Justin Carlson

2/13/2023

Prof. Mealy

CPE 233-07

Experiment 4: IMMED_GEN & BRANCH_ADDR_GEN

Executive Summary:

This experiment involves the implementation of two modules, the immediate value generators and branch address generators. The IMMED_GEN creates five instruction type-specific immediate fields solely from the instruction register output. The BRANCH_ADDR_GEN uses the program counter output, source register 1, and the B, I, and J-type outputs of the IMMED_GEN in order to handle branching instructions.

Verilog Code:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Justin Carlson
//
// Create Date: 02/01/2023 05:09:42 PM
// Design Name:
// Module Name: BRANCH_ADDR_GEN
// Project Name:
// Target Devices:
// Tool Versions:
// Description: Branch Address Generator
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module BRANCH_ADDR_GEN(
    input [31:0] i_type_imm, j_type_imm, b_type_imm, rs1, pc,
    output [31:0] jalr, branch, jal
    );

    assign jalr = rs1 + i_type_imm;
    assign branch = pc + b_type_imm;
    assign jal = pc + j_type_imm;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Justin Carlson
//
// Create Date: 02/01/2023 05:09:42 PM
// Design Name:
// Module Name: IMMED_GEN
// Project Name: mcu_gen
// Target Devices:
// Tool Versions:
// Description: Immediate value generator
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module IMMED_GEN(
    input [31:7] ir,
    output [31:0] u_type_imm, i_type_imm, s_type_imm, j_type_imm,
b_type_imm
    );

    assign u_type_imm = {ir[31:12], 12'b0}; // U-Type
    assign i_type_imm = {{21{ir[31]}}, ir[30:20]}; // I-Type
    assign s_type_imm = {{21{ir[31]}}, ir[30:25], ir[11:7]}; // S-Type
    assign j_type_imm = {{12{ir[31]}}, ir[19:12], ir[20], ir[30:21],
1'b0}; // J-Type
```

```
    assign b_type_imm = {{20{ir[31]}}, ir[7], ir[30:25], ir[11:8],
1'b0}; // B-Type


endmodule
```

Top Level:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////
////////////
// Company:
// Engineer: Justin Carlson
//
// Create Date: 02/01/2023 03:19:32 PM
// Design Name:
// Module Name: mcu_gen
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////
////////////


module mcu_gen(
    input rst, PCWrite, clk,
    input [1:0] pcSource,
```

```
    output [31:0] u_type_imm, s_type_imm
    );

    wire [31:0] pc, ir, i_type_imm, j_type_imm, b_type_imm, jalr,
branch, jal;

    PC_MOD PC_MOD (
        .rst (rst),
        .clk (clk),
        .PCWrite (PCWrite),
        .pcSource (pcSource),
        .PC (pc),
        .jalr (jalr),
        .branch (branch),
        .jal (jal)   );

    Memory OTTER_MEMORY (
        .MEM_CLK    (clk),
        .MEM_RDEN1  (1'b1),
        .MEM_RDEN2  (1'b0),
        .MEM_WE2    (1'b0),
        .MEM_ADDR1  (pc[15:2]),
        .MEM_ADDR2  (32'd0),
        .MEM_DIN2   (32'd0),
        .MEM_SIZE   (2'b10),
        .MEM_SIGN   (1'b0),
        .IO_IN      (1'b0),
        .IO_WR      (),
        .MEM_DOUT1  (ir),
        .MEM_DOUT2  ()  );

    IMMED_GEN IMMED_GEN (
        .ir (ir),
        .u_type_imm (u_type_imm),
        .i_type_imm (i_type_imm),
        .s_type_imm (s_type_imm),
        .j_type_imm (j_type_imm),
        .b_type_imm (b_type_imm)   );

    BRANCH_ADDR_GEN BRANCH_ADDR_GEN (
```

```
        .i_type_imm (i_type_imm),
        .j_type_imm (j_type_imm),
        .b_type_imm (b_type_imm),
        .rs1 (32'h0000000C),
        .pc (pc - 4),
        .jalr (jalr),
        .branch (branch),
        .jal (jal)   );

endmodule
```

Questions:

1. There are five non-nop instructions in the sample program. Each of these instructions includes a comment that contains an “equation of importance”, which you used in this experiment to prove that your hardware was working correctly. For this question, completely and explicitly explain how we formed that desired value on the right side of the equation relative to the given program. Don’t wimp out on the description for this problem; make your answer very complete.

For the first jal instruction, we took the label dog, which is at address 8. Since the other register was zero, we can treat jal like a jump to x0 + 8, which remains at address 8. For the jalr instruction, it advances the PC by 4 and puts it into the destination register x0, then it takes the immediate value -8 and adds it with rs1 which is x20, and sets this to the new PC value, creating a jump to 4. For the branch statement, it checks that x10 and x10 are equal registers, and since they are, it branches to the address at the label “cat” which is at the destination of 4. The sw instruction takes the immediate value outside of the parentheses which is 12 and extends it to the s-type format, which are the bits 32h’0000000C. The lui takes the immediate value 255, and left shifts the value 12 times with 0 and loaded into the register which is the data 0x000ff000

2. The stack is a useful “mechanism” in MCUs and we refer to the stack as an “abstract data type”. Define ADT in your own words and in the context of computer hardware.

An ADT is in essence just a concept or a model for a way somebody could represent data in a figurative context. In order to actually use an ADT in a program, it must be implemented based on how you want to handle your data. For the use in hardware, the designer must figure out a way with physical components how they can store and access data as given by the specifications of the ADT.

3. There are five instruction types that include immediate value; list which of those types are sign-

extended by the IMMED_GEN module.

The U-type requires 12 extra 0 bits at the end, and the J and B-types require a single bit at the end.

4. In computerland, we typically increase the bit-width of data in two ways. List those two ways and describe what type of data they are used on.

You can either replicate a set of bits n times or concatenate a given set of bits with your current data. The first case was used in this lab where we duplicated a single bit preserving the signed nature of the number, and the second one with concatenating sets of bits can be used to utilize a bus on a port with a different size than your data.

5. There is obviously no “stack” module in the RISC-V MCU architecture diagram. Where exactly is the stack then on the RISC-V MCU?

There is a reserved space inside our memory module purely dedicated to storing stack information.

6. We all know the stack for its push and pop operations, but there are no push and pop instructions in the RISC-V ISA. Show the two versions of how we implement both push and pop operations using the RISC-V ISA (show two sets of code for each the push and pop operations).

In order to push, the stack pointer must be increased by 4 to determine where the next word is stored, then we store the word.

```
addi sp, sp, 4
sw register, 0(sp)
```

In order to pop, we must retrieve the word from memory by loading it to a register then decreasing the stack pointer by 4.

lw register, 0(sp)

addi sp, sp, -4

7. We use the stack for “general data storage”, but we typically use the stack to store information in two scenarios. What are those scenarios and what information does the stack store?

The stack helps us preserve context when we need to call another subroutine, and the stack can also be used to store data in an output register when you exit a subroutine. In the first case, the stack stores register data that the new subroutine will overwrite, and the second case the stack will store the output register value while the program then restores context.

8. The RISC-V MCU has 32 “general purpose” registers. Briefly describe why RISC-V MCU programmers should never use registers x1 and x2.

Registers x1 and x2 are the return address and the stack pointer, and overwriting one of these registers will break the program flow.

9. What is the official term for the item the stack pointer is pointing at in the RISC-V MCU hardware?

Top of the stack

10. I just added 34 new instructions to the RISC-V instruction set to support the application I'm working on. Is the RISC-V MCU still a RISC-V architecture or not? Briefly but fully explain.

It's not really the RISC-V architecture anymore, as adding that many new instructions would require significant changes to the MCU, and at a certain point most of the underlying RISC-V MCU would be different than what it was before in order to support the new operations.

11. Briefly explain why stack overflow and underflow are major issues in assembly language programs.

Not pushing and popping the proper number of registers for each subroutine call will lead to registers being filled with information pertaining to other subroutines and the program will not work as intended.

12. Why is it that we need to match call/return instruction pairs (meaning the order they are called in matters) but we don't need to follow a similar approach with push/pop pairs. Briefly but fully explain.

Call and returns must be matched as not abiding to that as it will cause stack underflow or overflow, but pushing and popping isn't constrained by that same requirement. Since there's no dedicated instruction for pushing and popping and we just load and store words in the stack, we can essentially push them and pop don't have to be done on the top of the stack, which can sometimes be useful.

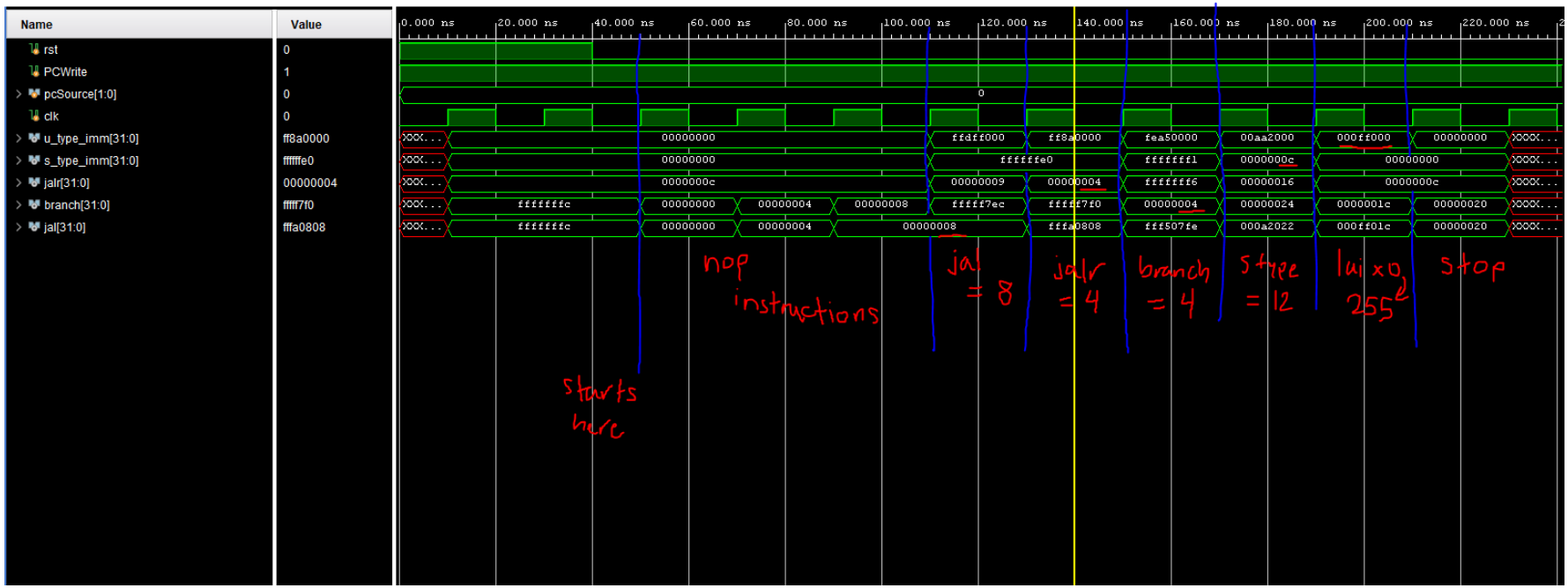


Figure 1. Timing Diagram

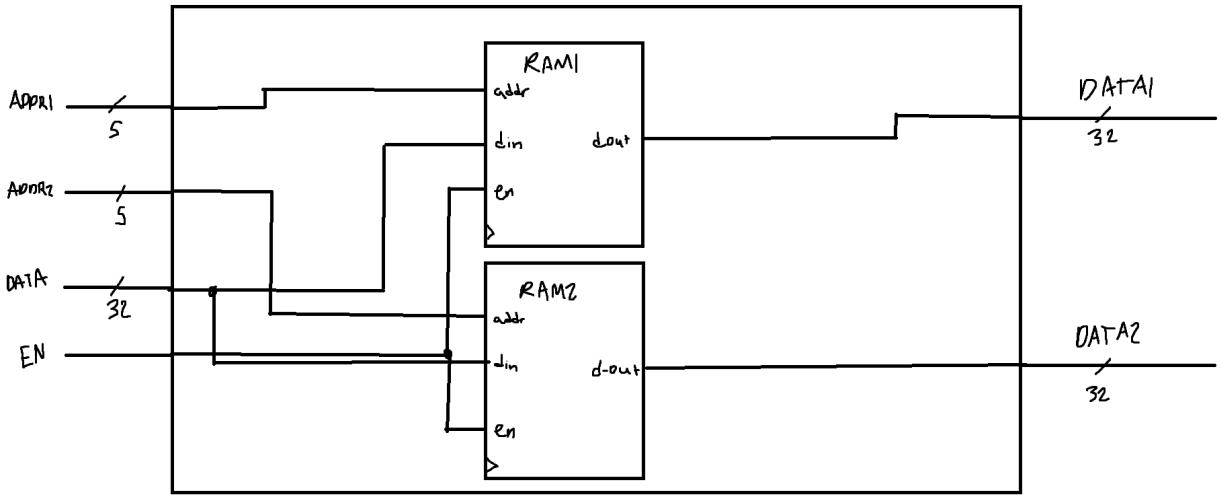
Programming Assignment:

```
store: addi sp, sp, 20
sw x20, 0(sp)
sw x5, 4(sp)
sw x6, 8(sp)
sw x7, 12(sp)
sw x8, 16(sp)


init: mv x20, x10
mv x5, x0
mv x6, 0
mv x7, 0
mv x8, 0

thous: blt x20, x0, hund_init
addi x20, x20, -1000
addi x5, x5, 1
j thous

hund_init: addi x20, x20, 1000

hund: blt x20, x0, tens_init
addi x20, 20, -100
addi x6, x6, 1
j hund

ten_init: addi x20, x20, 100

tens: blt x20, x0, ones_init
```

```
addi x20, x20, -10
addi x7, x7, 1
j tens

ones_init: addi x20, x20, 10

ones:
beq x20, x0, nibbs
addi x20, x20, -1
addi x8, x8, 1
j ones

nibbs: mv x11, x5
slli x11, x11, 4
add x11, x11, x6
slli x11, x11, 4
add x11, x11, x7
slli x11, x11, 4
add x11, x11, x8

restore: lw x20, 0(sp)
lw x5, 4(sp)
lw x6, 8(sp)
lw x7, 12(sp)
```

```
lw x8, 16(sp)
addi sp, sp, -20
ret
```

Hardware Problem:



Note: Since each RAM is 32x32 and you wanted two of them, I'm confused how the whole register is also supposed to be 32x32 since the two combined storages of the RAMs is 64x32.