0`

Branching:

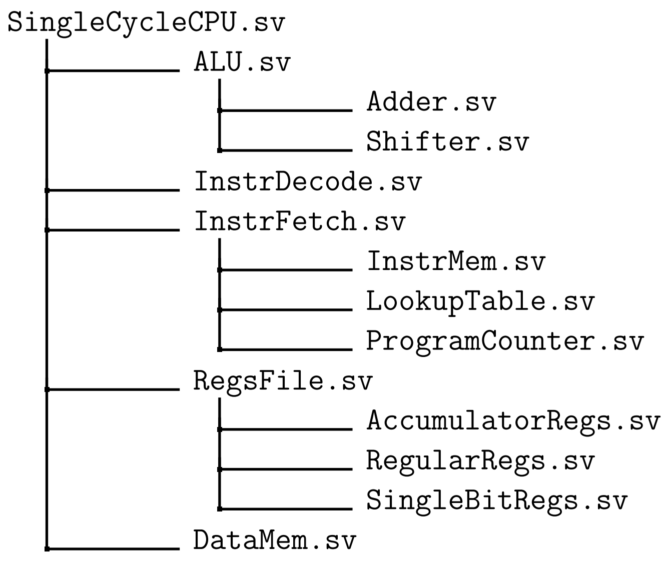
Branch if specified register is not 0.
- `bnzr`

Branch if upper-4-bit of specified register is not 0.
- `bnuzr`

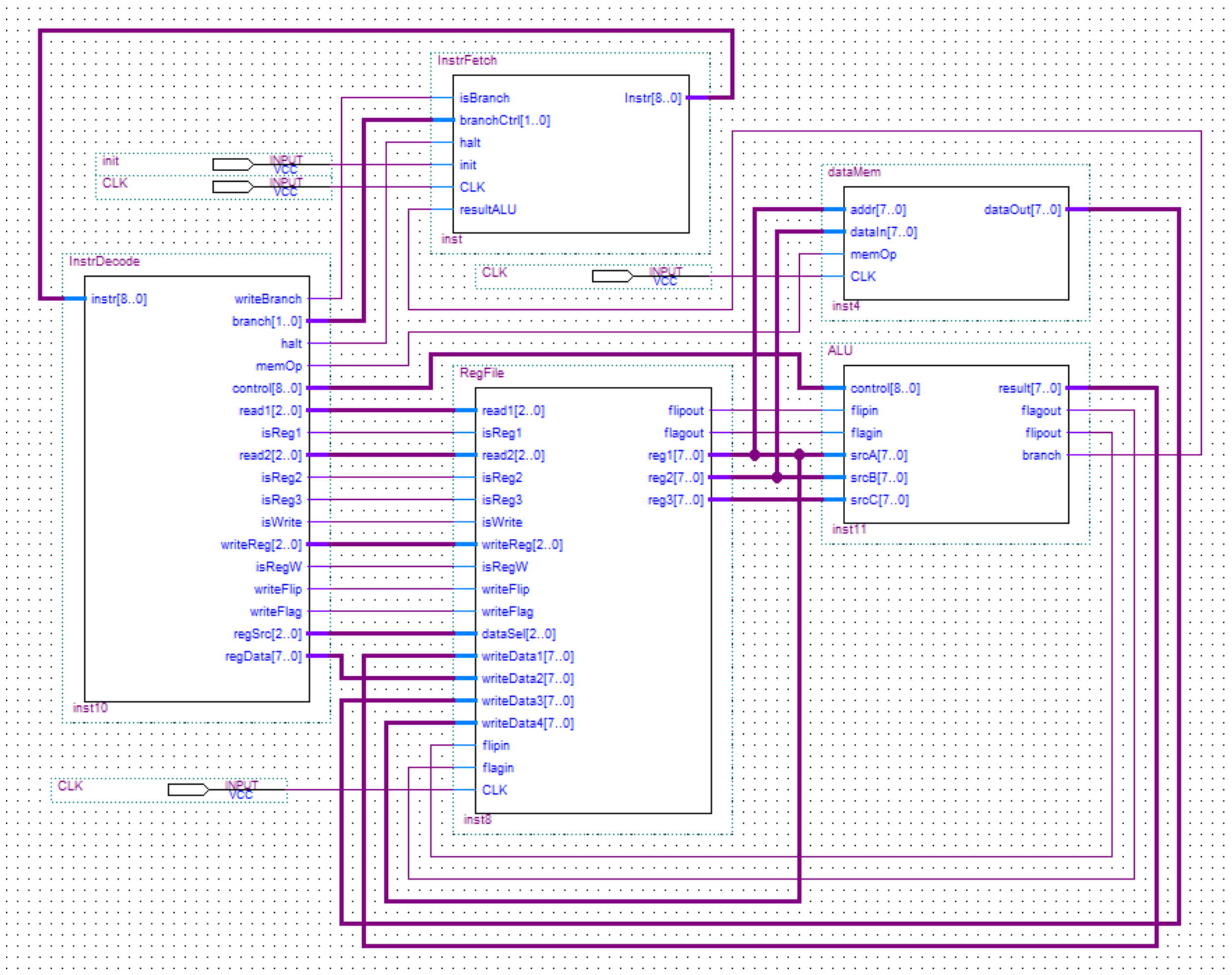
Branch if the register not equal to 31.
- `b1ne31`

4 Verilog Modules

4.1 Hierarchy Tree

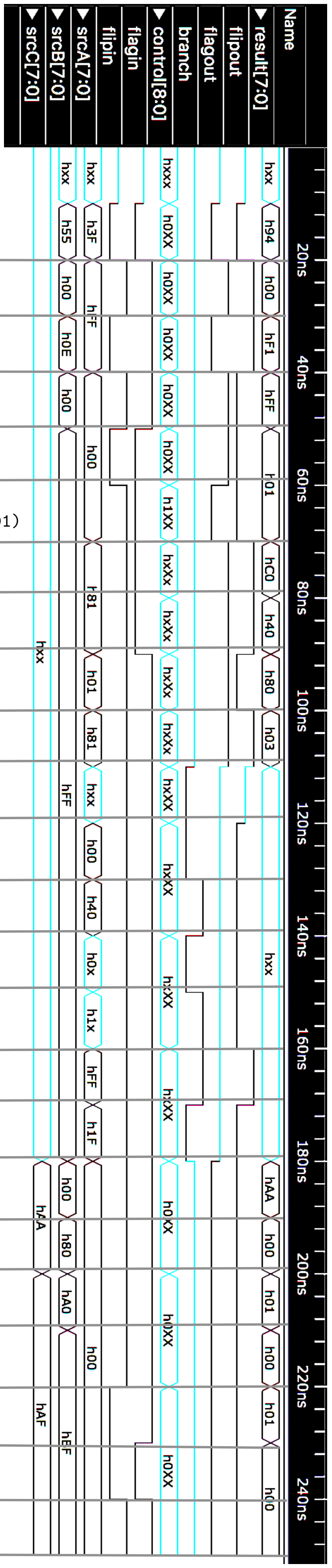


4.2 Schematics



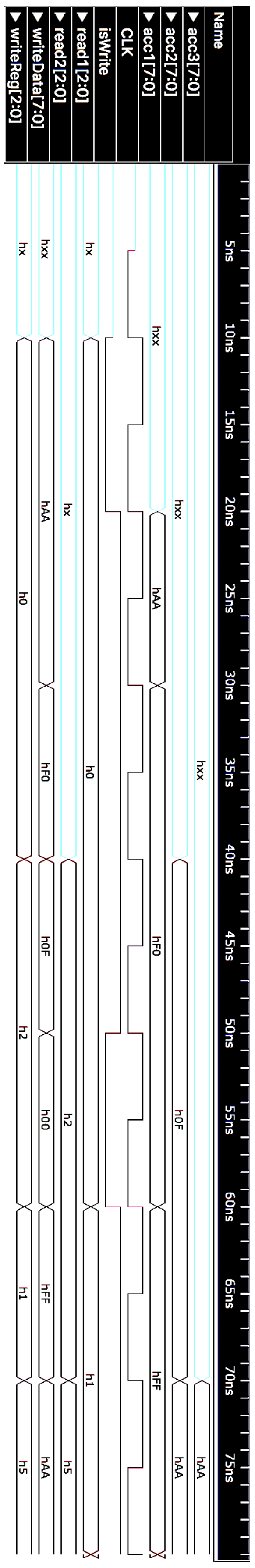
5 Time Diagram

5.1 ALU Test



```
#1 Add 2 Number Without Overflow:
srcA(00111111) + srcB(01010101) + flagin(0) = flagout(0) + result(10010100)

#2 Add 2 Number With Overflow and Carry In + test ignore Flip bit:
srcA(11111111) + srcB(00000000) + flagin(1) = flagout(1) + result(00000000)

#3 Sub 2 Number With no borrow + test ignore Flag bit:
srcA(11111111) - srcB(00001110) (flagin(1)) = flagout(0) + result(11110001)

#4 Sub 2 Number With borrow:
srcA(00000000) - srcB(00000000) - flagin(1) = flagout(1) + result(11111111)

#5 Sub 2 Number With borrow:
srcA(00000000) - srcB(11111111) - flagin(0) = flagout(1) + result(00000001)

#6 Sub 2 Number ignore srcB, use 1 instead:
srcA(00000000) - srcB(11111111) (1) - flagin(0) = flagout(0) + result(00000001)

#7 srl test right shift arithmetically
(0) srcA(10000001) >> 1 = result(11000000) (1)

#8 srl test right shift logically
(1) srcA(10000001) >> 1 = result(01000000) (1)

#9 srf test right shift use overflow bit as shift-in
(1) srcA(00000001) >> 1 = result(10000000) (1)

#10 slf test left shift use overflow bit as shift-in
srcA(10000001) (1) << 1 = (1) result(00000011)

#11 test never branch
srcA = xxxxxxxx, never branch control set, branch = 0;

#12 test branch when srcA != 0
srcA = 00000000, branch = 0;

#13 test branch when srcA != 0
srcA = 01000000, branch = 1;

#14 test branch when upper 4 srcA != 0
srcA = 0000xxxx, branch = 0;

#15 test branch when upper 4 srcA != 0
srcA = 0001xxxx, branch = 1;

#16 test branch when srcA != 31
srcA = 11111111, branch = 1;

#17 test branch when srcA != 31
srcA = 00011111, branch = 0;

#18 test result = srcA + (srcB < 0? 0: srcC)
srcA = 00000000, srcB = 00000000, srcC = 10101010; result = 10101010

#19 test result = srcA + (srcB < 0? 0: srcC)
srcA = 00000000, srcB = 10000000, srcC = 10101010; result = 00000000

#20 test result = srcA + (upper4 match srcB, srcC? 1: 0)
srcA = 00000000, srcB = 10100000, srcC = 10101111; result = 00000001

#21 test result = srcA + (upper4 match srcB, srcC? 1: 0)
srcA = 00000000, srcB = 10111111, srcC = 10101111; result = 00000000

#22 test result = srcA + flip ^ flag
srcA = 00000000, flip = 0, flag = 1; result = 00000001

#23 test result = srcA + flip ^ flag
srcA = 00000000, flip = 0, flag = 0; result = 00000000

#23 test result = srcA + flip ^ flag
srcA = 00000000, flip = 1, flag = 1; result = 00000000
```

5.2 Register File Test



```
#1 test write to and read from accumulator registers
read1 = 000, isReg1 = 0, writeReg = 000, isRegW = 0, writeData = 01101001
correctly write to $acc0
```

```
#2 test read from regular register of the same index
read1 = 000, isReg1 = 0
correctly write to $acc0
```

```
#3 test write to and read from regular registers
read2 = 001, isReg2 = 1, writeReg = 001, isRegW = 1, writeData = 11110000
correctly write to $t1
```

```
#4 test accumulative register not contaminate
isReg2 = 0
$acc1 still undefined
```

```
#5 test disabled
write new value does not write into
correctly write to $t1
```

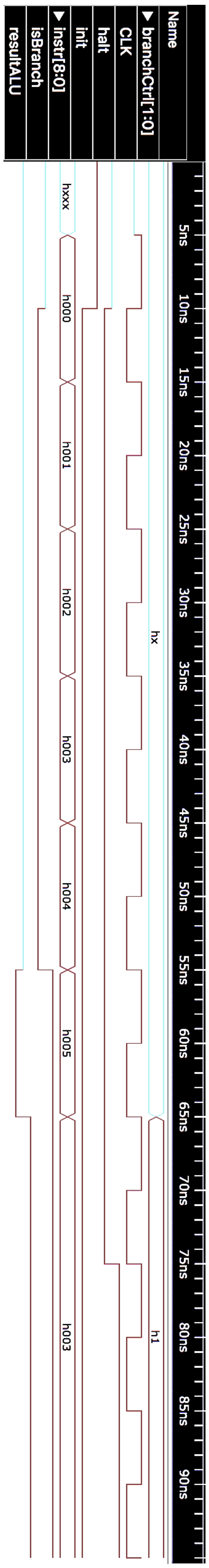
```
#6 test write to $acc0 again
$acc0 now write to 11111111
```

```
#7 test write to and read from acc5
read2 now read from acc5, it is set to new value
acc5 always output acc5, it is also set
```

5.3 Instruction Fetch Test

```
Instruction Memory is set to

  000000000
  000000001
  000000010
  000000011
  000000100
  000000101
  000000110
  ...
to match with PC

Look up table branch is
  000000000
  000000011
  100000000
  000000001
```

```
#1 test initialize
instruction set to 00000000

#2 test let clock run
instruction set to 00000001

#3 test let clock run
instruction set to 00000010

#4 test let clock run
instruction set to 00000011

#5 test let clock run
instruction set to 00000100

#6 test ALU judge branch fail
instruction set to 00000101

#7 test ALU judge branch successful
instruction set to 00000011

#8 test halt
instruction remain at 00000011
```



6 Non-Arithmetic Instruction in ALU

We use ALU for determining branching. We use 3 branching and the determination is paralleled with adder and shifter. Therefore, for time's sake there is no difference. But more gates are used to judge if branch or not.

7 Easier Instruction Fetch

We think our Instruction Fetch is already easy enough. It can be made without branching address stored in lookup table. But without lookup table the branching distance would limited to only +/- 8 instructions, which is not enough.

8 Easier ALU Design

Our ALU does too much jobs, since there are lots of instruction specifically made up for the 3 program, to reduce instruction count. If we don't need to reduce Instruction count, then the logic in ALU can be limited to adder and shifter, and become easier.

9 Easier Register File

Our register file is very complicated, since we divide it into 2 parts, Accumulator Registers and Regular Registers. It is divided, since we have only 9-bit instruction and we manage to give register 2-bit. Therefore, we came to the idea of dividing register. Regular register can be use as argument. and accumulator register can be used as accumulator, and is included in the instruction name. Division of register file is like put a mux manually between the 2 register files. If we have more bits for instruction code, then we will not need the division. And our register file can be easier.

10 Most Complex Instruction ALU

`add3_n2, add4_n2, add0c25, inc_af0` are 4 most complex instructions for ALU. They are the only 4 instructions that need ALU to preprocess the input to adder. The other instructions have value directly go to adder or shifter.

11 Additional Function of ALU

No. The design can only execute instruction that it is made for.