Single Cycle MIPS implementation in Verilog

```
single-cycle/
|
|-- mips-single-cycle.v: Design source
|-- testbench.v: Simulation source
|-- constrs.xdc: Constraints
|-- test.py: Checker in python3
|-- Instr_tb_behav.wdb: Waveform result of simulation
|-- lab1-single-cycle/: The Vivado project
```

40+ Supported Instructions

- R-Type

Func	rs	rt	rd	shift (shamt)	Opcode
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Instr	Func	Opcode	Format	Description
SLL	000000	000000	`SLL rd, rt, sa`	`GPR[rd] ← GPR[rt] << sa`
SRL	000000	000010	`SRL rd, rt, sa`	`GPR[rd] ← GPR[rt] >> sa (logical)`
SRA	000000	000011	`SRA rd, rt, sa`	`GPR[rd] ← GPR[rt] >> sa (arithmetic)`
SLLV	000000	000100	`SLLV rd, rt, rs`	`GPR[rd] ← GPR[rt] << GPR[rs]`
SRLV	000000	000110	`SRLV rd, rt, rs`	`GPR[rd] ← GPR[rt] >> GPR[rs] (logical)`
SRAV	000000	000111	`SRAV rd, rt, rs`	`GPR[rd] ← GPR[rt] >> GPR[rs] (arithmetic)`
JR	000000	001000	`JR rs`	`PC ← GPR[rs]`
JALR	000000	001001	`JALR rd, rs`	`GPR[rd] ← PC + 4, PC ← GPR[rs]`
ADD	000000	100000	`ADD rd, rs, rt`	`GPR[rd] ← GPR[rs] + GPR[rt]`
ADDU	000000	100001	`ADDU rd, rs, rt`	`GPR[rd] ← GPR[rs] + GPR[rt]`
SUB	000000	100010	`SUB rd, rs, rt`	`GPR[rd] ← GPR[rs] - GPR[rt]`
SUBU	000000	100011	`SUBU rd, rs, rt`	`GPR[rd] ← GPR[rs] - GPR[rt]`
AND	000000	100100	`AND rd, rs, rt`	`GPR[rd] ← GPR[rs] AND GPR[rt]`
OR	000000	100101	`OR rd, rs, rt`	`GPR[rd] ← GPR[rs] OR GPR[rt]`
XOR	000000	100110	`XOR rd, rs, rt`	`GPR[rd] ← GPR[rs] XOR GPR[rt]`
NOR	000000	100111	`NOR rd, rs, rt`	`GPR[rd] ← GPR[rs] NOR GPR[rt]`
SLT	000000	101010	`SLT rd, rs, rt`	`GPR[rd] ← (GPR[rs] < GPR[rt])`
SLTU	000000	101011	`SLTU rd, rs, rt`	`GPR[rd] ← (GPR[rs] < GPR[rt])`
MUL	011100	000010	`MUL rd, rs, rt`	`GPR[rd] ← GPR[rs] × GPR[rt]`

- I-Type

Opcode	rs (base)	rt	IMM
6 bits	5 bits	5 bits	16 bits

Instr	Opcode	Format	Description
BEQ	000100	`BEQ rs, rt, offset`	`if GPR[rs] = GPR[rt] then branch`
BNE	000101	`BEQ rs, rt, offset`	`if GPR[rs] ≠ GPR[rt] then branch`
BLEZ	000110	`BLEZ rs, offset`	`if GPR[rs] ≤ 0 then branch`
BGTZ	000111	`BGTZ rs, offset`	`if GPR[rs] > GPR[rt] then branch`
ADDI	001000	`ADDI rt, rs, immediate`	`GPR[rt] ← GPR[rs] + immediate`
ADDIU	001001	`ADDIU rt, rs, immediate`	`GPR[rt] ← GPR[rs] + immediate`
SLTI	001010	`SLTI rt, rs, immediate`	`GPR[rt] ← (GPR[rs] < immediate)`
SLTIU	001011	`SLTIU rt, rs, immediate`	`GPR[rt] ← (GPR[rs] < immediate)`
ANDI	001100	`ANDI rt, rs, immediate`	`GPR[rt] ← GPR[rs] AND immediate`
ORI	001101	`ORI rt, rs, immediate`	`GPR[rt] ← GPR[rs] OR immediate`
XORI	001110	`XORI rt, rs, immediate`	`GPR[rt] ← GPR[rs] XOR immediate`
LUI	001111	`LUI rt, immediate`	`GPR[rt] ← immediate << 16`
LB	100000	`LB rt, offset(base)`	`GPR[rt] ← memory[GPR[base] + offset]`
LH	100001	`LH rt, offset(base)`	`GPR[rt] ← memory[GPR[base] + offset]`
LW	100011	`LW rt, offset(base)`	`GPR[rt] ← memory[GPR[base] + offset]`
LBU	100100	`LBU rt, offset(base)`	`GPR[rt] ← memory[GPR[base] + offset]`
LHU	100101	`LHU rt, offset(base)`	`GPR[rt] ← memory[GPR[base] + offset]`
SB	101000	`SB rt, offset(base)`	`memory[GPR[base] + offset] ← GPR[rt]`
SH	101001	`SH rt, offset(base)`	`memory[GPR[base] + offset] ← GPR[rt]`
SW	101011	`SW rt, offset(base)`	`memory[GPR[base] + offset] ← GPR[rt]`

- J-Type

Opcode	Address
6 bits	26 bits

Instr	Opcode	Format
J	000010	`J target`
JAL	000011	`JAL target`
NOP	000000	`NOP`

Registers

Floating point registers are not implemented !

Register Number	Conventional Name	Usage
$0	$zero	Hard-wired to 0
$1	$at	Reserved for pseudo-instructions
$2 - $3	$v0, $v1	Return values from functions
$4 - $7	$a0 - $a3	Arguments to functions - not preserved by subprograms
$8 - $15	$t0 - $t7	Temporary data, not preserved by subprograms
$16 - $23	$s0 - $s7	Saved registers, preserved by subprograms
$24 - $25	$t8 - $t9	More temporary registers, not preserved by subprograms
$26 - $27	$k0 - $k1	Reserved for kernel. Do not use.
$28	$gp	Global Area Pointer (base of global data segment)
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address
$f0 - $f3	-	Floating point return values
$f4 - $f10	-	Temporary registers, not preserved by subprograms
$f12 - $f14	-	First two arguments to subprograms, not preserved by subprograms
$f16 - $f18	-	More temporary registers, not preserved by subprograms
$f20 - $f31	-	Saved registers, preserved by subprograms

Implementation

Control unit

The control unit computes the control signals based on `Opcode` , `Func` and `ZF` `OF` `SF` generated by ALU. Slightly different from that in the textbook, signal `Branch` and `Jump` are generalized and absorbed by a new signal `PCSrc` and the `BranchSrc` signal is computed to control the conditional branches.

- Signals for branch Instruction, based on `ZF` `OF` `SF`

  BranchSrc : Decide whether the branch target should be taken, when executing `BEQ` , `BNE` , `BLEZ` and `BGTZ` .

- Signals based on `Opcode` and `Func`

Signal	Description	Values or Select lines
PCSrc	Source of next PC	PC + 4, Branch Target, Immediate (`J` and `JAL`) , Instruction[25: 21] (`JR` and `JALR`)
RegWrite	Write enable of Register File	0, 1
RegDst	Destination register to write	Instruction[20:16] (I-Type), Instruction[15:11] (R-Type), 31 (`JAL` and `JARL`)
RegWriteData	Data to write to the destination register	ALU result, Read Data in Data Memory, PC + 4 (`JAL` and `JALR`)
ALUSrcA	Operand A of ALU	Read Data 1, Instruction[10: 5] (Shamt for shift operations)
ALUSrcB	Operand B of ALU	Read Data 2, Immediate
ALUControl	Operation of ALU	`SLL` , `SRL` , `SRA` , `LUI` , `MUL` , `MULU` , `ADD` , `ADDU` , `SUB` , `SUBU` , `AND` , `OR` , `XOR` , `NOR` , `SLT` , `SLTU`
MemWrite	Write enable of Data Memory	0, 1
MemSigned	Sign-extended options for `LOAD` instructions. (`LBU` and `LHU` should load zero-extended byte or half-word into the register.)	0, 1
MemWidth	The width of data fetched or written in Data Memory	byte, half-word, word

Memory-mapped I/O

To make it behaves as a real CPU, I try to simulate the I/O Mapping in MIPS.

- The inputs, e.g. `SW[1:0]`, are mapping directly into the Data Memory with a fixed address, decided by the designer.
- The outputs, e.g. `seed`, is mapped at another predefined address.
- Then, for CPU, it only need to read Data Memory to fetch the inputs, and modify the value at specific address to govern the outputs, using `LOAD` and `STORE` instructions.

Stack Frame

I have implemented the stack frame by making functions called and return. And the instruction combinations below are commonly used to according to **the Calling Convention** , which includes pushing and popping in the stack, calling a function and then returning from it.

- PUSH and POP

```
# PUSH
ADDI $sp, $sp, -4 # sp -= 4
SW $ra, 0($sp) # Store Return Address
......
# POP
ADDI $sp, $sp, 4 # sp += 4
LW $ra, 0($sp) # Load Return Address
```

- CALL and RET

```
# CALL
JAL target
......
# RET
JR $ra
```

- HLT

```
B -1
# Or BEQ $r0, $r0, -1
```

Demo Description

The demo is a random number generator using LCG (Linear Congruential Generator) Algorithm. It works as follows.

- Initially, a seed is given, denoted by s_0, e.g. seed = s_0 =0 .
- Then the outputs will be generated according to the recurrence

  s_{i+1} = (a \times s_{i}+c)\mod m ,

  where a, c, m are chosen carefully as 17, 3, 256 respectively since this combination can output every integer in [0, 255], which can be proved with the help of **Number Theory**

  For example,

  s_{1} = (a \times s_{0}+c)\mod m = (17 \times 0+3)\mod 256 = 3

  s_{2} = (a \times s_{1}+c)\mod m = (17 \times 3+3)\mod 256 = 54

And here is the description of I/O.

- `SW[0]`: Reset Signal. If `SW[0] = 1`, PC will be reset to 0.
- `SW[1]`: Load Enable Signal. If both `SW[0] = 1` and `SW[1] = 1`, seed will be set to `{SW[15: 4]}`.
- `SW[15: 4]`: The input seed.

Demo in Assembly

```
# LCG (Linear Congruential Generator) Algorithm
# seed = (a * seed + c) mod m;

# Initialize
ADDI $sp, $zero, STACK_BEGIN_ADDR # Set up stack pointer
```

```
LBU $s0, IN0_ADDR($zero) # Read from I/O device

LOOP:
    ORI $a1, $s0, 0 # Retrieve seed
    ADDI $a0, $zero, 17 # a = 17
    ADDI $a2, $zero, 3 # c = 3
    ADDI $a3, $zero, 256 # m = 256
    JAL LCG # Call
    ORI $s0, $v0, 0
    SB $v0, OUT0_ADDR($zero) # Write to I/O device
    B LOOP

#  return (a * seed + c) % m
LCG:
    MUL $t0, $a1, $a0 # x = a * seed
    ADDU $t0, $t0, $a2 # x += c
    ORI $a0, $t0 # Set parameter x = (a * seed) + c
    OR $a1, $a3, $zero # Set parameter m
    ADDI $sp, $sp, -4 # $sp -= 4
    SW $ra, 0($sp) # Push
    JAL MOD # Call
    LW $ra, 0($sp) # Pop
    ADDI $sp, $sp, 4 # sp -= 4
    JR $ra # RET

# return x % m
MOD:
    ADDI $v0, $a0, 0
    SUB $t0, $a1, $v0 # t0 = m - x
    SUB $t1, $zero, $v0 # t1 = -x
    BLEZ t0, 2 # If m <= x, go to Branch 1 (x -= m)
    BGTZ t1, 3 # Else if b < 0, go to Branch 2 (x += m)
    JR $ra # Else return
    SUB $v0, $v0, $a1 # Branch 1
    B -7 # Loop
    ADD $v0, $v0, $a1 # Branch 2
    B -9 # Loop
```

Demo in Python 3

Run `python3 test.py` and input a seed to check the outcome.

```
# This is used to help understanding the algorithm
# Can also used to check the answer
print ('Please input a seed')

seed = int(input())
a = 17
c = 3
m = 256

def lcg(modulus, a, c, seed):
    s = []
```

```
    while True:
        seed = (a * seed + c) % modulus
        if seed in s :
            print ('Total : ' + str(len(s)))
            break
        else :
            s.append(seed)
            yield seed

for x in lcg (m, a, c, seed):
    print (x)
```

Simulation

I have saved the simulation waveform of the module `Instr_tb` in `testbench.v` . Simulation code and answer, shown as comments, are in `testbench.v`. You can also create a project and add in `mips-single-cycle.v` as design source and `testbench.v` as simulation source to have a check.

The waveform database file `Instr_tb_behav.wdb` can be used in Vivado 2018.3 as follows :

- Open a new Vivado GUI or close all existing projects.
- Flow -> Open Static Simulation -> Open `Instr_tb_behav.wdb`
- Right-click the signals you want to trace -> Add to Wave Window (In my case, `clk` , `rst` , `ALUResult` , `Instr` , `pc` , `rf` , `dmem` , `test` are suggested)