

CME341 Part III: Designing a Microprocessor

Created by Eric Salt

REVISION HISTORY:

- Apr 8, 2013: document was created
- May 2, 2013: added section on making bidirectional buses with inout pins
- Oct 29, 2013: 1.Moved the circuit (single D flip/flop) from the program sequencer to the top module for the microprocessor
2.Added synchronous clearing of all 4-bit registers in the computational unit and the synchronous setting of zero flag
3.Added a section that walks through an example program
- Nov 7, 2013: Modified the way the sync_reset works. Made changes to the schematics for the Micro block diagram, instruction decoder and computational unit.
- Oct 17,2014: Corrected typo in Part III notes that referred to the zero flag as the carry flag.
- Nov 23, 2014: 1. Modified the implementation of the RAM based stack so that a pop could immediately follow a push or another pop. With the modified implementation there is no need for the assembler to insert a no-operation in places where a pop follows a push or another pop.
2. Added a description of the modifications required to implement multi word instructions.
- Nov 24, 2014: Corrected some typos in the section on stacks and clarified the explanation of expanding memory by paging.
- Dec 8, 2014: Corrected the following typos reported by students (primarily Ian Abbott):
1. Machine code for line H'22 in Table 5.
2. Address in last line of Tables 1 to 5 was changed from 8'H27 to 8'H30
3. Corrected erroneous statements in the comment field for addresses 8'H24 and 8'H25 in the example on page 23.
4. Corrected "else" part in the always @ * procedure in the Verilog HDL for the addresses generator for a RAM based stack (on page 55).
- Nov. 2, 2016: Corrected two errors in Table 8: Solution ... computational unit. The last two entries for the i register were changed from 4'H8 and 4'H7 to 4'H9 and 4'H8
- Oct. 18, 2017: Typed in the handwritten notes on extra information for the instruction decoder.
- Nov. 1, 2017: Corrected value of zero flag in Tables 4 and 8.

Nov. 24, 2017: Clarified some explanation in Part III b)
on circuits used for exam questions.

Oct. 30, 2018: Added notes on preamble for the second midterm.

Nov. 7, 2018: Added notes on organizing a Quartus project to facilitate
doing for second midterm.

Nov. 30, 2018: Corrected omission of "@" in the description of DAGs

Oct. 22, 2020: added the missing ir[3] wire in the computational unit

Nov. 1, 2020: Revised the timing diagram on page 31 and added the wires
from the output of register r to the ALU logic block

Nov. 23, 2020: Revised section on zero overhead loops, including figures
for the simple loop and the for loop.

Nov. 26, 2020: Revised sections on interrupts and subroutines

Nov. 29, 2020: Revised sections on paged memory and DAGs

Nov. 29, 2023: Revised the explanation of ALU instructions -x and ~x

Contents

1	Designing a Microprocessor	5
1.1	Concept of Executing Instructions	5
1.2	Description of the CME341 Microprocessor	7
1.2.1	General Description	7
1.2.2	Machine Code / Instruction set Mapping	8
1.2.3	Reset Circuit	8
1.2.4	Program Sequencer	9
1.2.5	Program Memory	10
1.2.6	Instruction Decoder	12
1.2.7	CME341 - Extra notes on Instruction Decoder	16
1.2.8	Computational Unit	23
1.2.9	Data Memory	23
1.3	Schematic Diagrams for the CME341 Microprocessor	24
1.4	Example	30
2	Preparation for the Second Midterm	41
3	Making Tristate Buses and Tristate Pins	54
4	Foundation for Exam Questions	63
4.1	Zero-Overhead Loops	63
4.2	Adding Interrupt Capability	69
4.3	Hardware Support for Breakpoints	72
4.4	Hardware Support for Subroutines	73
4.4.1	Example	80
4.5	Building LIFO (stack) and FIFO Queues with FSMs	82
4.6	Extending the Address Range for Memory	85
4.7	Variable Length Instruction Set	87
4.8	Wait on External Flag	88
4.9	Co-processor	89
4.10	Data Address Generators (DAGs) for Circular Buffers	89
4.11	Timers: Regular and Watchdog	90

List of Figures

1	Timing for instructions flowing through program memory	31
2	Schematic diagram for a Cyclone I/O element (IOE)	55
3	A block diagram of devices on a PCB connected with a tristate bus	56
4	Test bench waveforms for the system in Figure 3	62
5	A block diagram of the hardware that supports a simple overhead loop	65
6	A block diagram of the program-sequencer hardware that supports a simple For-Loop	67
7	Part of a block diagram of the program-sequencer hardware that supports a simple while-Loop	69

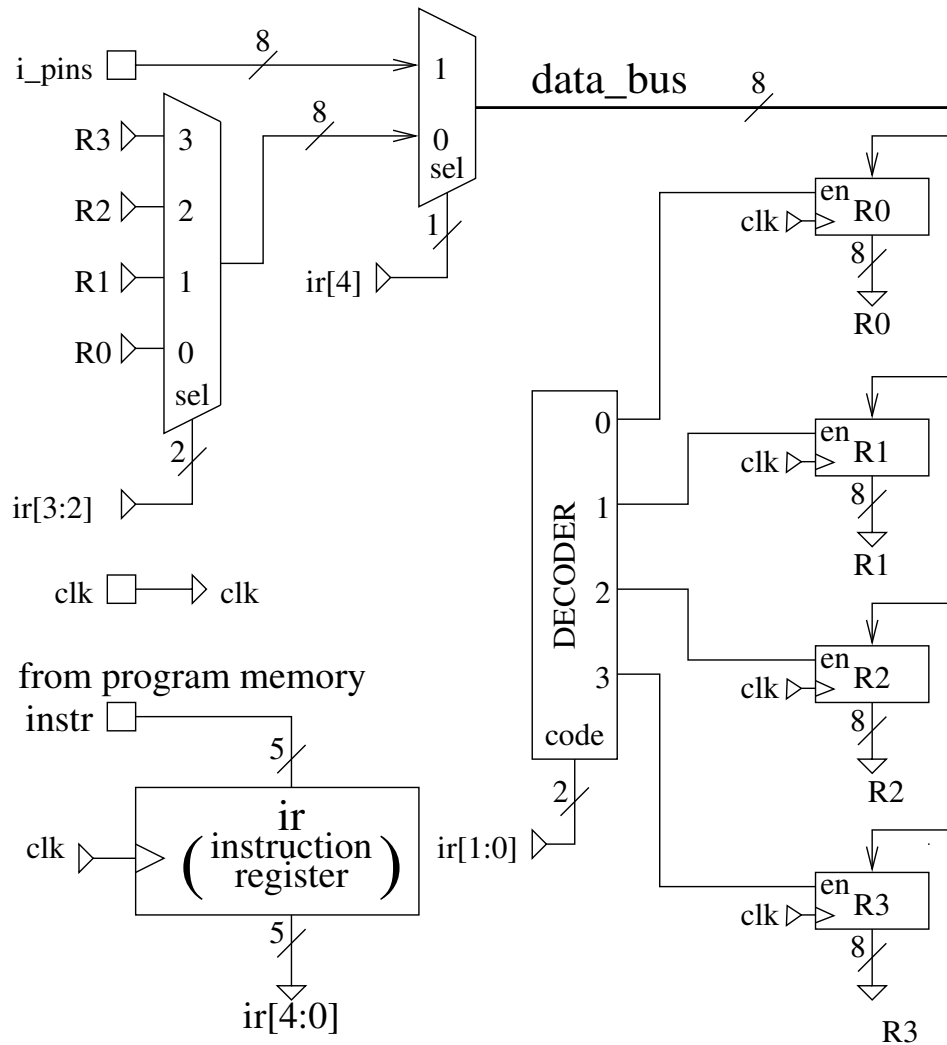
8	The interface between the program sequencer and the external interrupt signal	71
9	Illustration of the transfer of control to and from subroutines	74
10	A circuit for an 4-word by 8-bits/word stack based Registers	76
11	A circuit for an 8-word by 8-bits/word stack based on a 2-port synchronous RAM	78
12	A Stack implemented with registers and a FSM	83
13	A FIFO Queue implemented with registers and a FSM	84
14	A block diagram of the hardware need to support paging of program memory.	86
15	A block diagram for a Data Address Generator (DAG)	91

1 Designing a Microprocessor

1.1 Concept of Executing Instructions

Principle of operation

The circuit below illustrates how instructions are executed in microprocessors. This circuit has four 8-bit data registers labelled R0, R1, R2 and R3. It also has a 5 bit instruction register. It has but three inputs: a clock named *clk*, an 8-bit input called *i_pins* and a 5-bit program memory input called *instr*.



The circuit can only move data around so is not a functional micro processor. On every positive clock edge whatever is on the data bus is loaded into one of the data registers. What is loaded and where it is loaded is determined by the instruction in the instruction register (*ir*).

The least two significant bits of the *ir*, denoted *ir[1 : 0]* are the code input to the decoder. The decoder makes the output that correspond to the unsigned value of the code high and the other 3 outputs low. Therefore, one and only one register will be enabled and that register is determined by unsigned value of *ir[1 : 0]*.

The select input to the 4-to-1 multiplexer is $ir[3 : 2]$, therefore $ir[3 : 2]$ determine which of the 4 data registers is selected as a candidate for the data bus.

The select input to the 2-to-1 multiplexer is $ir[4]$. It determines whether the input `i_pins` or a data register is put on the data bus.

Examples

Suppose the contents of the instruction register is $ir == 5'b01100$. Then $ir[4] == 1'b0$ which has the 2-to-1 multiplexer selecting a data register so a data register will be put on the data bus. Furthermore $ir[3 : 2] == 2'b11$, which forces the 4-to-1 multiplexer to select register R3. Finally $ir[1 : 0] == 2'b00$ which has the decoder enabling R0. This means that the contents of R3 will be on the data bus and register R0 will be enabled. On the next positive edge of `clk` register R0 will be loaded with the contents of register R3. The instruction that was in the ir (i.e. $5'b01100$) could be described as move R3 to R0. The instruction is said to be executed at the instant R3 is loaded, which is on the clock same edge that loads the next instruction into the ir .

Exercise 1

Describe the operation of the execution of the three instructions: $5'b10010$, $5'b01001$, $5'b01011$.

Exercise 2

Find the 5 bit machine code for the two instructions below:

1. load R3 with `i_pins`
2. move R3 to R1

1.2 Description of the CME341 Microprocessor

1.2.1 General Description

The CME341 is a very simple microprocessor (4-bit data path and 8-bit instruction) with a modified Harvard architecture. A Harvard architecture is one that has separate memories for program store and data store. The program store is ROM and the data store is RAM. The modified Harvard architecture allows for data constants to be fetched from program store. This architecture is in contrast to the von Neumann architecture, which uses the same memory for program store and data store.

The program memory for the CME341 microprocessor is 256 words with each word being 8 bits. The data memory is only 16 words with each word being 4 bits.

All instructions are 8 bits in length and occupy exactly one program memory word. This means two things: First, since there are 2^8 codes in an 8 bit word, there can be at most $2^8 = 256$ unique instructions. Second, since program memory is 256 words in length, a program for the CME341 microprocessor can be at most 256 instructions long.

The microprocessor is constructed from five circuits plus a single flip/flop referred to as the reset circuit. The structure of the microprocessor is shown on Page 1/5 of the schematic diagrams which begin on page 25. The microprocessor has 6 inputs: a clock, an asynchronous reset, and a 4-bit input port. The output is one 4-bit port.

The operation of the microprocessor is as follows. A new instruction is loaded into the instruction register on every positive going edge of the system clock. The instruction comes from (is read from) the program memory. The instruction decoder decodes the instruction in the instruction register with combinational logic to produce signals that control the program sequencer and the computational unit. No action is taken on these control signals until the next positive going clock edge, at which time a register is written. The writing of this register occurs at the same time as the instruction register is loaded with the next instruction.

Commonly the phrase “executing an instruction” is used. Executing is the instantaneous action of writing a register. It happens virtually instantaneously. The instruction in the instruction register is executed on the first positive going clock edge after it was loaded into the instruction register.

The program sequencer is a circuit that, under control of signals generated by the instruction decoder, generates the address for program memory. This address determines the output of the program memory, which is the next instruction to be loaded into the instruction register.

The computational unit performs mathematical operations and also moves data from one register to another.

The data memory is a store for data that can be read or written.

1.2.2 Machine Code / Instruction set Mapping

The 8-bit instruction is partitioned into 4 types of instruction: load, move, arithmetic/logic, and jump. Each type of instruction partitions the instruction into fields as shown on page 26, which is the second page of the five page set of schematic diagrams.

The load instruction loads the register specified in the destination field with the lower four bits that are currently in the instruction register.

The move instruction will normally overwrite the register specified in the destination field with the contents of the register specified in the source field. There is a exception, which is a move instruction where the source and destination fields have the same register ID and that ID is not 3'H4. In the exception the input *i_pins* is moved to the register specified in the destination field.

The arithmetic/logic instruction will evaluate a function of two variables and write the result to the result register, which is labeled the *r* register. One is called the *x* argument and the other is called the *y* argument. The *x* argument is either register x_0 or register x_1 and the *y* argument is either register y_0 or register y_1 . Which *x* and *y* register is used is determined from the *x* and *y* fields in the ALU instruction. The function that is to be performed, e.g. $x + y$, is specified in the function field. Since the function field has 3-bits, 8 different functions can be specified.

The last instruction type is a jump, which can be a jump (unconditional) or a conditional jump. A “jump” instruction, if executed, forces the next address for program memory to be a specified value. For any other instruction type, as well as a conditional jump without the accompanying necessary condition, the next program memory address is the address of the instruction currently in the instruction register (this is the address in the PC) plus 8'H01.

The jump instruction causes the next instruction to come from program memory address { *addr*, 4'H0 }, where *addr* is the least significant four bits of the jump instruction. Therefore, the jump instruction can force a jump to one of the following 16 memory locations: 8'H00, 8'H10, 8'H20, ..., 8'HF0.

The conditional jump instruction becomes a jump instruction if and only if the most recent arithmetic/logic instruction yielded $r \neq 4'H0$ (i.e. resulted in the *zero_flag* being zero). In other words the jump is conditional on the zero flag being 1'b0. In the case of the zero flag being zero the conditional jump instruction does nothing and the next program memory address is $PC + 8'H01$.

1.2.3 Reset Circuit

The reset circuit consists of a single D flip/flop. The input is from a pin called **reset**. It is connected to the D input of the flip/flop and the output is a signal called **sync_reset**. The function of the circuit is to synchronize the reset input to the positive edge of the clock. This will ensure **sync_reset** is asserted and de-asserted immediately after a positive edge of the clock.

The **sync_reset** signal is to force the following actions:

1. Force the 8-bit vector output “`pm_addr`” of the program sequencer to be 8'H00 while `sync_reset = 1'b1`.
2. Synchronously clear the eight 4-bit registers in the computational unit.
3. Synchronously set the zero flag flip/flop in the computational unit.

The `sync_reset` signal **must not reset** the instruction register nor the program counter.

The `sync_reset` signal is connected to the program sequencer, instruction decoder and computational unit.

1.2.4 Program Sequencer

The program sequencer computes the program memory address for the instruction that is to be loaded into the instruction register on the next positive clock edge. It must present this address to program memory and the program memory must output the associated data prior to the next positive edge of the clock. The block diagram for the program sequencer is given on page 28, which is the fourth page of the five page set of schematic diagrams.

To compute the address of the next instruction, i.e. to compute output `pm_addr`, the program sequencer must know the address of origin of the instruction that is currently in the instruction register. For easy reference the instruction in the *ir* it is referred to as the current instruction.¹ The address of origin of the current instruction is held in an 8-bit register that resides in the program sequencer called the program counter. The mnemonic for the program counter register is *pc*.

The program sequencer continually outputs the address for the instruction that is to be loaded into the instruction register on the next positive clock edge. The output of program memory, which connects to the *ir*, is the next instruction to be loaded into the *ir* and is referred to as the next instruction.

The operation of the program sequencer is quite straight forward, especially when it comes to resetting it. The input called “`sync_reset`” forces the program sequencer to make `pm_addr` the value 8'H00 while “`sync_reset=1'b1`”. **`sync_reset` does not clear the program counter.**

When `sync_reset` is not asserted, the program sequencer computes the address for program memory according to the following rules:

1. If input `jmp` is asserted then the current instruction is a “jump”(unconditional). The output, i.e. `pm_addr`, must be { `jmp_addr`, 4'H0 }.
2. If input `jmp_nz` is asserted then the current instruction is a “conditional jump”. The jump is only executed if the zero flag in the computational unit is 1'b0 (The zero flag is 1'b1 if and only if the result register contains 4'b0.) The input `dont_jmp` is connected to the zero flag so the conditional jump is executed if and only if `dont_jmp` is **not asserted**, i.e. `dont_jmp == 1'bo`. If the conditional jump is to be executed then `pm_addr` must be { `jmp_addr`, 4'H0 }.

¹Referring to the instruction in the *ir* as the current instruction may seem counter intuitive as it is executed on the next clock edge, which happens to be the clock edge that overwrites it. For that reason it could have been called the “next instruction”, but it is not.

- If `dont_jump == 1'b1`, then output `pm_addr` must be `pc + 8'H01`. In the event the `pc == 8'HFF`, then adding `8'H01` will roll the `pc` over to `8'H00`.
3. If inputs `jump` and `jump_nz` are both `1'b0`, then the current instruction is either a load, a move or an ALU instruction. In this case the program sequencer makes `pm_addr` equal to `pc + 8'H01`. In the event the `pc == 8'HFF`, then adding `8'H01` will roll the `pc` over to `8'H00`.

1.2.5 Program Memory

The program memory is a read-only memory. It is made from a RAM on the FPGA by initializing the RAM at compile time. While the FPGA is being configured, the write enable for the RAM is enabled and the RAM is written with the initial values. At the end of the configuration process the RAM is disabled and becomes a ROM.

The values used to initialize the RAM are contained in a hex file, which is a file with a “.hex” extension. Such a file is more properly referred to as an Intel hex file, as Intel specified the format for the file. Quartus provides a tool to generate hex files.

The memory blocks in FPGAs consist of a core of asynchronous RAM with registers connected to the inputs and/or the output of this asynchronous RAM. Whether or not the registers can be bypassed to configure the memory block as asynchronous RAM/ROM depends on the type of FPGA. For example the FLEX10K family has circuitry that allows the registers around the asynchronous memory core to be switched in or out, which allows the RAM/ROM to be configured as either asynchronous or synchronous RAM/ROM. The memory blocks in a Cyclone II family has the circuitry to bypass the register connected to the output, but does not have the circuitry to bypass the registers connected to the inputs. Registers are permanently connected to the address, data-in and write-enable inputs of the Cyclone II memory blocks. This means the memory block can not be configured as asynchronous RAM/ROM.

Synchronous RAM/ROM can easily be converted to pseudo-asynchronous RAM/ROM, but at the expense of propagation delay. The conversion is done by simply clocking the input registers of the RAM/ROM with the negative edge of the clock. For this simple conversion to work the address (and in the case of RAM also the input data) must be stable before the negative edge of the clock. The data at the output appears some time after the negative edge of a clock as the propagation through the address decoding circuit does not start until the input registers are loaded. To view this as pseudo-asynchronous RAM/ROM, one needs to view the address-to-data-out propagation time as 1/2 the period of the clock plus the address-to-data-out propagation time for the asynchronous RAM.

Making the initialization file for program memory

ROM built by Quartus is initialized, i.e. its contents loaded, at compile time. The content of the ROM is specified in a separate file in either Intel-format hexadecimal format, i.e. a .hex file, or memory initialization file format, i.e. a .mif file. Quartus provides a spread-sheet type of editor to construct the initialization file and save it as either a .hex or .mif file.

It is probably a wise to use the .hex format at the initialization files provided in exams will be in .hex format.

The process for constructing the initialization files in Quartus is as follows:

1. Pull down the file menu and select **new** → **Memory Files** → **Hexadecimal (Intel-format) file** or pull down the file menu and select **new** → **Memory files** → **Memory initialization file**.
2. Fill out the pop-up window to specify the dimensions of the ROM, i.e. make the **number-of-words** 256 and the **word-size** 8-bits.
3. The editor opens as a spread sheet with each cell representing the contents of one word in the memory. The address at which the contents of the cell will be stored in ROM is the sum of its vertical and horizontal co-ordinates.
4. The radix of the co-ordinates can be changed by pulling down the **view** menu. Set the radix for the address (i.e. the co-ordinates for the cells) to hexadecimal.
5. The radix for the number typed into the cell can also be set by pulling down the **view** menu. Set the radix for the memory contents to hexadecimal.
6. The number of cells in one row (i.e. the aspect ratio of ratio of the spread sheet) can be changed by pulling down the **view** menu. Set the spread sheet to 8 cells per row.
7. After entering the desired contents the file can be saved and re-opened in the usual fashion.

Making the ROM for program memory using a Megafunction

Notification of a Bug in Quartus: There is a bug in the current Linux version of Quartus that affects the construction of the program memory ROM. The bug causes one of the windows in the sequence of windows that open when a function in the IP catalog is clicked upon to freeze. The fix is to grab a corner or side of the first window that opens and adjust its size, even slightly, before proceeding in the sequence and before filling in any of the boxes.

work in progress

1.2.6 Instruction Decoder

Instruction Set for the CME341 Microprocessor

There are four types of instructions, each of them are single cycle (executed in one clock cycle), 8-bit instructions. The types of instruction are: load instruction, move instruction, ALU instruction and jump instruction. A graphical illustration of how the 8-bit instruction is partitioned into the 4 different types is given on page 26.

Each instruction, no matter what type, is transferred from the 256X8 program memory (the program memory is 256 words in length with each word being 8 bits) to the instruction register (*ir*) on the rising edge of the clock. The instruction is executed on the next rising edge of the clock. The instruction register is 8 bits wide (*ir*[7 : 0]), with the most significant bit being *ir*[7] and the least significant being *ir*[0].

There are nine 4-bit data registers (the 8-bit *pc* and 8-bit *ir* are not data registers). The *r* register can only be used as a source and the *o_reg* can only be used as a destination. The nine registers are assigned the 3-bit identification numbers as follows:

ID#	reg	comments
000	x_0	
001	x_1	
010	y_0	
011	y_1	
100	<i>r</i>	when the ID is used in a source register field
100	<i>o_reg</i>	when the ID is used in a destination register field
101	<i>m</i>	
110	<i>i</i>	
111	<i>dm</i>	not a single register, the source or destination is data memory

Note that the *r* register and *o_reg* have the same IDs, but are uniquely specified because one is a source and the other a destination.

Load Instruction

This instruction loads a destination register with the contents of the least significant 4 bits of the *ir* register. i.e. copies the contents of the least significant 4 bits of the *ir* to a data register or data memory.

ir[7] == 0 specifies a load instruction

ir[6 : 4] holds the ID of the destination register

ir[3 : 0] is the four bit constant loaded into the destination register,

An assembler statement for the load instruction that load the 4-bit constant 4'HE into register y_0 could be something like: " $y_0 = EH$ " or " $ld\ y_0, EH$ ".

There is a special case, which is the “load data memory” instruction. The “load data memory” instruction executes a second instruction in parallel with loading the data memory. That second instruction is

$i = i + m$; which increments the i register by the value held in the m register.

Mov Instruction

This instruction copies the contents of a source register to a destination register.

$ir[7] == 1$ and $ir[6] == 0$ specifies a move (mov) instruction

$ir[5 : 3]$ holds the ID of the destination register

$ir[2 : 0]$ holds the ID of the source register

Exceptions: As there is no point in copying a register to itself, the machine codes for a “mov” instruction where the source and destination IDs are the same are interpreted as follows:

If the source and destination are the same, but not equal to 3'H4, then input i_pins is used in place of the source register and is moved to the destination register that specified ID in the destination field.

There is not an exception if both the source and destination IDs are 3'H4, i.e., if $ir[5 : 3] == ir[2 : 0] == 3'H4$. In this case the source register is the r register and the destination is the o_reg register and the instruction moves r to o_reg .

As with the “load” instruction, there are special cases of the “mov” instruction where a second instruction is executed in parallel. The special cases are:

1. When the destination register is data memory.
2. When the source register is the data memory and the destination register is any register except the i register.

As with the “load” instruction, the second instruction executed in parallel is $i = i + m$;

An assembler statement for the mov instruction that moves r to x_0 could be something like: “ $x_0 = r$ ” or “mov r, x_0 ”.

ALU Instruction

ALU instructions have two operands. One is the x -input operand, which is either the x_0 or x_1 register, and the other is the y -input operand, which is either the y_0 or y_1 register. The result of every ALU instruction is written to the r register and the zero flag flip/flop. The r register and zero flag flip/flop are only written on ALU instructions.

There are four fields in an ALU instruction (see page 26 for graphical illustration). The fields are: instruction type (3 bits), x-register select (1 bit), y-register select (1 bit) and ALU function (3 bits). These fields are described below.

Instruction type: The 3-bit field is $ir[7 : 5]$. It must have values $ir[7 : 6] == 2'b11$ and $ir[5] == 1'b0$ to specify an ALU instruction.

x-register select: The 1 bit field is $ir[4]$. If $ir[4] == 1'b0$ register x_0 is selected as the x-input to the ALU. If $ir[4] == 1'b1$ register x_1 is selected as the x-input to the ALU.

There are two single argument functions that do not involve the y input. These functions are $r = -x$ and $r = \sim x$. For these functions to take effect the y -register select (i.e. $ir[3]$) must be $1'b0$. If $ir[3] == 1'b1$, the single argument instructions become write r and zero flag to themselves. In the latter case $ir[4]$ has no effect.

y-register select: The 1 bit field is $ir[3]$. If $ir[3] == 1'b0$ register y_0 is selected as the y -input to the ALU. If $ir[3] == 1'b1$ register y_1 is selected as the y -input to the ALU.

This field plays a different role for the one argument ALU functions, which are $r = -x$ and $r = \sim x$. The one argument instructions do not involve the y -input and therefore $ir[3]$ is not used to specify the y register. For the single argument instructions $ir[3]$ must be $1'b0$. If $ir[3] == 1'b1$, the single argument instructions become write r and zero flag to themselves.

ALU function: The three bit field is $ir[2 : 0]$. This field specifies the ALU function. These functions are described in detail below.

The three bit ALU function field allows for 8 different ALU operations. Two of the 8 ALU function codes are divided into two instructions. In both cases one of the two instructions writes the r register to its self and the zero flag to its self. As these actions amount to doing nothing, the write r and zero flag to themselves instructions are referred to as “no-operation” instructions or simply no-ops. The 8 ALU functions and associated codes are described below.

ALU function

```

000 & ir[3]==0:  r=-x; r gets two's complement of x. This instruction could
                  also be called "test for x==4'H0" because the zero
                  flag will be set if, and only if, x has value 4'H0.
000 & ir[3]==1:  r=r; and zero_flag = zero_flag;
                  i.e. no operation,
                  Note: ir[4] could be 1'b0 or 1'b1
001:             r=x-y; subtraction using unsigned arguments
010:             r=x+y; addition using unsigned arguments
011:             r = most significant nibble of x*y;
```

```

                                multiplication using unsigned arguments
100:                            r = least significant nibble of x*y;
                                multiplication using unsigned arguments
101:                            r = x^y;
110:                            r = x&y;
111 & ir[3]==0:  r = ~x; r gets ones complement of x. This instruction could
                                also be called "test for x==4'HF" because the zero
                                flag will be set if, and only if, x has value 4'HF.
111 & ir[3]==1:  r = r; and zero_flag = zero_flag;
                                i.e. no operation,
                                Note: ir[4] could be 1'b0 or 1'b1

```

NOTE 1. The *zero_flag* is written (updated) at the same time the *r* register is written. (In hardware the *zero_flag* is enabled with the same clock enable as the *r* register) It is set (i.e. equal to 1'b1) if the result register is written with 4'b0000. It is cleared if the result register is written with anything else. The *r* register and *zero_flag* are written on every ALU instruction and **not** written on any other type of instruction. In the case of a 'no-operation' instruction, which is considered an ALU instruction, the *r* register is written to its self as is the *zero_flag*.

NOTE 2. An assembler statement for an ALU instruction that adds x_0 and y_1 and writes the result into *r* could be something like " $r = x_0 + y_1$ " or "add x_0, y_1 ".

Jump Instructions

There are two types of jump instruction, a jump (unconditional) and a conditional jump. The jump instruction has two fields: instruction type and jump address. The instruction type field is *ir*[7 : 4].

If *ir*[7 : 4] == 4'HE the instruction is a jump. The jump instruction causes the program sequencer to make the program memory address equal to {*ir*[3 : 0], 4'b0000}. No registers are enabled by a jump instruction.

If *ir*[7 : 4] == 4'HF the instruction is a conditional jump. This instruction does one of two things depending on the status of the *zero_flag* at the time the instruction is executed. If the *zero_flag* is zero the jump is executed, which means the program sequencer must make the address for program memory {*ir*[3 : 0], 4'b0000}.

If the *zero_flag* is 1'b1 then the jump will not be executed which means the program sequencer must make the address for program memory one higher than the address for the current instruction. If the address for the current instruction is 8'HFF, then the next address should be 8'H00.

The condition for activating the conditional jump instruction can be a bit confusing. The conditional jump instruction could be better described as "jump if the *r* register is non-zero". The contents of *r* is **non-zero** if and only if the *zero_flag* is **not set** (i.e., is 1'b0).

An assembler statement for a jump to program memory address 8'HA0 could be something like "jump A0H" or "jmp A0H". An assembler statement for a conditional jump to

program memory address 8'H70 could be something like “jnz 70H”, where “jnz” is the mnemonic formed from the first letters of the words “jump not zero”.

sync_reset

While input `sync_reset` is 1'b1, the instruction decoder is to ignore the instruction in the instruction register and force `reg_en` to 9'H1FF, `source_sel` to 4'd10 and all of `i_sel`, `x_sel`, `y_sel`, `jmp` and `jmp_nz` to 1'b0.

Input `sync_reset` must **not** clear the instruction register.

1.2.7 CME341 - Extra notes on Instruction Decoder

There are two different coding styles that can be used to make the instruction decoder. Both seems logical, but one is much more difficult to write and debug than the other. The two styles are referred to as “parsing the input” and “generating one output at a time”. The first is not recommended.

Parsing the input (*NOT GOOD*)

Here parsing means separating into its parts.

```
case ir
  8'd0: begin
    ...specify all outputs for this case
  end
  8'd1: begin
    ...specify all outputs for this case
  end
  ...
  8'd255: begin
    ...specify all outputs for this case
  end
end
```

The problem with parsing the input is that each output must be specified 256 times; once for each case. The process of debugging involves checking an output. If the output is not functioning properly, the VERILOG code is examined. When input parsing is used, 256 statements have to be checked. Not only are there many statement to be checked, they are embedded in a sea of other output statements. This frustrates the checking process.

Just One output per “always” (*GOOD STYLE*)

In this style most of the outputs are most logically described with an `if - else` statement. When `case` statements are used they do not use 256 case items.

When it comes to debugging, this style is much better than parsing the input. The VERILOG code will be compact, probably, less than 20 lines.

Example on a mini-scale

The instruction decoder has to make `enables` for 9 registers. When it comes to instantiation, it is convenient to put all 9 `enables` in a vector, say, `reg_enables[8 : 0]`, where `reg_enables[8]` is the clock enable for `o_reg`, otherwise `reg_enables[n]` is the enable for register `[n]`.

The logic for the `reg_enable` can be built in two ways:

1. One “`always @*`” where the entire `reg_enables` vector is defined.
2. One “`always @*`” for each of the 9 `enables` in `always @*`

In my opinion, the probability of making a mistake is much less with method 2. And if a mistake is made it is much easier to find.

In class exercise

1. Make `reg_enables[6]`, which is the enable for the `i` register in one `always` procedure.
2. Make the entire `reg_enables` vector in one `always` statement.

There is a test bench for the instruction decoder posted on the CME341 website, inside a folder called “`instruction_decoder_testbench`” which is inside the “`Micro Extras`” folder.

```

1  /*****
2
3      Instructions For Using This Testbench
4  1. Make a Modelsim Altera project called
5     test_bench_for_instruction_decoder and include this as
6     well as the .vo file for your instruction decoder.
7
8  2. At the bottom of this file there is an instantiation of
9     the instruction decoder. Change the connection list so that
10    the names for the ports in the instruction decoder are the
11    one you used.
12
13  3. Compile the project and load the simulation.
14
15  4. In the transcript window do the wave.do file.
16
17  5. Load the memory file ``memory_for_inst_decoder_test.hex''
18     as follows:
19     i) With the wave window active select view. Check to see
20        if the ``Memory List'' option is check marked. If it
21        is not, click on it to activate it.
22     ii) Select the "Memory List" tab ( in the same row as the
23         project tab). This will bring up a window that
24         shows all the memories in the test bench.
25         In this case there is only one memory.
26     iii) Double click on the memory in the Memory List window.
27          This will bring up a window showing the contents of
28          the memory. At this point it should be filled with
29          ``don't cares'' i.e. x's.
30     iv) Select File -> import -> Memory Data to bring up
31          a window that allows you to select the file you
32          need to import.
33     v) In the ``Import Memory'' window check the appropriate
34         radio buttons and type in the file name to make:
35         Load Type = File Only
36         File Format = Verilog Hex
37         File name = memory_for_inst_decoder_test.hex
38
39         Then click O.K.
40     vi) Run the simulation (i.e. type the command run -all in
41         the transcript window.)
42
43  ****
44
45  DESCRIPTION OF THE TEST BENCH
46
47  This test bench will check the non-trivial outputs of the
48  instruction decoder.
49
50  The heart of the test bench is a ram, which is initialized
51  with a hex file and used as a ROM. The initialization hex
52  file was created by writing the outputs of a working instruction
53  decoder to the ram. The contents of the ram were then copied
54  to the initialization hex file "memory_for_inst_decoder_test.hex"
55
56  After the initialization hex file was created, the test bench
57  was modified so that the ram is no longer written
58  (i.e. the write enable was connected to 1'b0).
59
60
61  The test bench produces an output vector called
62  output_vector_comparison. This vector is the
63  xnor of the output of the ram with outputs
64  from the instruction decoder under test and then
65  ored with a mask that masks all the ``don't cares''
66  occurrences for the signals.
67
68  To make things work a two-register `pipe-line' is
69  needed. (If you count the register in the instruction
70  decoder it would be called a three register pipeline.)
71  This means the first valid output is after the

```

```

72 third rising edge of the clock.
73
74 To show the value of ir that caused a particular
75 ``output_vector_comparison'', the input pm_data has been delayed
76 3 clk cycles to produce ``instr_reg'', which is ir delayed
77 to correspond to ``output_vector_comparison''.
78
79 NOTE: The select line that controls the 2-1 mux at the input
80 to the i register can be either a 1'b0 or 1'b1
81 (i.e. is a don't care) when the clock enable for the
82 i register is inactive (i.e. low). To make the
83 ``output_vector_comparison[15]'' reflect this fact,
84 it is forced high when the clock enable for i
85 register is low.
86
87
88
89 NOTE: The ``source_register_select'' can be any value during
90 an ALU, jump or conditional jump instruction as the the
91 source bus multiplexer is not used for these instructions.
92 This test bench forces ``output_vector_comparison[5:2]'',
93 which are the bits that verify ``source_register_select'',
94 high for these instructions.
95
96 NOTE: The x and y register select lines that control the muxes
97 on the input of the ALU and are not verified in this test bench.
98 */
99
100
101 `timescale 1 us / 1 ns
102 module test_bench_for_instruction_decoder ();
103
104 reg clk;
105 reg [7:0] pm_data;
106 reg [7:0] pm_data_delayed_1, pm_data_delayed_2, instr_reg;
107 reg [15:0] test_inst_decoder_ram [0:255];
108 reg [15:0] output_vector_comparison, output_vector_delayed;
109 reg we;
110 reg load_or_mov_instrucion;
111 reg sync_reset;
112 wire [15:0] output_vector, correct_output_vector;
113
114 wire unconditional_jump, conditional_jump, i_mux_select;
115 wire[3:0] source_register_select;
116 wire[8:0] register_enables;
117 wire [3:0] LS_nibble_of_ir; // output of instruction decoder but
118                             // not used in this test bench
119 wire y_mux_select, x_mux_select; // output of instruction decoder but
120                             // not used in this test bench
121
122 initial #260 $stop;
123
124 // modified by NTN on Nov.1
125 initial
126     begin
127         sync_reset = 1'b1;
128
129         #0.080 sync_reset = 1'b0; // at 80ns turn to zero
130     end
131
132 initial
133     we = 1'b0; // this must be 1'b0 when in test mode
134             // and 1'b1 when the gold standard is used
135             // to generate the memory
136
137 initial clk = 1'b0;
138 always #0.5 clk =~clk;
139
140 initial pm_data = 8'H0; // input to the instruction decoder
141 always @ (posedge clk)
142     #0.010 pm_data <= pm_data+8'b1; // test sequence is a counter

```

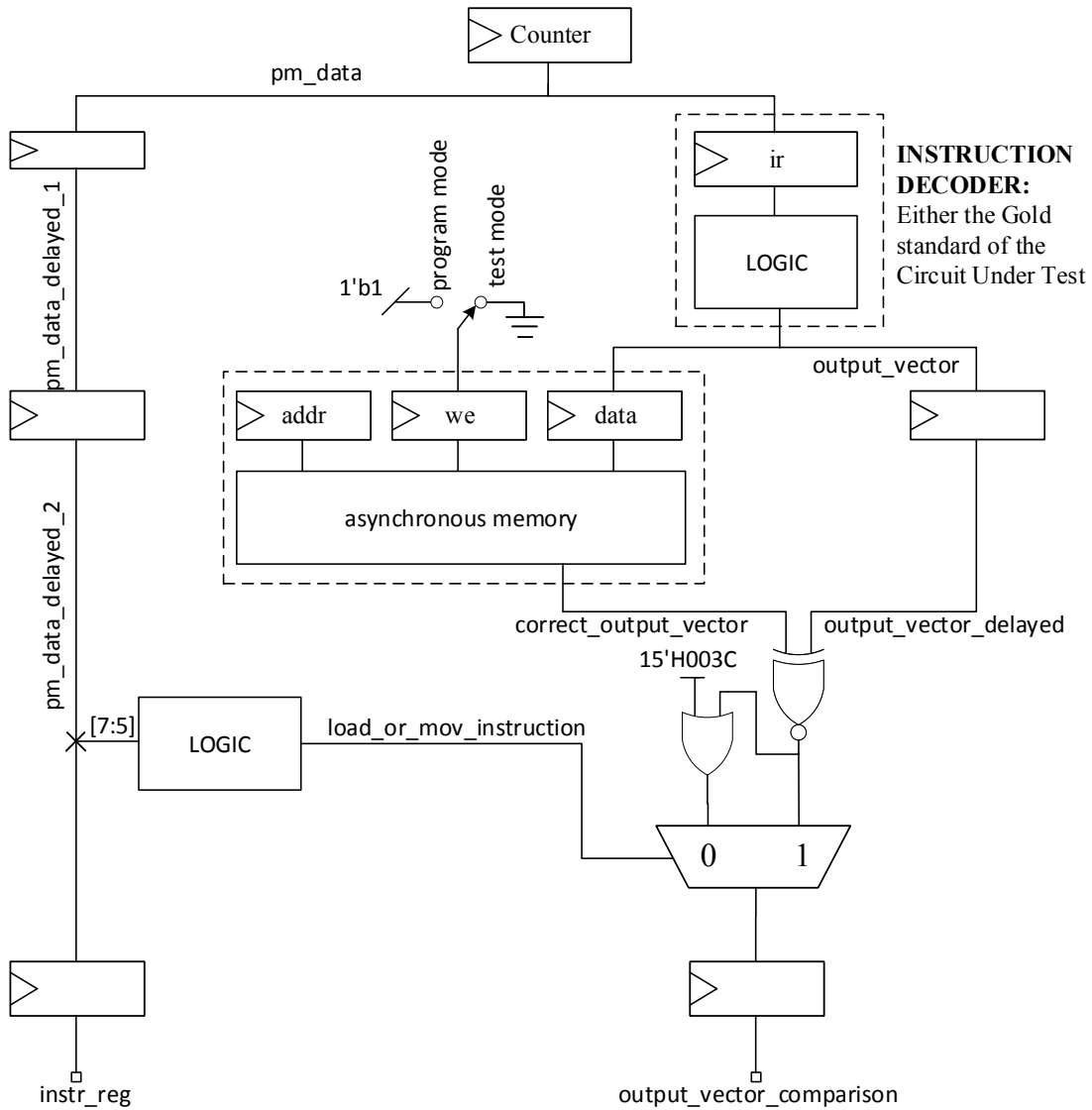
```

143
144 always @ (posedge clk)
145     begin
146         pm_data_delayed_1 <= pm_data;
147         pm_data_delayed_2 <= pm_data_delayed_1;
148         instr_reg <= pm_data_delayed_2; //a delay of 3 clock periods
149         // is necessary to make the contents of `instr_reg'
150         // line up with output_vector_comparison
151     end
152
153 always @ *
154     load_or_mov_instrucion <= ~pm_data_delayed_2[7] | (pm_data_delayed_2[7] &
155         ~pm_data_delayed_2[6]);
156
157 always @ (posedge clk)
158     output_vector_delayed <= output_vector;
159
160 /* *****
161 *****
162
163 Test result are in the vector `output_vector_comparison'.
164 `output_vector_comparison' should be FFFF from the instant
165 instr_reg == 8'H00 to the end. Any bits in `output_vector_comparison'
166 that are not 1 indicate the instruction in instr_reg is not executed properly.
167
168 The 16 bits in `output_vector_comparison' indicate whether or not there is an
169 error
170 in the corresponding bit position of the concatenation of signals
171 {i_mux_select, register_enables[8:0], source_register_select[3:0],
172 conditional_jump, unconditional_jump}.
173 A 0 or x in `output_vector_comparison' indicates an error.
174
175 ***** */
176
177 always @ (posedge clk)
178     if (load_or_mov_instrucion == 1'b1) // source bus is used
179         output_vector_comparison <= (output_vector_delayed ~^
180             correct_output_vector) |
181             {correct_output_vector[12],15'h0};
182         //correct_output_vector[12] corresponds to
183         // reg_enables[6]
184     else // source bus is not used so mask bits corresponding to source_bus_select
185         output_vector_comparison <= (output_vector_delayed ~^
186             correct_output_vector) | 16'b0000_0000_0011_1100
187         | {correct_output_vector[12],15'h0};
188
189 /* *****
190 *****
191
192 MEMORY SECTION
193 ***** */
194
195 always @ (posedge clk)
196     if (we)
197         // test_inst_decoder_ram[pm_data] = output_vector;
198         test_inst_decoder_ram[pm_data_delayed_1] = output_vector;
199     else
200         test_inst_decoder_ram[pm_data_delayed_1] = test_inst_decoder_ram[pm_data_delayed_1];
201
202 assign correct_output_vector = test_inst_decoder_ram[pm_data_delayed_2];
203
204 /* *****
205 *****
206
207 INSTANTIATION SECTION
208 ***** */
209
210 //THIS TO BE ASSUMED THAT i_sel will be 1'b1 ON IDLE
211
212 assign output_vector = {i_mux_select, register_enables, source_register_select,
213     conditional_jump, unconditional_jump};
214
215 instruction_decoder inst_decoder_1 (
216     .clk(clk),
217     .sync_reset(sync_reset), // added in Nov.1
218     .next_instr(pm_data),
219     .jmp(unconditional_jump),

```

```
210         .jmp_nz(conditional_jump),
211         .ir_nibble(LS_nibble_of_ir),
212         .i_sel(i_mux_select),
213         .y_sel(y_mux_select),
214         .x_sel(x_mux_select),
215         .source_sel(source_register_select),
216         .reg_en(register_enables));
217
218     endmodule
219
```

TEST BENCH FOR THE INSTRUCTION DECODER



If a bit in “output_vector_comparison” is HIGH then decoder under test generated the bit correctly for the instruction in “instr_reg”

1.2.8 Computational Unit

The essence of the operation of the computational unit is captured in the block diagram on page 29. Some helpful nomenclature is given below.

Data Registers and zero flag

The 4-bit registers referred to as data registers are x_0 , x_1 , y_0 , y_1 , o_reg , m , i , and dm . The r register is called a result register and the zero flag is considered to be an extension of the result register.

Registers x_0 , x_1 , y_0 , y_1 , o_reg , m , and i are synchronously cleared when `sync_reset` is high, but the `sync_reset` input to the computational unit does not do this. The reset action is set up by the instruction decoder. It enables all registers and forces the source register select multiplexer to select 4'd10 while `sync_reset` is high.

The `sync_reset` input to the computational unit only affects the ALU circuit. While `sync_reset` is 1'b1 the ALU outputs `alu_out` and `alu_out_eq_0` are forced to 4'H0 and 1'b1, respectively. This action will set the zero flag and clear the r register as the enables for both will be 1'b1 while `sync_reset` is 1'b1.

index Register

One of the data registers is also used as an index register. The index register is register i . It is a post “auto increment” index register. It behaves exactly like the other registers for every instruction that writes to register i . It behaves differently than the other registers for instructions that read or write data memory, with one exception, which is the “move data memory to register i ” instruction. For all instructions where data memory is read or written, except the instruction mentioned above, register i is to be written with $i + m$, i.e. $i = i + m$, on the clock edge that executes the instruction. Obviously register i must be written with dm on the “move data memory to register i ” instruction.

Offset Register

The m register is a general register that is also used as the offset register. Register m determines the size of the auto-increment of the i register.

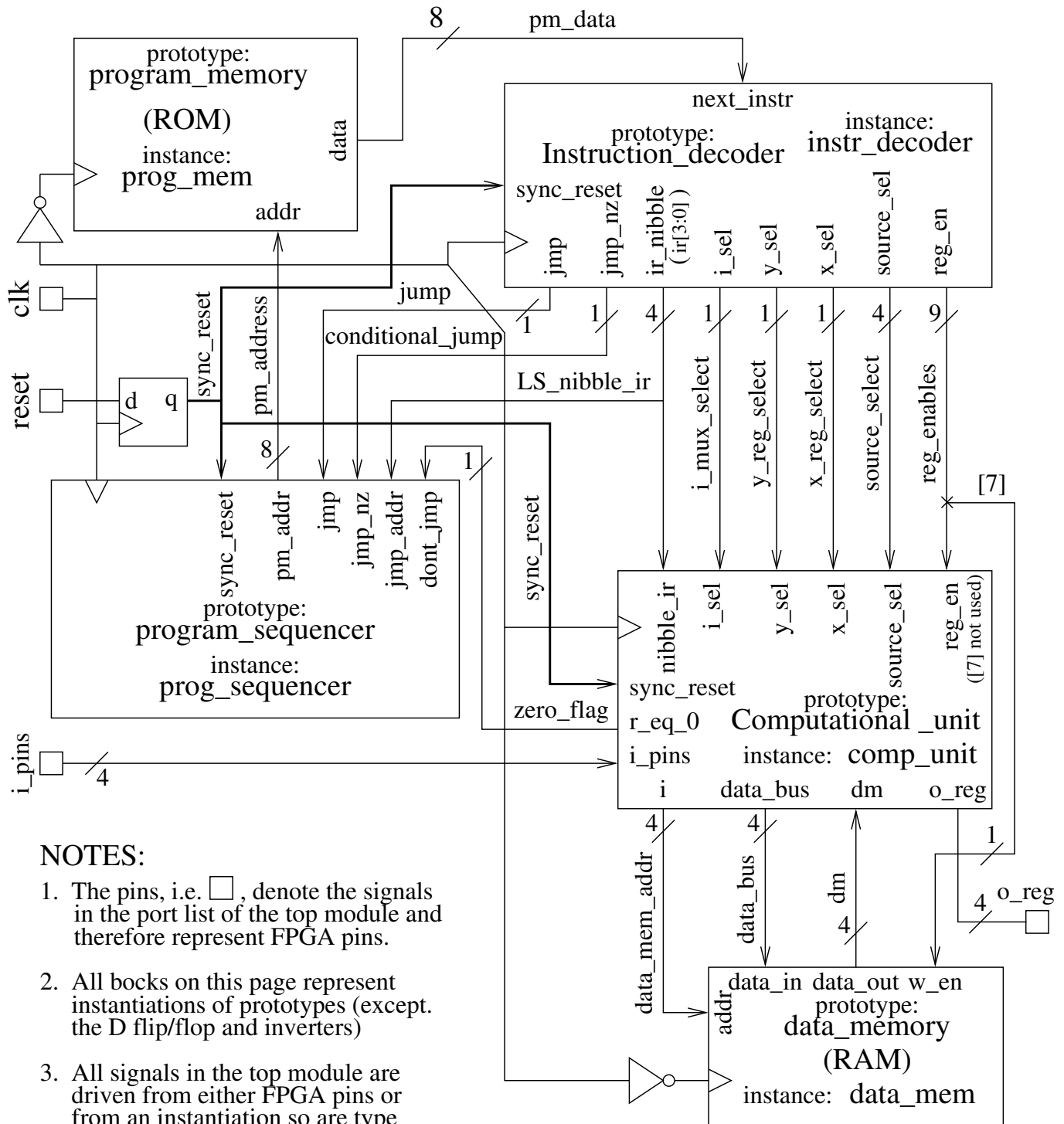
Data Memory

Data memory is connected to the data bus and is treated like a register by the computational unit. The address for the data memory is the i register. A read or write to dm is treated the same as a read or write to any other register.


1.2.9 Data Memory

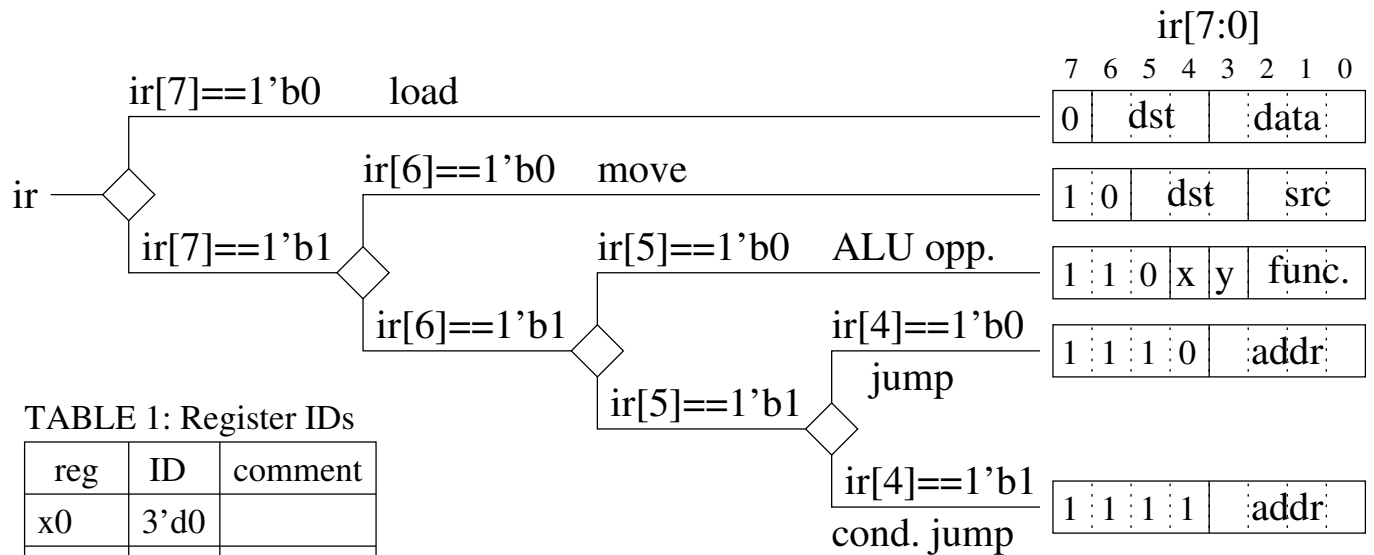
Data memory is synchronous random access memory. Its inputs are registered (both data and address) and its outputs are not. It is write-enabled with the “dm” register enable and clocked with the negative edge of “clk”. Its address is the output of the i register.

1.3 Schematic Diagrams for the CME341 Microprocessor



NOTES:

1. The pins, i.e. , denote the signals in the port list of the top module and therefore represent FPGA pins.
2. All blocks on this page represent instantiations of prototypes (except the D flip/flop and inverters)
3. All signals in the top module are driven from either FPGA pins or from an instantiation so are type wire.
4. The two inverters can be implemented implicitly inside the connection list with the association ".clk(~clk)".



Notes: when ID 3'd4 is used in the "src" field it refers to the "r" register. When it is used in the "dst" field it refers to the "o_reg" register.

Special Cases for the Move Instruction: (i.e. exceptions to the rule)

1. Move i_pins to register with ID 'dst'

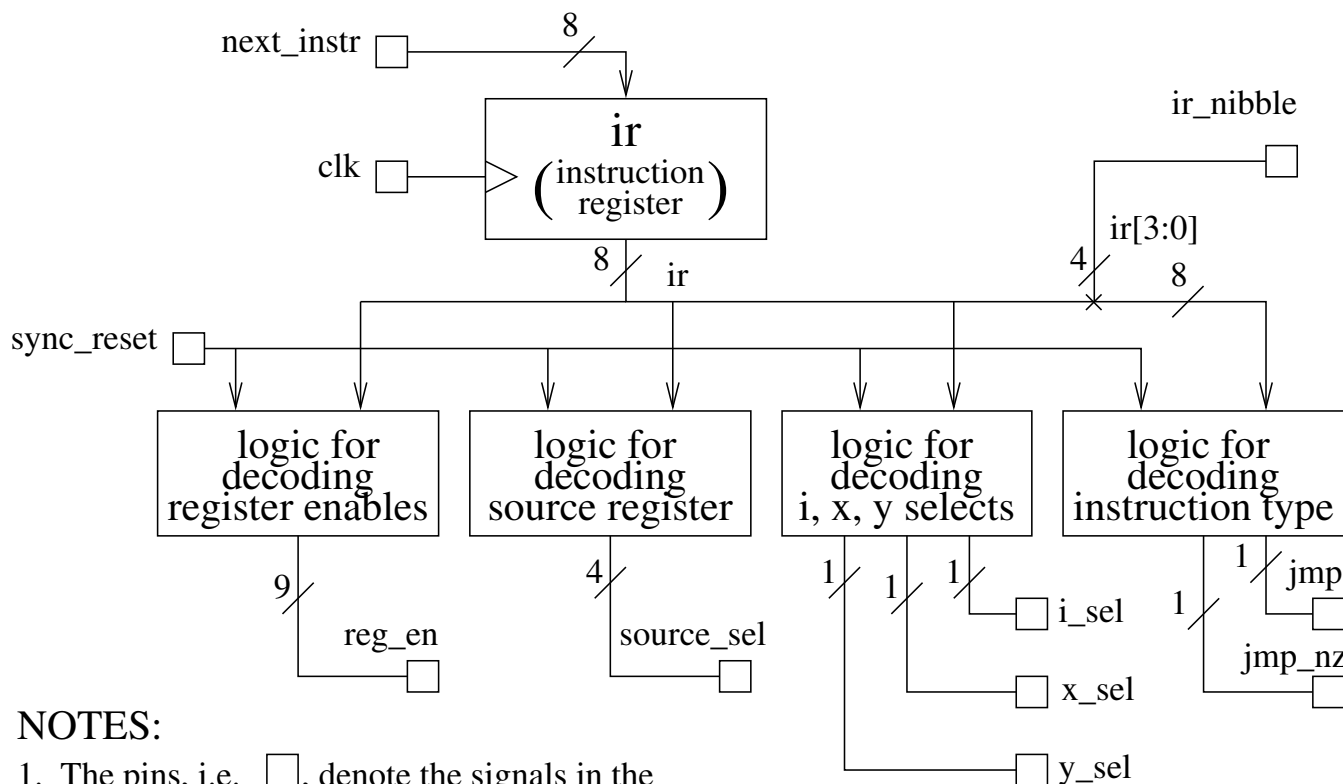
If the "src" and "dst" fields in a move instruction are such "dst==src" and "dst" is not the ID for o_reg, then the move instruction moves "i_pins" to the register with ID "dst".

2. Move r to o_reg

If the "src" and "dst" fields in a move instruction are such "dst==src==3'd4", then 'r' is moved to o_reg.

Auto Increment of i register:

The i register is automatically incremented by the value in the m register upon execution of any load or move instructions where "dm" is in the "src" or "dst" field except the move instructions where "dm" is the "src" and "i" is the "dst".



NOTES:

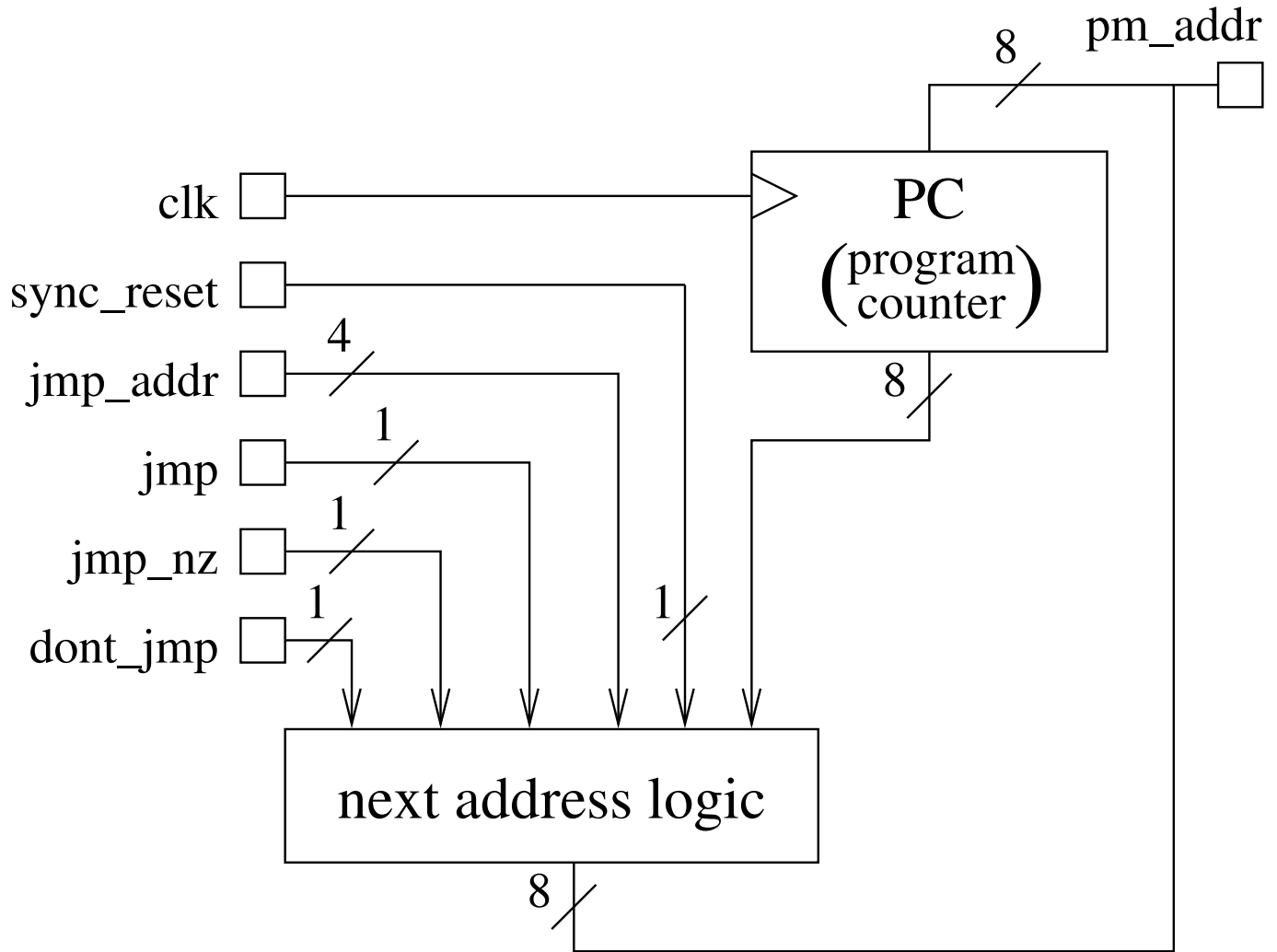
1. The pins, i.e. ☐, denote the signals in the port list of the instruction decoder module.
2. While sync_reset is high: reg_en must be 9'H1FF, source_sel must be 4'd10, i_sel, x_sel and y_sel must be 1'b0, jmp and jmp_nz must be 1'b0

TABLE 1: Assignments for reg_en[8:0]

reg_en								
[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
o_reg	dm	i	m	r	y1	y0	x1	x0

Microprocessor page 3/5

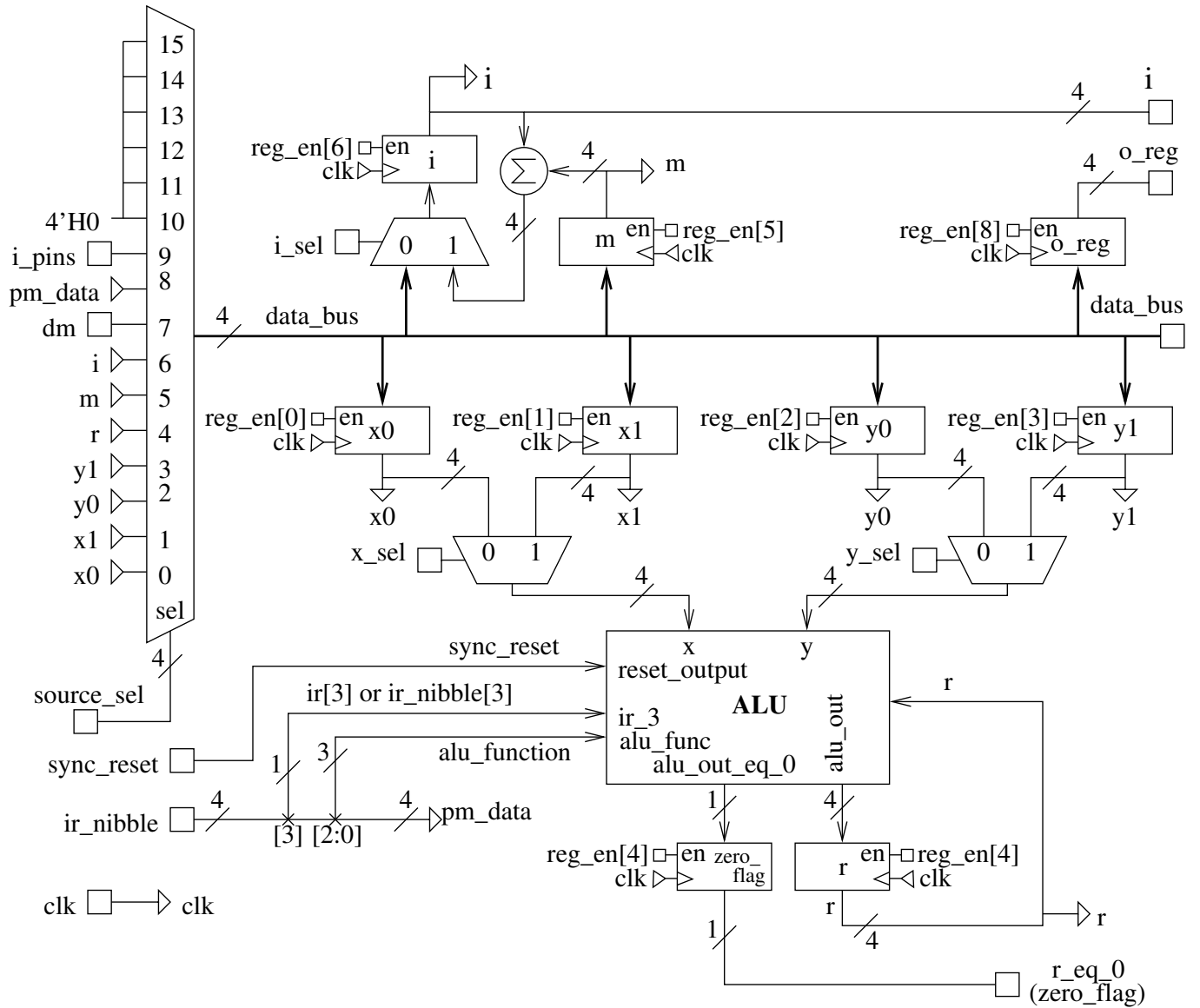
Block Diagram of Instruction Decoder



NOTES: The pins, i.e. ☐, denote signals from the port list of the program sequencer module.

Microprocessor page 4/5

Block diagram of program sequencer



- NOTES:**
1. The pins, i.e. ☐, denote signals in the port list of the computational unit module
 2. While sync_reset is high, alu_out must be 4'H0 and alu_out_eq_0 must be 1'b1

1.4 Example

This section uses an example to cement the operation of program memory, the program sequencer, the instruction decoder and the computational unit. The instruction sequence for this example is

```

address instruction
8'H00  jnz 20H;  sync_reset has set the
          ; zero flag so don't jump
8'H01  x0 = 4'H7;
8'H02  y0 = 4'H9;
8'H03  r = x0 + y0;  r gets 0, zero flag gets 1
8'H04  m = 4'H1;
8'H05  r=~x0; r gets 8 (ones complement of 7),
          ; zero flag gets 0
8'H06  dm[i] = r; dm[0] gets 8,
          ; i gets i+m = 0+1 = 1
8'H07  jnz 8'H20; zero flag is 0 so jump to
          ; address 8'H20

8'H20  dm[i] = y0; dm[1] gets 9,
          ; i gets i+m = 1+1 = 2
8'H21  m = -1; m gets two's complement of 1,
          ; i.e. m gets F
8'H22  dm[i] = i_pins; dm[2] gets i_pins
          ; (the value of i_pins at
          ; the time the instruction
          ; is executed),
          ; i=i+m=2+(-1)
8'H23  i = dm[i]; i gets dm[1], i.e. i gets 9
8'H24  o_reg = dm[i]; o_reg gets dm[9],
          ; since dm[9] was not written
          ; dm[9] has value 0.
          ; i.e. o_reg gets 0,
          ; i=i+m = 9+(-1) = 8
8'H25  o_reg = r; o_reg gets 8
8'H26  jump 8'H30; jump to address 8'H30

8'H30  jump 8'H30; trap at address 8'H30

```

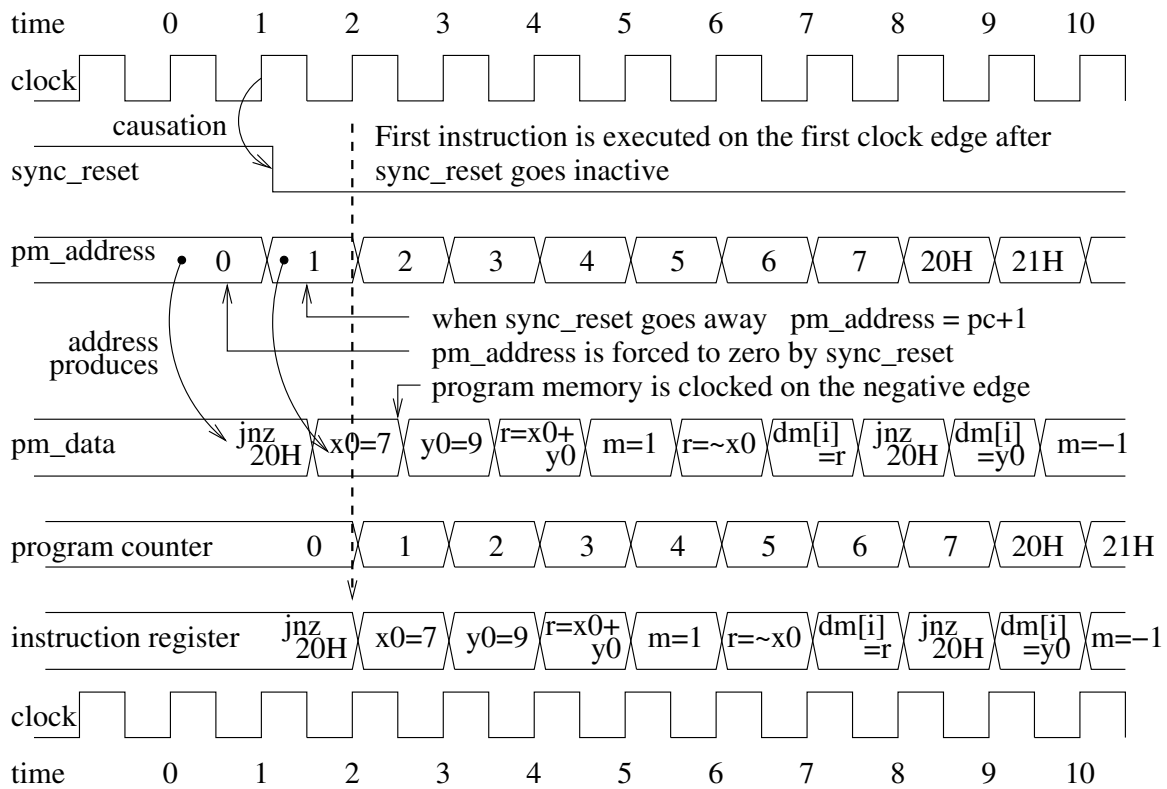


Figure 1: Timing for instructions flowing through program memory

Table 1: Template for the construction of machine code from the instruction

[illegible]

Table 2: Template for calculating pm_address from instruction sequence

NOTE: snyc_reset = 1'b0

value of PC	instruction in instruction reg.	x0 reg	r reg	zero flag	pm_address
8'H00	jnz 8'H20	4'H0	4'H0	1'b1	
8'H01	x0 = 4'H7	4'H0			
8'H02	y0 = 4'H9	4'H7			
8'H03	r = x0 + y0				
8'H04	m = 1				
8'H05	r = ~x0				
8'H06	dm[i] = r				
8'H07	jnz 8'H20				
8'H20	dm[i] = y0				
8'H21	m = -1				
8'H22	dm[i] = ipins				
8'H23	i = dm[i]				
8'H24	o_reg = dm[i]				
8'H25	o_reg = r				
8'H26	jump 8'H30				
8'H30	jump 8'H30				

Table 3: Template for calculating *ir* outputsNOTE: `snyc_reset = 1'b0`

value of PC	instruction in instruction reg.	source select	reg_enables (9 bits)	selects			cond jump	jump
				x	y	i		
8'H00	jnz 8'H20	4'HX	0_0000_0000	X	X	X	1	0
8'H01	x0 = 4'H7	4'H8		X	X	X	0	0
8'H02	y0 = 4'H9							
8'H03	r = x0 + y0							
8'H04	m = 1		0_0010_0000					
8'H05	r = ~x0							
8'H06	dm[i] = r							
8'H07	jnz 8'H20							
8'H20	dm[i] = y0							
8'H21	m = -1							
8'H22	dm[i] = ipins							
8'H23	i = dm[i]							
8'H24	o_reg = dm[i]							
8'H25	o_reg = r							
8'H26	jump 8'H30							
8'H27	jump 8'H30							

Table 4: Template for tracking data through the computational unit

NOTE: snyc_reset = 1'b0

value of PC	instruction in instruction reg.	data bus	x0	y0	m	i	r	zero flag
8'H00	jnz 8'H20	4'HX	4'H0	4'H0	4'H0	4'H0	4'H0	1'b1
8'H01	x0 = 4'H7							
8'H02	y0 = 4'H9							
8'H03	r = x0 + y0							
8'H04	m = 1							
8'H05	r = ~x0							
8'H06	dm[i] = r							
8'H07	jnz 8'H20							
8'H20	dm[i] = y0							
8'H21	m = -1							
8'H22	dm[i] = ipins							
8'H23	i = dm[i]							
8'H24	o_reg = dm[i]							
8'H25	o_reg = r							
8'H26	jump 8'H30							
8'H30	jump 8'H30							

Table 5: Solution for the construction of machine code from the instruction

addr	instruction	type	dst	src	x sel	y sel	func	data / j. addr	machine code (word store in memory)	
8'H00	jnz 8'H20	cond jmp						4'H2	8'b1111_0010	8'HF2
8'H01	x0 = 4'H7	load	3'H0					4'H7	8'b0_000_0111	8'H07
8'H02	y0 = 4'H9	load	3'H2					4'H9	8'b0_010_1001	8'H29
8'H03	r = x0 + y0	alu			0	0	3'H2		8'b110_0_0_010	8'HC2
8'H04	m = 1	load	3'H5					4'H1	8'b0_101_0001	8'H51
8'H05	r = ~x0	alu			0	0	3'H7		8'b110_0_0_111	8'HC7
8'H06	dm[i] = r	move	3'H7	3'H4					8'b10_111_100	8'HBC
8'H07	jnz 8'H20	cond jmp						4'H2	8'b1111_0010	8'HF2
8'H20	dm[i] = y0	move	3'H7	3'H2					8'b10_111_010	8'HBA
8'H21	m = -1	load	3'H5					4'HF	8'b0_101_1111	8'H5F
8'H22	dm[i] = ipins	move	3'H7	3'H7					8'b10_111_111	8'HBF
8'H23	i = dm[i]	move	3'H6	3'H7					8'b10_110_111	8'HB7
8'H24	o_reg = dm[i]	move	3'H4	3'H7					8'b10_100_111	8'HA7
8'H25	o_reg = r	move	3'H4	3'H4					8'b10_100_100	8'HA4
8'H26	jump 8'H30	jump						4'H3	8'b1110_0011	8'HE3
8'H30	jump 8'H30	jump						4'H3	8'b1110_0011	8'HE3

Table 6: Solution for calculating pm_address from instruction sequence
NOTE: sync_reset = 1'b0

value of PC	instruction in instruction reg.	x0 reg	r reg	zero flag	pm_address
8'H00	jnz 8'H20	4'H0	4'H0	1'b1	PC+1
8'H01	x0 = 4'H7	4'H0	4'H0	1'b1	PC+1
8'H02	y0 = 4'H9	4'H7	4'H0	1'b1	PC+1
8'H03	r = x0 + y0	4'H7	4'H0	1'b1	PC+1
8'H04	m = 1	4'H7	4'H0	1'b1	PC+1
8'H05	r = ~x0	4'H7	4'H0	1'b1	PC+1
8'H06	dm[i] = r	4'H7	4'H8	1'b0	PC+1
8'H07	jnz 8'H20	4'H7	4'H8	1'b0	8'H20
8'H20	dm[i] = y0	4'H7	4'H8	1'b0	PC+1
8'H21	m = -1	4'H7	4'H8	1'b0	PC+1
8'H22	dm[i] = ipins	4'H7	4'H8	1'b0	PC+1
8'H23	i = dm[i]	4'H7	4'H8	1'b0	PC+1
8'H24	o_reg = dm[i]	4'H7	4'H8	1'b0	PC+1
8'H25	o_reg = r	4'H7	4'H8	1'b0	PC+1
8'H26	jump 8'H30	4'H7	4'H8	1'b0	8'H30
8'H27	jump 8'H30	4'H7	4'H8	1'b0	8'H30

Table 7: Solution for mapping instructions to *ir* outputsNOTE: `snyc_reset = 1'b0`

value of PC	instruction in instruction reg.	source select	reg_enables (9 bits)	selects			cond jump	jump
				x	y	i		
8'H00	jnz 8'H20	4'HX	0_0000_0000	X	X	X	1	0
8'H01	x0 = 4'H7	4'H8	0_0000_0001	X	X	X	0	0
8'H02	y0 = 4'H9	4'H8	0_0000_0100	X	X	X	0	0
8'H03	r = x0 + y0	4'HX	0_0001_0000	0	0	X	0	0
8'H04	m = 1	4'H8	0_0010_0000	X	X	X	0	0
8'H05	r = ~x0	4'HX	0_0001_0000	0	0	X	0	0
8'H06	dm[i] = r	4'H4	0_1100_0000	X	X	1	0	0
8'H07	jnz 8'H20	4'HX	0_0000_0000	X	X	X	1	0
8'H20	dm[i] = y0	4'H2	0_1100_0000	X	X	1	0	0
8'H21	m = -1	4'H8	0_0010_0000	X	X	X	0	0
8'H22	dm[i] = ipins	4'H9	0_1100_0000	X	X	1	0	0
8'H23	i = dm[i]	4'H7	0_0100_0000	X	X	0	0	0
8'H24	o_reg = dm[i]	4'H7	1_0100_0000	X	X	1	0	0
8'H25	o_reg = r	4'H4	1_0000_0000	X	X	X	0	0
8'H26	jump 8'H30	4'HX	0_0000_0000	X	X	X	0	1
8'H27	jump 8'H30	4'HX	0_0000_0000	X	X	X	0	1

Table 8: Solution for tracking the data movement in the computational unit

NOTE: snyc_reset = 1'b0

value of PC	instruction in instruction reg.	data bus	x0	y0	m	i	r	zero flag
8'H00	jnz 8'H20	4'HX	4'H0	4'H0	4'H0	4'H0	4'H0	1'b1
8'H01	x0 = 4'H7	4'H7	↓	↓	↓	↓	↓	↓
8'H02	y0 = 4'H9	4'H9	4'H7	↓	↓	↓	↓	↓
8'H03	r = x0 + y0	4'HX	↓	4'H9	↓	↓	↓	↓
8'H04	m = 1	4'H1	↓	↓	↓	↓	4'H0	1'b1
8'H05	r = ~x0	4'HX	↓	↓	4'H1	↓	↓	↓
8'H06	dm[i] = r	4'H8	↓	↓	↓	↓	4'H8	1'b0
8'H07	jnz 8'H20	4'HX	↓	↓	↓	4'H1	↓	↓
8'H20	dm[i] = y0	4'H9	↓	↓	↓	↓	↓	↓
8'H21	m = -1	4'HF	↓	↓	↓	4'H2	↓	↓
8'H22	dm[i] = ipins	4'H?	↓	↓	4'HF	↓	↓	↓
8'H23	i = dm[i]	4'H?	↓	↓	↓	4'H1	↓	↓
8'H24	o_reg = dm[i]	4'H9	↓	↓	↓	4'H9	↓	↓
8'H25	o_reg = r	4'H8	↓	↓	↓	4'H8	↓	↓
8'H26	jump 8'H30	4'HX	↓	↓	↓	↓	↓	↓
8'H30	jump 8'H30	4'HX	↓	↓	↓	↓	↓	↓

Notes on Compiler Directives
in file `notes_on_compiler_directives.pdf`

Altera's Notes on Synthesis Directives in Verilog 2001
in file `ALTERA_notes_on_synthesis_directives.pdf`

Notes on the CME341 cross assembler
in file `notes_on_the_CME341_crossassembler.pdf`

2 Preparation for the Second Midterm

This section was added on Oct. 30, 2018.

Students must modify a few of the microprocessor modules and set up both Quartus and Modelsim projects prior to entering the exam. The step-by-step instructions for what must be done prior to the second midterm are given below.

Modify the prototype of your instruction decoder to have an 8-bit output called `from_ID`.

This output should be of type `reg` and constructed to be the constant `8'H00`. This signal is set up as a conduit from the instruction decoder to the testbench.

Signals constructed in the instruction decoder during the course of the exam need to be connected to the testbench in order for them to affect the output code. The new signals designed during the course of the exam are “channelled” to the testbench by connecting them to `from_ID` inside the instruction decoder. However, since no new signals are constructed in the preamble, the signal `from_ID` is to be hardwired to the constant `8'H00`.

Modify the prototype of your program sequencer to have an 8-bit output called `from_PS` and set it to the constant `8'H00`. This is done for the same reasons as above.

After making the modifications, place copies of your prototypes for the program sequencer, program memory and instruction decoder in a separate folder on their H drive - say a folder called `copy_of_prototypes`. These files are the starting points of all questions.

Prepare a Quartus project and a Modelsim-Altera project in accordance with the instructions below.

1. Make a folder on the H drive called `second_midterm_quartus`.
2. Copy the files in folder `copy_of_prototypes` folder `second_midterm_quartus`. Also copy the following files (found on the class website) `second_midterm_quartus.v`, `second_midterm_quartus.sdc`, `program_memory_preamble.hex` and `second_midterm_preamble.lst` to the same folder.
3. Make a Quartus project called `second_midterm_quartus` and place it in the folder `second_midterm_quartus`.
4. Make the top module for the project `second_midterm_quartus`. It is contained in file `second_midterm_quartus.v`, which is given on the class website.
5. The Verilog HDL module `second_midterm_quartus`, which is included after immediately following this section on page 44, instantiates three prototypes that were constructed in assignments 6, 7 and 8. Notice that it channels the new signals created in the instruction decoder and program sequencer, i.e. `from_ID` and `from_PS`, from the instantiations to its output.

The file `second_midterm_quartus.v` found among the preamble files contains this module.

6. The `.sdc` (synopsys design constraints) file for the project is also provided. It may be incompatible with the current version of Quartus in the labs. In which case generate `.sdc` file for `clk` being 1 MHz.

7. Modify `second_midterm_quartus.v` so that the connection lists in the instantiations match the port lists for your prototypes.
8. Use the hex file `program_memory_preamble.hex` to initialize the program memory. It is pointed out that the first time this file is used in a compile the compiler will generate a warning indicating the .hex file does not fill the entire memory. That warning is accurate, but can be ignored.
9. After successfully compiling `second_midterm_quartus` set up a Modelsim-Altera project called `second_midterm_testbench` in the directory where Quartus places `second_midterm_quartus.vo`, i.e. in subdirectory `simulation/modelsim`.
10. The test bench, which is also called `second_midterm_testbench`, is provided by the instructor in file `second_midterm_testbench.v`. It is also on the class website. The testbench module is appended immediately following the prototype module `second_midterm_quartus`. It starts at the top of page 46.
11. Open `second_midterm_testbench.v` and observe that it instantiates `second_midterm_quartus` so both `second_midterm_quartus.vo` and `second_midterm_testbench.v` must be included in the project.

Also verify that `exam_dependent_seed` is set to 8'HFF.

12. Download files `wave.do` to your Modelsim-Altera project folder.
13. Compile `second_midterm_testbench.v` and load the simulation.
14. Down load “`second_midterm_preamble.do`” and execute the command “do `second_midterm_preamble.do`” by typing it into the command line in the transcript window. This command will set up the wave window.
15. Waveform data from a correctly operating circuit was saved in the file `second_midterm_preamble` which is one of the preamble files for the second midterm. Normally this data file could be loaded as data in Modelsim and then displayed in a second Modelsim wave window, but the free version of Modelsim, which is what we have, does not allow this. The free version only allows one wave window to be open.

The workaround is to create a second Modelsim project in a second folder. The second project can have any name, which could be `gold_waveforms_for_second_midterm`. It needs to have just two files in it. The data file containing the information in the waveforms, which is `second_midterm_preamble.wlf`, and the command script that transfers the data to the wave window, which is `second_midterm_preamble.do`. These files can be found among the other preamble files.

The waveform data in file `second_midterm_preamble.wlf` is viewed in the wave window by issuing two commands in the command line of the Modelsim transcript window. However, it must be emphasized for the commands to work **the wave window must be closed** before the commands are issued. The two commands are:

```
vsim -view gold=second_midterm_preamble.wlf
do second_midterm_preamble.do
```

The first command opens a data set and assigns it a name, which could be anything, but the command above names the data set `gold`. The second command invokes the `.do` file `second_midterm_preamble.do`. The commands in this script displays the data set just named in the wave window.

Once the data is displayed in the wave window it can be manipulated with the display tools, which of course includes the zoom.

OPTIONAL: Procedure for verifying the pre-exam preparation

To make absolutely sure that `from_PS` and `from_ID` are properly connected all the way through to the test bench follow the sequence of steps below.

1. Verification of the conduit that takes `from_PS` from inside the program sequencer to the test bench begins by rewiring `from_PS` inside the program sequencer so that it is connected to the ones complement of the output of register `pc`, i.e. `from_PS = ~ pc`.

After rewiring `from_PS` recompile the Quartus project and after that recompile and reload the simulation in Modelsim-Altera. If the conduit that uses `from_PS` has been properly established `accumulator_output` should read **16'H1BF0** at 300 μ s and **16'HCC85** at 620 μ s

2. To verify the conduit that uses `from_ID` first rewire `from_PS` back to the constant `8'H00`, i.e. `from_PS = 8'H00`. Then, from inside the instruction decoder, rewire `from_ID` so that it is connected to the output of register `ir`, i.e. `from_ID = ir`.

Then recompile in Quartus and Modelsim. If conduit that uses `from_ID` has been properly established `accumulator_output` should read **16'HD3E7** at 300 μ s and **16'HB832** at 620 μ s

3. **Remember to rewire `from_ID` back to the constant `8'H00`.**

The top module for the Quartus project to be used in the second midterm is listed below.

```

module second_midterm_quartus  (
input clk, reset,  zero_flag,
output reg sync_reset,
output wire [7:0] pm_address, pm_data,
output wire jump, conditional_jump,
           x_mux_select, y_mux_select, i_mux_select,
output wire [3:0] source_register_select, LS_nibble_of_ir,
output wire [8:0] register_enables,
output wire [7:0] pc, instr_register,
output wire [7:0] from_ID, from_PS // conduits from prog sequencer and
                                   // and instruction decoder to testbench
                                   // used in second midterm and final exam
);

always @ (posedge clk)
sync_reset = reset;

program_memory prog_memory(
.address(pm_address),
.clock(~clk),
.q(pm_data));

program_sequencer prog_sequencer (
.clk(clk),
.sync_reset(sync_reset),
.dont_jump_flag(zero_flag),
.jump(jump),
.conditional_jump(conditional_jump),
.jump_addr(LS_nibble_of_ir),
.pm_address(pm_address),
.pc(pc), //pc is taken out for purposes of debugging
.from_PS(from_PS) // conduit to testbench for exams
);

instruction_decoder inst_decoder(
.clk(clk),
.sync_reset(sync_reset),
.pm_data(pm_data),
.jump(jump),
.conditional_jump(conditional_jump),
.LS_nibble_of_ir(LS_nibble_of_ir),
.i_mux_select(i_mux_select),
.y_mux_select(y_mux_select),
.x_mux_select(x_mux_select),

```

```
        .source_register_select(source_register_select),  
        .register_enables(register_enables),  
        .ir(instr_register), // ir is for purposes of debugging  
        .from_ID(from_ID) // conduit to testbench for exams  
    );  
  
endmodule
```

The Verilog HDL for the test bench used in the second midterm is listed below.

```
'timescale 1us/1ns
module second_midterm_testbench ();
reg clk, reset;
wire sync_reset;
reg [7:0] seed, exam_dependent_seed;
reg [7:0] stimulus;
reg counter_full_bar;
reg [15:0] accumulator_output;
reg [7:0] scrambled_output;
reg [7:0] adder_output;
reg [15:0] rotator_output;
reg zero_flag;
wire [7:0] student_scrambler_output;
wire [7:0] pm_address, pm_data;
wire jump, conditional_jump, x_mux_select, y_mux_select, i_mux_select;
wire [3:0] source_register_select, LS_nibble_of_ir;
wire [8:0] register_enables;
wire [7:0] pc, instr_register;
wire [7:0] from_PS, from_ID;

// define the length of the simulation
initial #640 $stop;

// set the seed associated with the exam
initial exam_dependent_seed = 8'HFF;

// seed used in simulation
initial
    begin seed = 8'HAA;
        #320 seed = exam_dependent_seed;
    end

// make the zero flag
initial zero_flag = 1'b0;
always @ *
    zero_flag = stimulus[7] & stimulus[5];

// generate reset and sync_reset
initial    begin reset = 1'b1; #5.2 reset = 1'b0; end
initial    begin #9.2 reset = 1'b1; #1 reset = 1'b0; end
initial    begin #100.1 reset = 1'b1; #0.2 reset = 1'b0; end
initial    begin #320 reset = 1'b1; #5.2 reset = 1'b0; end
initial    begin #329.2 reset = 1'b1; #1 reset = 1'b0; end
initial    begin #420.1 reset = 1'b1; #0.2 reset = 1'b0; end
```

```

// make the clock
initial clk = 1'b0;
always #0.5 clk = ~clk; // 1 MHz clock

//design the accumulator
always @ (posedge clk) // make hold time 0.01 microseconds
if (sync_reset == 1'b1)
    accumulator_output <= #0.01 8'b0;
else if (counter_full_bar == 0)
    accumulator_output <= #0.01 accumulator_output;
else
    accumulator_output <= #0.01 rotator_output;

//design the counter
always @ (posedge clk) // make hold time 0.01 microseconds
if (sync_reset == 1'b1)
    stimulus <= #0.10 8'b0;
else if (counter_full_bar == 0)
    stimulus <= #0.01 stimulus;
else
    stimulus <= #0.01
        stimulus + 8'b1;

//design the rotator
always @ *
rotator_output = {accumulator_output[14:8],
    adder_output, accumulator_output[15]};

//design the adder
always @ *
adder_output = accumulator_output[7:0]+scrambled_output;

// design the counter full dectector
always @ *
if (& stimulus == 1'b1) // counter is full
    counter_full_bar = 1'b0;
else    counter_full_bar = 1'b1;

/* make the scrambler */
always @ *
scrambled_output = seed ^ pm_address ^ pc
    ^ register_enables[7:0]
    ^ {5'b0, jump, conditional_jump, register_enables[8]}
    ^ from_PS ^ from_ID;

/* *****

```

```
    instantiate the .vo file
    that resulted from the student
    design
    *****/

second_midterm_quartus exam_quartus_1 (
    .clk(clk),
    .reset(reset),
    .sync_reset(sync_reset),
    .zero_flag(zero_flag),
    .pm_address(pm_address),
    .pm_data(pm_data),
    .jump(jump),
    .conditional_jump(conditional_jump),
    .x_mux_select(x_mux_select),
    .y_mux_select(y_mux_select),
    .i_mux_select(i_mux_select),
    .source_register_select(source_register_select),
    .LS_nibble_of_ir(LS_nibble_of_ir),
    .register_enables(register_enables),
    .pc(pc),
    .instr_register(instr_register),
        .from_PS(from_PS), // conduit from prog sequencer
        .from_ID(from_ID) // conduit from instr decoder
    );
endmodule
```


The second midterm will have several questions and valuable time can be saved by using one project to answer all of the questions. There are many ways in which this can be done. Probably the most straight forward is to comment out the instantiations for a question immediately after the question has been completed. The procedure for this “comment-out-and-go” approach is given below:

1. Start with the top module for the preamble.
2. Comment out the instantiation of the three modules used in the preamble.
3. At the end of the instantiations that were just commented out make the banner below as a comment:

```

        /* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        Instantiations for
        Question 1

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */

```

4. Copy the three instantiations made in the preamble, i.e. the three instantiations that were just commented out, and paste them after the “Question 1” banner.
5. Decide which of the three modules being instantiated have to be changed to complete Question 1. Then change the names of those modules by appending `_Q1`.
6. If necessary modify the connection lists for the instantiations of the modules (i.e. the prototypes) that have to be changed.
7. Open a `.v` file containing one of the modules that needs to be changed and then immediately save it with “`_Q1`” appended. For example, if completing Question 1 required the instruction decoder to be changed, then open `instruction_decoder.v` and save it as `instruction_decoder_Q1.v`.
8. Change the name of the module in by `instruction_decoder_Q1.v` appending `_Q1` to make it the same as the file name. For example, change the module declaration of `module instruction_decoder` of `module instruction_decoder_Q1`.
9. Make changes to the renamed modules as necessary to answer Question 1.
10. If changes need to be made to a second module to answer Question 1, then follow the same procedure to make the changes.
11. After all the modifications are made, make a copy of the `.hex` file for Questions 1 and rename the copy to be that of the initialization file for the ROM used for program memory.
12. Compile the project and debug as necessary to complete the question.

13. Start Question 2 by first commenting out all of the instantiations for Questions 1.
14. Make the banner for Question 2 and proceed using the process that was used for question 1.

The procedure above is illustrated with an example Verilog HDL of a top module suitable for the second midterm. The example Verilog HDL begins at the top of page 51.

Note the instantiations for Question 1 in the example HDL show a new 8-bit signal named `new_signal` runs between the program sequencer and the instruction decoder. This new signal must be declared an 8-bit wire in the top module. The declaration is best made in the location shown in the example Verilog HDL, which is immediately after the banner for Question 1. Locating the declaration after the banner allows it to be commented out in the same block comment as the instantiations for Question 1.

```

module second_midterm_quartus (
input clk, reset,  zero_flag,
output reg sync_reset,
output wire [7:0] pm_address, pm_data,
output wire jump, conditional_jump,
           x_mux_select, y_mux_select, i_mux_select,
output wire [3:0] source_register_select, LS_nibble_of_ir,
output wire [8:0] register_enables,
output wire [7:0] pc, instr_register,
output wire [7:0] from_ID, from_PS // conduits from prog sequencer and
                                   // and instruction decoder to testbench
                                   // used in second midterm and final exam
);

always @ (posedge clk)
sync_reset = reset;

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Preamble
instantiations

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */

/* comment out preamble

program_memory prog_memory(
.address(pm_address),
.clock(~clk),
.q(pm_data));

program_sequencer prog_sequencer (
.clk(clk),
.sync_reset(sync_reset),
.dont_jump_flag(zero_flag),
.jump(jump),
.conditional_jump(conditional_jump),
.jump_addr(LS_nibble_of_ir),
.pm_address(pm_address),
.pc(pc), //pc is taken out for purposes of debugging
.from_PS(from_PS) // conduit to testbench for exams
);

instruction_decoder inst_decoder(
.clk(clk),
.sync_reset(sync_reset),

```

```

        .pm_data(pm_data),
        .jump(jump),
        .conditional_jump(conditional_jump),
        .LS_nibble_of_ir(LS_nibble_of_ir),
        .i_mux_select(i_mux_select),
        .y_mux_select(y_mux_select),
        .x_mux_select(x_mux_select),
        .source_register_select(source_register_select),
        .register_enables(register_enables),
        .ir(instr_register), // ir is for purposes of debugging
        .from_ID(from_ID) // conduit to testbench for exams
    );

end of preamble %/

/% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Question 1
instantiations

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% %/

wire [7:0] new_signal; % make wires for the new signal
                    % running between the instruction decoder
                    % and the program sequencer

program_memory prog_memory(
    .address(pm_address),
    .clock(~clk),
    .q(pm_data));

program_sequencer_Q1 prog_sequencer (
    .new_signal(new_signal),
    .clk(clk),
    .sync_reset(sync_reset),
    .dont_jump_flag(zero_flag),
    .jump(jump),
    .conditional_jump(conditional_jump),
    .jump_addr(LS_nibble_of_ir),
    .pm_address(pm_address),
    .pc(pc), //pc is taken out for purposes of debugging
    .from_PS(from_PS) // conduit to testbench for exams
);

instruction_decoder_Q1 inst_decoder(
    .new_signal(new_signal)
    .clk(clk),

```

```
.sync_reset(sync_reset),  
.pm_data(pm_data),  
.jump(jump),  
.conditional_jump(conditional_jump),  
.LS_nibble_of_ir(LS_nibble_of_ir),  
.i_mux_select(i_mux_select),  
.y_mux_select(y_mux_select),  
.x_mux_select(x_mux_select),  
.source_register_select(source_register_select),  
.register_enables(register_enables),  
.ir(instr_register), // ir is for purposes of debugging  
.from_ID(from_ID) // conduit to testbench for exams  
);
```

```
endmodule
```

3 Making Tristate Buses and Tristate Pins

Most FPGAs have the capability to configure the input/output (I/O) pins as either input pins, output pins or bidirectional pins (inout pins). In addition, each of the I/O pins can be programmed with things like: a open collector output, a pull up resistor, a bus hold, and a delay.

The I/O pin together with surrounding registers, logic and delay elements is called an Input/Output Element (IOE). The IOEs are quite different than the Logic Elements (LEs). The IOEs are constructed with bigger transistors and powered with a higher voltage. This gives the IOEs much more drive capability, but this comes at the expense of propagation delay.

A schematic diagram for an IOE is given in Figure 2. The element in the IOE that is key to providing bidirectional pins is the tristate driver/buffer. Each IOE has but one tristate buffer. It is located midway up and a little right of center in Figure 2. This tristate buffer either acts like a bistate buffer by driving its output to make it mimic the input or like an open switch. When it acts like an open switch its Thevenin equivalent is a high impedance in series with a voltage source that is unspecified, but the voltage is certain to be between 0 and V_{cc} volts.

When the output enable is active the tristate buffer is a bus-driver buffer and when the output enable is inactive it is an open switch. The output enable for the tristate buffer shown in the IOE of Figure 2 is active low. The IOE can be programmed so that the enable is registered (top register in Figure 2) or not.

The tristate buffers in the IOEs are the only tristate logic devices in the entire FPGA. The logic internal to the FPGA does not contain any tristate devices. This means I/O pins programmed as tristate must be converted to/from bistate signals before they can be connected to the logic in the core of the FPGA. The Verilog HDL has to be written in way that the compiler knows the intent is to interface a tristate pin (i.e. an inout pin) to two bistate logic signals.

Bidirectional pins are used to support two directional communication among several devices on a single tristate bus. Such buses greatly reduce the number of tracks that connect the devices. While the tristate bus can transfer information in either direction, the information can not be transferred in both directions at the same time.

A Printed Circuit Board (PCB) mounted with devices that communicate over a tristate bus is illustrated in Figure 3. In this figure the bus is controlled by a master device and all communication is between the master and one of the other devices, which are referred to as the slave devices.

When the system of Figure 3 is modelled in Modelsim-Altera the tracks on the printed circuit board become wires in Modelsim-Altera. Wires are used to connect all types of ports, **including inout ports**. The master and slave devices are synthesized in Quartus and the resulting .vo file/files is/are instantiated in the Modelsim-Altera test bench.

The operation is explained through an example. If the master device wants to transfer a word from a slave device to itself (i.e. read a word from a slave device) it does the following:

1. The master activates **read**, deactivates **write** and puts the address of the slave on **slave_addr**. All of these signal are bistate with power flowing in one directional, which is from the master to the slaves.

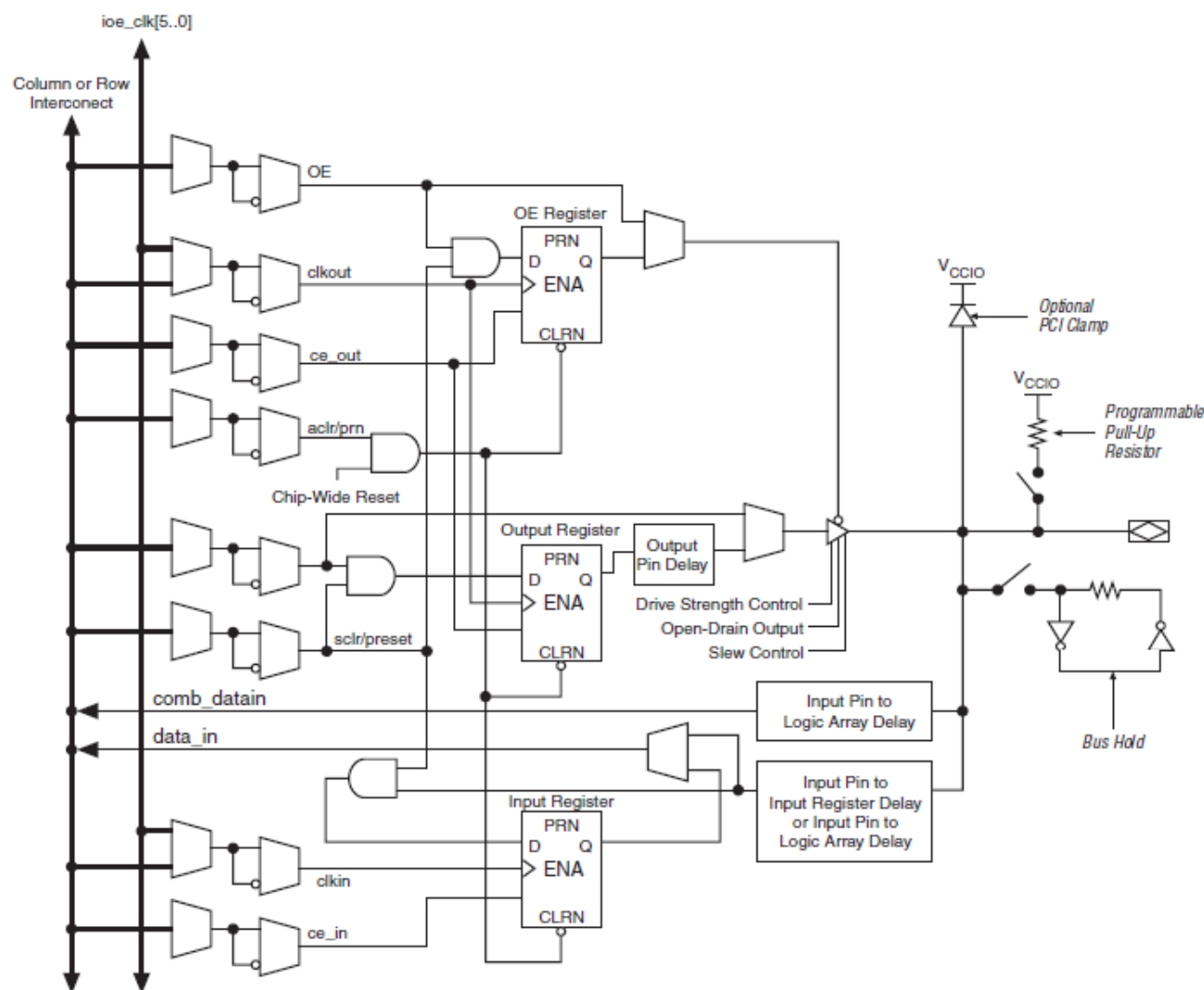
Figure 2–32. Cyclone IOE in Bidirectional I/O Configuration

Figure 2: Schematic diagram for a Cyclone I/O element (IOE)

2. Coincidentally the master puts its tristate bus driver into high impedance state.
3. The slave recognizes its address and enables itself. No other slave will be enabled.
4. After the slave recognizes its address, it enables its tristate driver and puts its data onto the bidirectional tristate bus.
5. After the bidirectional bus is stable the master will clock the data that the slave is

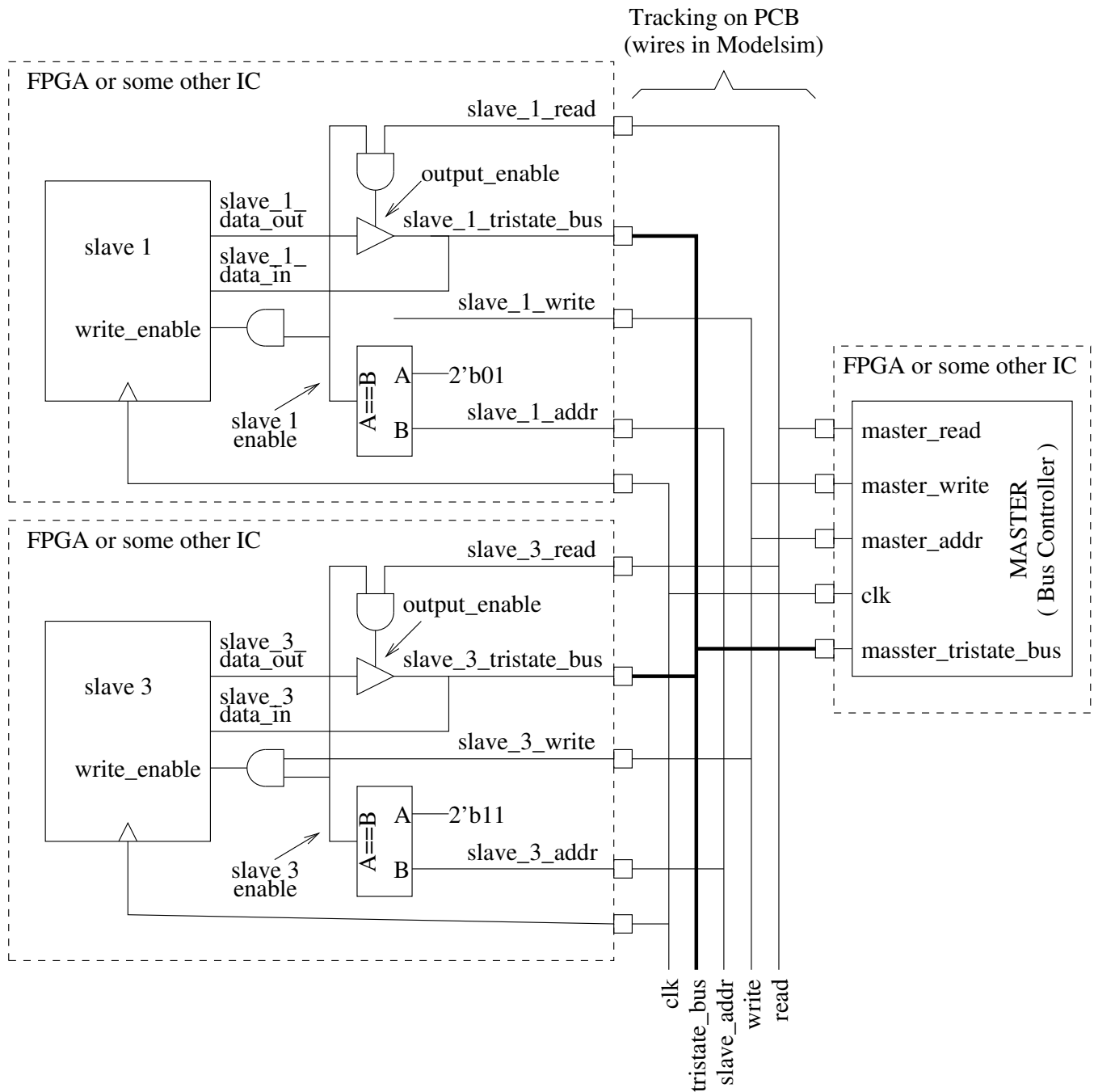


Figure 3: A block diagram of devices on a PCB connected with a tristate bus

putting on the bus into its input register.

If the master device wants to transfer a word from itself to a slave device (i.e. write a word to a slave) it does the following:

1. The master deactivates `read`, activates `write` and puts the address of the slave on `slave_addr`.
2. Coincidentally the master enables its tristate driver to put the data it wishes to transfer to the slave on the tristate bus.
3. The slave recognizes its address and enables itself. No other slave will be enabled.
4. After the slave recognizes its address, it enables its input register to load the data on the tristate bus on the next rising clock edge.

The Verilog HDL does not have a single construct to interface bistate signals with tristate signals. The tristate IO pin has to be connected to two bistate logic signals as shown in Figure 3. The bistate signal `slave_1_data_out` drives the tristate pin `state_1_tristate_bus` at the request of the master and `slave_1_data_in` provides the value on the tristate bus to the core of the FPGA.

Since `slave_1_tristate_bus` is a pin that connects to the tristate bus on the printed circuit board it must appear in the port list for the slave 1 the prototype. To be a tristate pin it must be declared “inout”.

A pin is configured to be either an input, an output or an inout pin by programming its IOE. The pin is made an “input” by permanently disabling the tristate buffer to make it an open switch. The pin is made an “output” by permanently enabling the tristate buffer to make it an ordinary bistate buffer. The pin is made an inout pin by connecting the enable for the tristate buffer to a signal that controls when the pin is to drive the bus. If the tristate buffer is enabled then the pins drives the bus. Otherwise the bus drives the pin.

The Verilog HDL that creates an interface between the binary logic in the core of FPGA to the IOE requires two conditional assignments. One that interfaces the bistate signal that, upon request, drives the tristate bus. A second that interfaces the tristate bus to bistate signal that can be used by the bistate logic internal to core of the FPGA. The two conditional assignments are given below:

```
inout slave_1_tristate_bus;
wire slave_1_data_in;
assign slave_1_tristate_bus = ( slave_1_tristate_buffer_enable )
                             ? slave_1_data_out : 8'bz;
assign slave_1_data_in = (slave_1_tristate_buffer_enable)
                         ? slave_1_data_out : slave_1_tristate_bus;
```

The pin that connects to the tristate bus is called `slave_1_tristate_bus`. The internal signal that is to drive the tristate bus is named `slave_1_data_out`. The binary signal that reads the tristate bus is called `slave_1_data_in`.

The input signal `slave_1_data_in` is a binary signal so whenever the bus is in a high impedance state (i.e. when the bus is not driven by either the master or any of the slaves), its state, which can be either zero or one, will depend on the noise on the bus. However, the

simulator will show `slave_1_data_in` as being undefined when the tristate bus is in a high impedance state.

The structure of the two conditional assignments is straight forward. The “test condition” in both conditional assignments must be true when the slave is to drive the bus. One conditional assignment must be written to assign `slave_1_data_out` to the tristate bus when the test condition is true and assign high impedance when it is false. The second conditional assignment assigns the binary input signal, which is `slave_1_data_in`, the value of `slave_1_data_out` when the test condition is true and the value of `slave_1_tristate_bus` when it is false.

While perhaps obvious, it is pointed out that when the first conditional assignment assigns “high impedance” to `slave_1_tristate_bus`, the pin will likely be driven by the tristate bus and therefore not be high impedance.

The test bench for the system shown in Figure 3 is given below.

```

`timescale 1 us / 1 ns;
module testbench_tristate_bus ();
    wire [7:0] masters_input_register;
    wire [7:0] tristate_bus;
    wire [1:0] slave_addr;
    wire read, write;
    reg clk;

    initial #17 $stop;

    initial clk = 1'b0;
    always #0.5 clk = ~clk;

    tristate_buses instance_1(
        .clk(clk),
        .masters_input_register(masters_input_register),
        .master_tristate_bus(tristate_bus),
        .master_addr(slave_addr),
        .master_read(read),
        .master_write(write),
        .slave_1_tristate_bus(tristate_bus),
        .slave_1_addr(slave_addr),
        .slave_1_read(read),
        .slave_1_write(write),
        .slave_3_tristate_bus(tristate_bus),
        .slave_3_addr(slave_addr),
        .slave_3_read(read),
        .slave_3_write(write)
    );
endmodule

```

Normally each of the slaves as well as the master would be in different FPGAs and there

would be a separate instantiation for the master and each of the slaves. However, to simply project, the master and slaves were put in the same FPGA, but are independent with each having it own set of pins. With this approach only one Quartus project had be be created and only one instantiation was needed in the testbench.

The prototype Verilog HDL that builds the master and two slaves is given below. Slaves 0 and 2 are not constructed and slaves 1 and 3 are a single register that can be written and read.

```

module tristate_buses (
input clk,
// *****
// pins for the master
output reg [7:0] masters_input_register, // storage of data read from a slave
inout wire [7:0] master_tristate_bus,
output reg [1:0] master_addr,
output reg master_read, master_write,
// *****

// pins for slave number 1
inout[7:0] slave_1_tristate_bus,
input [1:0] slave_1_addr,
input slave_1_read, slave_1_write,
// *****

// pins for slave number 3
inout[7:0] slave_3_tristate_bus,
input [1:0] slave_3_addr,
input slave_3_read, slave_3_write
// *****
);
// end of port list
// *****

// *****
// circuit for master
reg[7:0] counter; /* This counter serves three roles:
                    a) it determines when to read from and
                       write to the slaves
                    b) it provides the slave's address
                    c) it is the source of data that is to
                       be written to the slaves
                    */
wire [7:0] master_read_data;
always @ (posedge clk)
counter = counter+8'b1;

```

```

always @ *
if (counter[2]==1'b1) // 4 consecutive reads from slaves
    begin master_read = 1'b1; master_write = 1'b0; end
else // 4 consecutive writes to slaves
    begin master_read = 1'b0; master_write = 1'b1; end

always @ *
master_addr = counter[1:0];

assign master_tristate_bus = (master_write) ? counter : 8'bz;
assign master_read_data = (master_write) ? counter : master_tristate_bus ;

always @ (posedge clk)
if (master_read)
    masters_input_register = master_read_data;
else
    masters_input_register = masters_input_register;
// end circuit for master
// *****

// *****
// circuit for slave number 1
reg [7:0] slave_1_data_out;
wire [7:0] slave_1_data_in;
reg slave_1_tristate_buffer_enable, slave_1_write_enable;

always @ *
if ( (slave_1_read==1'b1) && (slave_1_addr==2'b01) )
    slave_1_tristate_buffer_enable = 1'b1;
else
    slave_1_tristate_buffer_enable = 1'b0;

always @ *
if ( (slave_1_write==1'b1) && (slave_1_addr==2'b01) )
    slave_1_write_enable = 1'b1;
else
    slave_1_write_enable = 1'b0;

assign slave_1_tristate_bus = ( slave_1_tristate_buffer_enable )
                             ? slave_1_data_out : 8'bz;
assign slave_1_data_in = (slave_1_tristate_buffer_enable)
                        ? slave_1_data_out : slave_1_tristate_bus;

```

```

always @ (posedge clk)
if (slave_1_write_enable)
    slave_1_data_out = slave_1_data_in;
else
    slave_1_data_out = slave_1_data_out;

// end circuit for slave 1
// *****

// *****
// circuit for slave number 3
reg [7:0] slave_3_data_out;
wire [7:0] slave_3_data_in;
reg slave_3_tristate_buffer_enable, slave_3_write_enable;

always @ *
if ( (slave_3_read==1'b1) && (slave_3_addr==2'b11) )
    slave_3_tristate_buffer_enable = 1'b1;
else
    slave_3_tristate_buffer_enable = 1'b0;

always @ *
if ( (slave_3_write==1'b1) && (slave_3_addr==2'b11) )
    slave_3_write_enable = 1'b1;
else
    slave_3_write_enable = 1'b0;

assign slave_3_tristate_bus = ( slave_3_tristate_buffer_enable )
                             ? slave_3_data_out : 8'bz;
assign slave_3_data_in = (slave_3_tristate_buffer_enable)
                         ? slave_3_data_out : slave_3_tristate_bus;

always @ (posedge clk)
if (slave_3_write_enable)
    slave_3_data_out = slave_3_data_in;
else
    slave_3_data_out = slave_3_data_out;

// end circuit for slave 3
// *****
endmodule

```

The output of the test bench is shown in Figure 4. It portrays the writing and reading of 4 slaves (which are a single register) numbered slave_0, slave_1, slave_2 and slave_3. In accordance with Figure 4 slaves 0 and 2 don't exist so will not respond when the master tries to read them. On the first four rising clock edges the master writes 8'H00, 8'H01, 8'H02 and

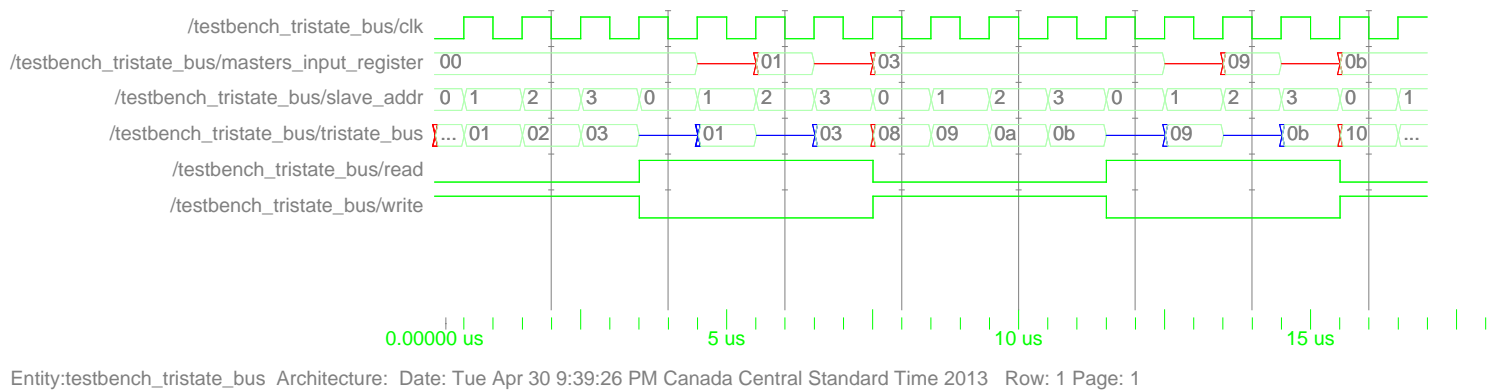


Figure 4: Test bench waveforms for the system in Figure 3

8'H03 into slaves 0, 1, 2 and 3, respectively. On the next 4 clock edges the master reads slaves 0, 1, 2 and 3, in that order, by transferring the value on the tristate bus into its input register, which is called `masters_input_register`. The master's input register is has been made an output so that it could be displayed in Modelsim-Altera. Notice that when slaves 0 and 2 are read the simulator shows the tristate bus as blue line midway between a 0 and a 1. This indicates "high impedance". Also notice that when this "high impedance" is clocked into `masters_input_register` it shows as a red line midway between a 0 and a 1. This indicates the value in undefined.

PART IIIb

4 Foundation for Exam Questions

4.1 Zero-Overhead Loops

Concept of a simple zero-overhead loop

Loops executed in software have extra instructions that keep track of how many times the loop has been executed as well as the instruction that jumps to the beginning of the loop. An example of a loop is given below:

```

    load y1, #-4'h1;    make increment value equal -1
    load x1, #N;       execute the loop N times
start: mov x0, ireg;    first instruction in the loop

    add x1, y1;        decrement x1 and set/clear zero flag
    mov x1, r;         store the decremented value
    jnz start;         jump to start until x1==0

```

The instructions in bold face print are overhead. Two of the overhead instructions are executed before the loop is entered and three are executed as part of the loop. The total number of overhead instructions executed are $3N + 2$, where N is the number times the loop is executed.

Hardware can be used to eliminate the overhead instructions that are in the loop. However, there is always some overhead as the hardware requires instructions to initialize it. The overhead instructions that initialize the hardware are executed prior to entering the loop, so are not too significant.

Loops implemented without overhead instructions inside the loop are referred to as zero-overhead loops even though there are a couple of overhead instructions that need to be executed before entering the loop.

Principle of operation of a Simple Overhead Loop

The hardware that supports zero overhead loops controls the program sequencer. Basically the next address logic is altered so that when the last instruction in the loop is executed, i.e. when the address for the last instruction in the loop is in the PC, **pm_address** is assigned the address for the first instruction in the loop. In essence the last instruction in the loop becomes a execute-and-jump-to-start-of-loop instruction. Of course after the loop is executed a predetermined number of times the jump-to-start-of-loop is not done.

The hardware that supports zero overhead loops needs the following information:

1. The program memory address for the first instruction in the loop;
2. The program memory address for the last instruction in the loop; and
3. The number of times the loop is to be executed.

These three pieces of information are held in registers. Furthermore, it is not necessary to store the full address for both the first and last address of the loop. The address for the last instruction could be specified relative to the address for the first instruction. This relative address would be the number of instructions in the loop minus 1.

The three registers are called: `start_addr`, `loop_length` and `loop_count`. They can be described as follows:

1. `start_addr` holds the address for the first instruction in the loop.
2. `loop_length` (which could be more aptly named `loop_length_minus_1`) indirectly provides the address for the last instruction in the loop. The address of the last instruction in the loop is `start_addr` plus the contents of `loop_length`, i.e. the address of the last instruction in the loop is `start_addr` plus (the number of instructions in the loop minus 1).
3. `loop_count` holds the number of times the loop is **repeated**, i.e. the loop is **executed** `loop_count+1` times.

The instruction set for the microprocessor has to be modified or expanded to have two additional “load” instructions. One that loads the `loop_length` register and one that loads the `loop_count` register. The `start_addr` register can be (and is) loaded at the same time as `loop_count`. Loading `start_addr` register with the “load `loop_count`” instruction is made possible by defining the start of the loop to be the instruction that follows the “load `loop_count`” instruction. That being the case, `start_addr` must be loaded with the address for the “load `loop_count`” instruction + 1. Since at the time “load `loop_count`” is executed its address is in the PC, `start_addr` must be loaded with PC+1.

The actual loop length is 1 more than the value in `loop_length`. For example if `loop_length==0` the loop would be of length 1 and the address for the first and last instructions in the loop are the same. The address for the last instruction in the loop is given by
address for last instruction = `start_addr + loop_length`;

The `loop_count` register is loaded with the number of times the loop is to be repeated, which is the number of times a jump back to the beginning of the loop is executed. Therefore, if the `loop_count` is loaded with 4'H0, the instructions in the loop are executed, but there is no jump back to the start of the loop. This means `loop_count` is loaded with a number that is one less than the number of times the loop is to be executed.

If `loop_count==0` when the last instruction in the loop is in the instruction register, then the instruction following the last instruction in the loop is executed next and `loop_count` is not changed. If `loop_count!=0` when the last instruction in the loop is in the instruction register, then `loop_count` is decremented on the clock edge that executes the last instruction in the loop and the next instruction to be executed is the first instruction in the loop.

Jump and conditional jump instructions can not be the last instruction in the loop as this sets up a conflict. It is not possible to execute the jump instruction and also jump to the first instruction in the loop.

An example program for an 8-sample (samples appear on `i_pins`) accumulator implemented with a zero overhead loop is given below:

```
load loop_length, #4'd2; loop length is 2+1=3
```

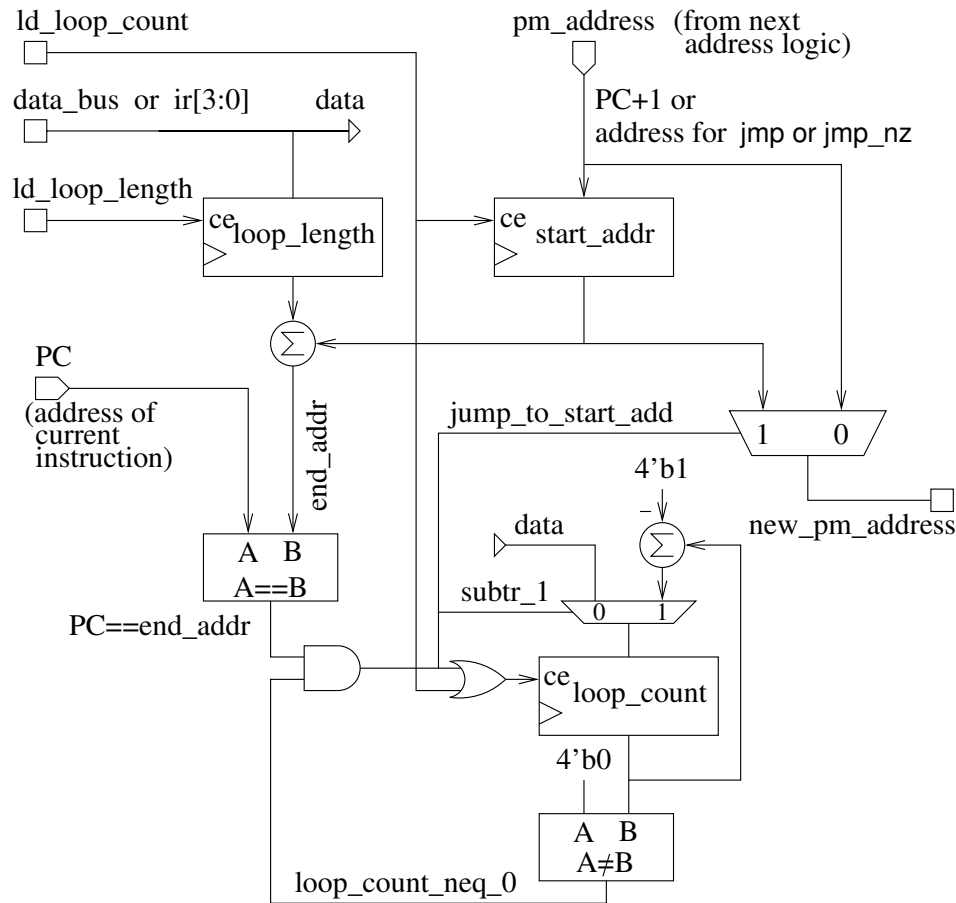



Figure 5: A block diagram of the hardware that supports a simple overhead loop

could be any number of instructions here

```

load y0, #4'H0; use y0 as accumulator
load loop_count, #4'd7; //execute the loop 8 times
LOOP:  mov x0,ireg;          // first instruction in the loop
      add x0, y0
      mov y0,r;             // last instruction in the loop
      mov o_reg,r;

```

The block diagram for the hardware that supports a zero overhead loop is given in Figure 5. The circuit is implemented in the program sequencer. This means the port list of the program sequencer would have to be expanded to include two 1-bit signals from the instruction decoder called `ld_loop_length` and `ld_loop_count`. It may also have to be modified to take in the `data_bus`.

The instruction set would have to be modified to include a “load loop length” instruction and a “load loop count” instruction. There are a variety of ways the new instructions could be created.

1. One way is to convert existing load instructions, for example convert “load x1” and

“load y_1 ” to “load loop length” and “load loop count” instructions. Since the data for a load instruction, which is the `ir[3:0]`, already is an input port to the program sequencer there would be no need to make `data_bus` and input.

2. Another way is to use no-ops, for example `NOPC8` and `NOPD8` as the “load loop length” and “load loop count” instructions. In this case the values to be loaded would have to come from two registers, say x_1 and y_1 , so `data_bus` would have to be connected to the program sequencer.

Hardware support for simple Zero-Overhead For Loop

Often compilers support for loops. A simple for loop could have syntax

```
for i = init_contant, i < limit_constant, i = i + incr_constant,
end
```

All of the control parameters are limited to constants making it a simple for loop construct. An usage example is

```
for i = 2, i < 9, i = i + 2,
executable statements
endfor
```

The CME341 assembler version of the simple for loop is given below

```
    load loop_length, #N_length; length of loop - 1
    load limit, #N_limit;      upper limit for index
    load increment, #N_incr;   amount by which index is incremented
    load index, #N_init;       load initial value to mark start of loop
LOOP: mov x0, ireg;           first instruction in the loop

    loop instructions

    mov o_reg,r;              last instruction in the loop
```

Jump and conditional jump instructions can not be the last instruction in the “for loop” as this sets up a conflict. It is not possible to execute the jump instruction and also jump to the first instruction in the “for loop”.

The portion of the hardware for the “for loop” that is integrated into the program sequencer is shown in Figure 6.

Hardware support for a simple Zero-Overhead While Loop

Often compilers support while loops. A simple while loop may have syntax

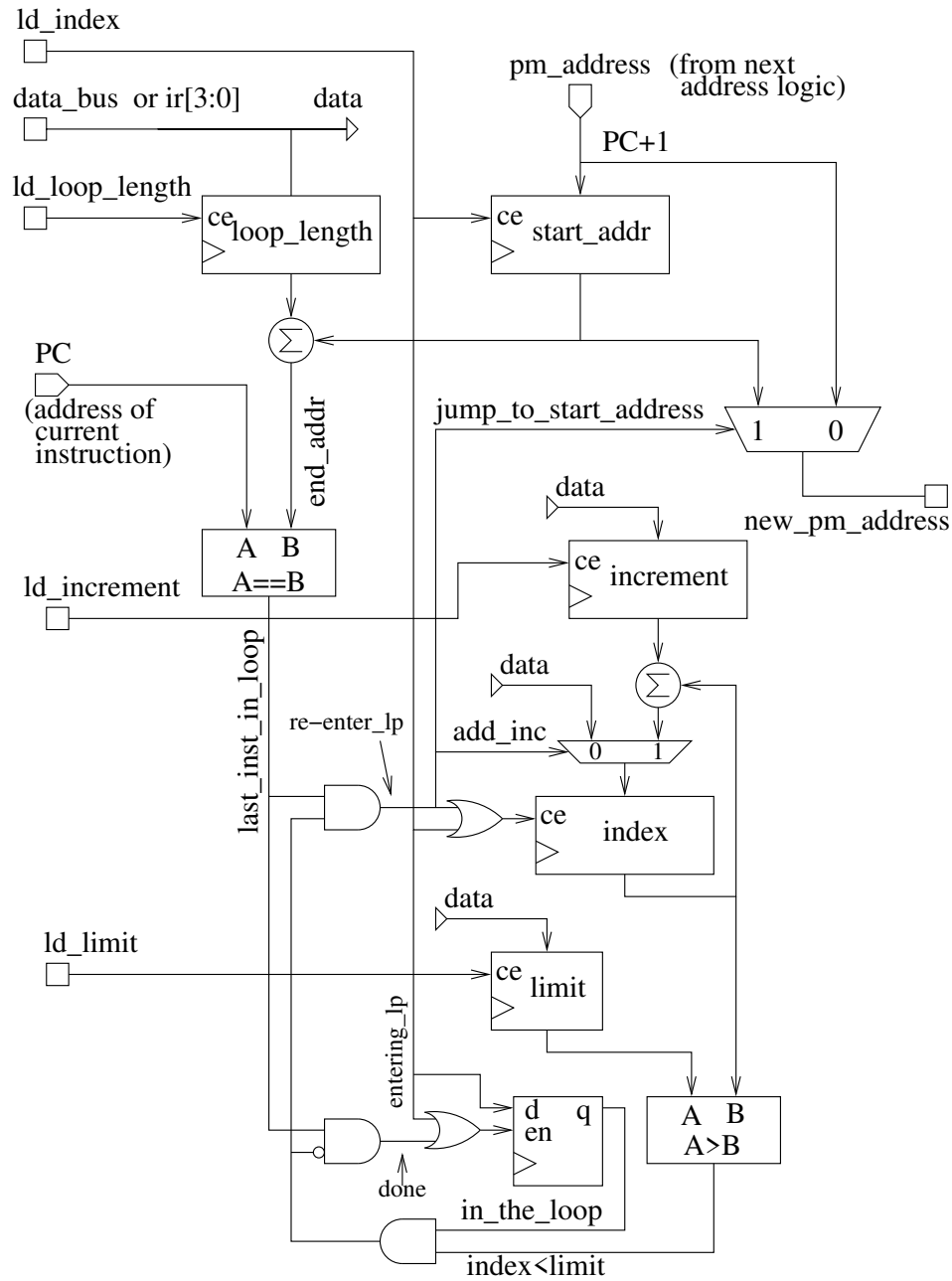


Figure 6: A block diagram of the program-sequencer hardware that supports a simple For-Loop

```
do  while ( x < constant )
```

```
    instructions in loop
```

```
endwhile
```

The loop is executed at least once, whether or not the expression (`x < constant`) is true. The test expression, i.e. (`x < constant`) is evaluated at the end of the loop. If it is true, the next instruction executed is the first instruction in the loop.

There is a big difference between the “simple while” loop and the “simple for” loop. The index for the simple for loop is a new register that is updated on the positive edge of the clock that executes the last instruction in the loop. The test variable in the while loop is a data memory variable that can be written to by a variety of instructions anywhere in the loop.

The CME341 assembler version of the simple while loop is

```
    load loop_length, #4'H3; loop length minus 1
    load limit, #4'H7; load the limit constant
    load i, #addr_of_x; set up address of data variable x
    load addr_x, #addr_of_x; store the address of x
    mov shadow_x, dm;    up to date copy of x
                        ; marks the start of the while loop
Loop: mov x0, ireg;    first instruction in the loop
```

At the end of the while loop the value of the test variable `x`, i.e. the data memory location with address `addr_of_x`, must be compared to the constant in `limit`. A while loop would not be considered a zero-overhead loop if data memory variable `x` had to be read with an assembler instruction in order to test it. Such a read can be avoided if an up to date copy of `x` is kept in a special register, say the `shadow_x` register, for the purpose of performing the test at the end of the loop. To be sure the copy of `x` is up to date upon entering the loop, `shadow_x` must be updated with `x` one instruction prior to entering the loop. This means the beginning of the loop is defined by the location of the `mov shadow_x, dm;` instruction. To keep `shadow_x` up to date it must be written whenever `x` is written and with the same value.

This means `shadow_x` is updated with a newly created `mov shadow_x, dm;` instruction and also with the existing move to `dm` and load `dm` instructions providing the `i` register contains the value `addr_of_x`, which is the address of `x`.

The zero overhead “while loop” hardware has other things to do as well. It must also evaluate the expression “`shadow_x < limit`”. If the expression evaluates to true when the last instruction in the “while loop” is in the instruction register, then the next instruction executed must be the first instruction in the “while loop”. Otherwise the execution order falls through the bottom of the loop to the instruction that follows the last instruction in the loop. A partial block diagram of the circuit for a zero-overhead while loop is shown in Figure 7

Jump and conditional jump instructions can not be the last instruction in the “while loop” as this sets up a conflict. It is not possible to execute the jump instruction and also jump to the first instruction in the “while loop”.

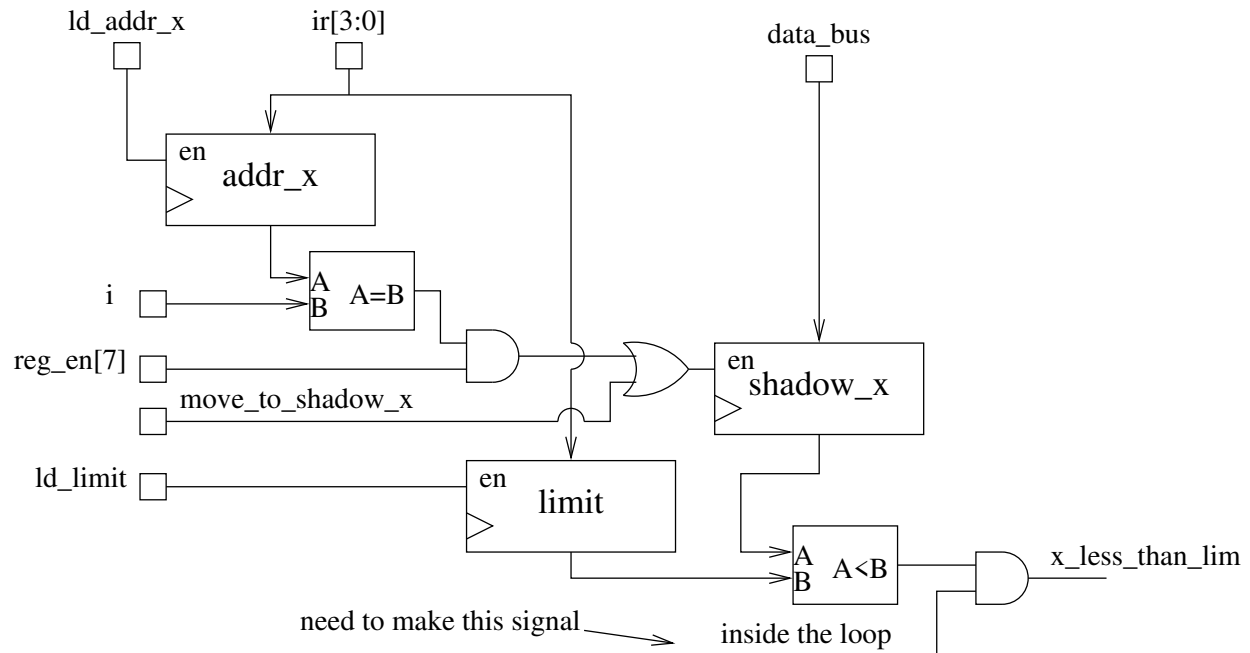


Figure 7: Part of a block diagram of the program-sequencer hardware that supports a simple while-Loop

4.2 Adding Interrupt Capability

Functionality

Microprocessors often have a feature where the assertion of an external pin forces the program sequencer to jump to a predefined address. Obviously, such action will interrupt the execution of the program that was running at the time the external signal was asserted. For this reason the name of the external pin is usually an mnemonic for “external interrupt” or “hardware interrupt”, which could be `ext.int`.

The predefined address (sometimes referred to as the interrupt vector) is the address of the first instruction of a separate program called an Interrupt Service Routine (ISR). The program that was interrupted is referred to as the main program. The ISR can be viewed as the interruption to the main program.

Microprocessors that have an interrupt feature will also have a special “return” instruction that exits the ISR and returns to the main program. The point of return is the address of the next instruction in the main program, i.e. the instruction in the main program that follows the point of interruption. This special “return” instruction is called Return From Interruption (RFI) or Return To Interrupted program (RTI) or something to that effect.

The predefined address is the address of the first instruction in the ISR. This predefined address will be referred to as the interrupt vector. A microprocessor may have more than 1 external interrupt pin. If so, there will a distinct interrupt vector associated with each pin.

Microprocessors with external interrupt capability require extra circuitry in the program sequencer. For one thing, there must be an additional register (actually one register per ISR) to save the return address. The register that holds the return address will be labelled `return_addr` or something to that effect.

At the time of an interruption the `return_addr` register is loaded with the value that `pm_address` would otherwise have. For example, if the interrupt signal goes active while a load, move or ALU instruction is in the instruction decoder, then `return_addr` is loaded with `PC+1` on the same clock edge that executes the load, move or ALU instruction. If, on the other hand, the interrupt signal goes active while a “jump” instruction is in the instruction decoder, then the `return_addr` register is loaded with the target address of the jump instruction. It is loaded on the positive edge of the clock that executes the jump instruction.

Obviously, it is not possible to execute a jump instruction and also “jump to the interrupt vector” at the same time. The “jump to the interrupt vector” must be executed by definition of the “interrupt”. The interrupted jump instruction is executed upon return from the ISR since `return_addr` holds the target address of the jump instruction.

Microprocessors with external interrupt capability have two other instructions: one that disables and one that enables the external interrupt signal. These instructions, which will be referred to a `enable_interrupt` and `disable_interrupt`, are necessary to prevent an interrupt from interrupting the interruption. By making the first instruction in the ISR the `disable_interrupt` instruction the ISR is not interruptible. Of course, the `enable_interrupt` must be used to enable the interrupts just prior to exiting the ISR.

The `disable_interrupt` and `enable_interrupt` instructions do not have arguments. This means no-operation instructions, like `NOPC8` and `NOPD8`, can be used for `disable_interrupt` and `enable_interrupt` instructions.

The hardware interface between the external interrupt pin and the program sequencer is shown in Figure 8. Pins `disable_interrupt` and `enable_interrupt` are high if and only if the instructions in the instruction register are `disable_interrupt` and `enable_interrupt`, respectively. The instruction disables the interrupt beginning with the instruction following it. The `enable_interrupt` instruction takes effect one instruction later. The interrupt is enabled starting with the second instruction after it.

The external interrupt signal is synchronized to the clock using a flip/flop. It is processed by the edge detector to produce a pulse one clock pulse wide each time the synchronized interrupt signal makes a low to high transition.

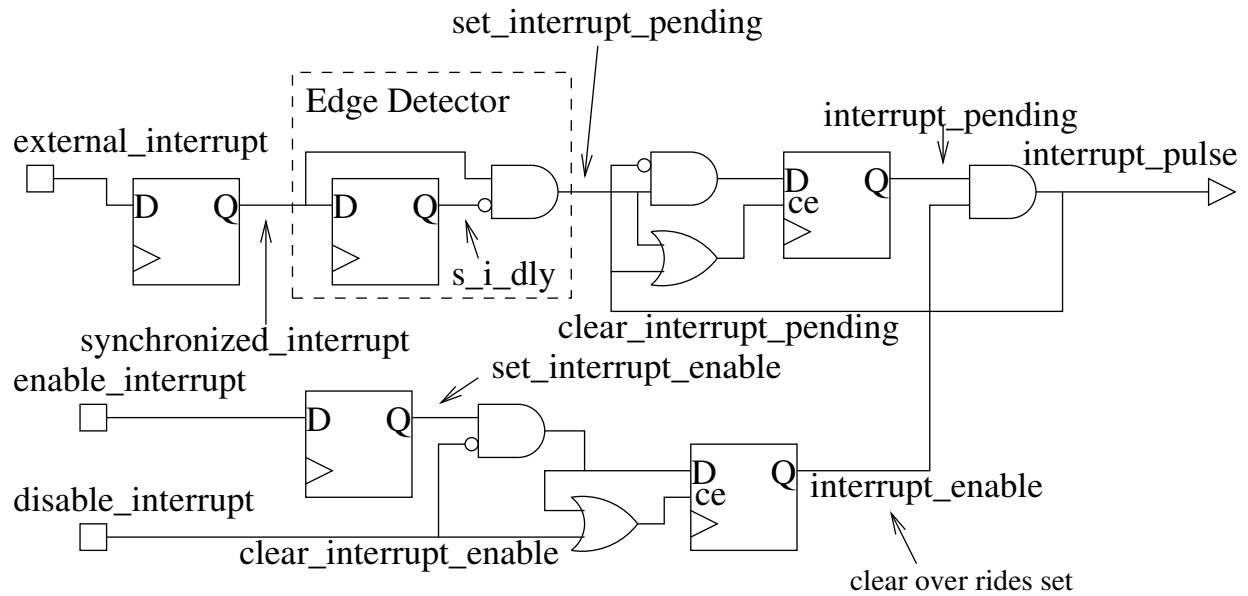
It is pointed out that it is not possible for the circuit output, which is called `interrupt_pulse`, to be two clock periods wide. This ensures the first instruction in the ISR can not be interrupted. That being the case, if `disable_interrupt` is the first instruction in the ISR, the ISR can not be interrupted.

The `enable_interrupt` instruction does not take effect on the instruction following it, but on the instruction after that. This means if an `enable_interrupt` instruction is placed just before the RFI instruction in the ISR, the RFI instruction can not be interrupted. Such placement ensures the ISR will not be interrupted.

The Verilog HDL for the circuit is given below (Beware the code has not been debugged.):

```
always @ (posedge clk)
synchronized_interrupt = external_interrupt;

always @ (posedge clk)
s_i_dly = synchronized_interrupt;
```



NOTES: The `interrupt_enable` flip/flop is a synchronous set/clear flip/flop where clear over rides set. The over ride is necessary for correct operation in cases where a `disable_interrupt` instruction follows immediately after an `enable_interrupt` instruction. In this situation both `set_interrupt_enable` and `clear_interrupt_enable` will be high at the same time. Since the disable instruction follows the enable instruction it must have priority.

Figure 8: The interface between the program sequencer and the external interrupt signal

```

always @ *
set_interrupt_pending = (~s_i_dly) & synchronized_interrupt;

always @ (posedge clk)
if (interrupt_pulse) // i.e. if (clear_interrupt_pending)
    interrupt_pending = 1'b0;
else if (set_interrupt_pending)
    interrupt_pending = 1'b1;
else
    interrupt_pending = interrupt_pending;

always @ *
    interrupt_pulse = interrupt_pending & interrupt_enable;

always @ (posedge clk)
    set_interrupt_enable = enable_interrupt; // delay effect of instruction

always @ (posedge clk)
if (disable_interrupt)
    interrupt_enable = 1'b0;
else if (set_interrupt_enable)
    interrupt_enable = 1'b1;
else
    interrupt_enable = interrupt_enable;

```

4.3 Hardware Support for Breakpoints

Introduction

Hardware can be added to the microprocessor to support the debugging of a program. One of the simplest features allows the programmer to interrupt the program at a pre-specified address and vector it to a special service routine called a monitor. Normally the monitor communicates with the keyboard and screen allowing the programmer to examine registers and memory locations. Of course the keyboard/screen interface will not be implemented in this class.

Another more complicated feature allows a programmer to determine when a certain memory location is written by an instruction that is located inside or outside a pre-specified address range. In this case the hardware must compare the value in the PC at the time the memory location of interest is written to the pre-specified endpoints of the address range. Say, the first and last addresses in the monitored range is `address_of_first_instr` and `address_of_last_instr`, respectively, then a monitor interrupt is generated if $\text{address_of_first_instr} \leq \text{PC} \leq \text{address_of_last_instr}$.

Break Immediately

The word “break” is used to mean interrupt the program immediately and pass control to the monitor. This is the most basic feature of the “breakpoint” suite. Normally an

interrupt pin is used to interrupt the program and pass control to the monitor. The monitor is normally a subroutine that is called from the interrupt service routine, but the monitor can be invoked with a jump instruction. In the latter case the monitor would return control to the interrupted program by the execution of an `enable_interrupt` instruction followed by an `RTI` instruction.

Obviously the program sequencer would have to be modified to support the interrupt and a return to interrupted program instruction would have to be added to the instruction set.

Address Breakpoint

The word “breakpoint” is used to mean interrupt the program at a certain point and pass control to the monitor. An address breakpoint interrupts the program after the execution of the instruction at a specified address. In this case the interrupt signal is generated internally.

The hardware support circuitry includes an 8-bit register called `pm_breakpoint_addr` that can be loaded with the specified program memory address and a comparator that tests the equality `pm_breakpoint_addr == PC`. The output of the comparator is used to generate the interrupt.

Data Breakpoint

A data breakpoint causes the program to break when the instruction in the instruction register is about to write to a specified data memory location.

The hardware support circuitry includes a 4-bit register called `dm_breakpoint_addr` that can be loaded with the specified data memory address. The hardware must check that a memory write instruction is in the instruction decoder and the target of that write is has data memory address `dm_breakpoint_addr`.

4.4 Hardware Support for Subroutines

A subroutine is a little program that resides in program memory along with the main program, but obviously in a different address space. A subroutine is “called” from the main program, which means the main program transfers control to the subroutine by executing a Jump-To-Subroutine (JSR) instruction. Control is transferred back to the main program by the instruction in the last line of a subroutine, which is referred to as a Return-From-Subroutine (RFS) instruction.

Basically, a JSR instruction generates an interrupt with the interrupt vector being the “jump address” contained in the address field of the JSR instruction. Since a JSR instruction in essence generates an interrupt, it also saves the return address in a register, say `return_addr`. In the case of a JSR the return address is always `pc+1`. The RFS (return from subroutine) instruction behaves exactly like a RFI (return from interrupt) instruction. It instructs the program sequencer to make `pm_addr` equal to `return_addr`.

A subroutine call could be considered a software interrupt or a “jump a to interrupt vector” instruction. It is basically a jump instruction, but the program sequencer has to be modified to store the address of the next instruction, i.e. `PC + 1`, in a register called

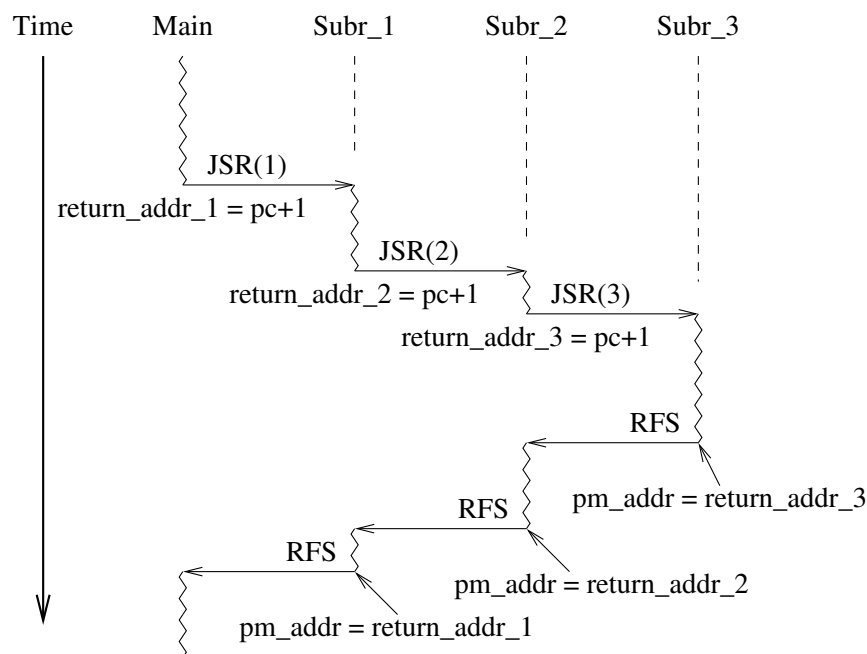


Figure 9: Illustration of the transfer of control to and from subroutines

return_addr. This is accomplished by loading **return_addr** with **PC+1** on the same edge of the clock that executes the jump.

In addition, a **RFS** instruction must be added to the instruction set. This instruction causes the program sequencer to make **pm.address = return_addr**. Since an **RFS** instruction does not have an argument one of the no-op instruction could be reassigned to be an **RFS** instruction.

Most microprocessors allow a subroutine to be called from another subroutine. This means more than 1 return address has to be saved. The order in which the return addresses have to first be saved and then be retrieved is shown in Figure 9. The program starts in main and remains there until instruction **JSR(1)**, which is jump to subroutine 1, is reached. The execution of **JSR(1)** transfers control from main to subroutine 1 and, at the same time, **pc+1** is clocked into **return_addr_1**.

The program then runs in subroutine 1 until instruction **JSR(2)** is executed. At that time control is transferred to subroutine 2 and **pc+1** is clocked into **return_addr_2**.

The program then runs in subroutine 2 until instruction **JSR(3)** is executed. At that time control is transferred to subroutine 3 and **pc+1** is clocked into **return_addr_3**.

The program then runs in subroutine 3 until instruction **RFS** is executed. At that time control is transferred back to subroutine 2 by setting **pm_addr = return_addr_3**.

The program then runs in subroutine 2 until instruction **RFS** is executed. At that time control is transferred back to subroutine 1 by setting **pm_addr = return_addr_2**.

The program then runs in subroutine 1 until instruction **RFS** is executed. At that time control is transferred back to main by setting **pm_addr = return_addr_1**.

For purposes of implementation the order in which the return addresses are saved and then retrieved is of importance. They are saved with 'tt JSRs in the order: **return_addr_1**, **return_addr_2** and **return_addr_3** and then retrieved with **RFSs** to return control to the calling subroutine in the reverse order. This means the return addresses can be saved to a

Last-In-First-Out (LIFO) queue instead of separate registers. LIFO queues are often referred to as stacks because they operate much like a stack of dinner plates where the last plate washed and placed on the stack will be the next plate used.

Using a stack (LIFO Queue) for the return address

LIFO queues, i.e. stacks, are so commonly used that their operation is described with generic terminology. The act of placing data on a stack is called “pushing” the data on the stack and the act of removing data from a stack is called “popping” data from the stack. An instruction that places data on a stack is referred to as a “push” instruction and an instruction that removes data from a stack is referred to as a “pop” instruction. The “the top of the stack” is referred to as TOS.

A stack that stores the return addresses for jumps to subroutines would be implemented in the program sequencer. In this instance the “push” signal would be the JSR signal generated by the instruction decoder. The JSR signal is high while a JSR instruction is in the instruction register. The push is executed (i.e. the return address is written to the stack) on the same clock edge that executes the JSR instruction.

The “pop” signal would be the RFS signal generated by the instruction decoder. The RFS signal is high while the RFS instruction is in the instruction register. The TOS, which is the return address for the current RFS instruction, is removed and all other elements in the stack moved up one position on the same clock edge that executes the RFS instruction.

The program sequencer makes `pm_addr = TOS` while the RFS signal is high. This places the return address on `pm_addr` while the RFS signal is high. Doing so ensures the next instruction loaded into the instruction register comes from program memory address `pm_addr = TOS`. The TOS is popped after it has been used for the return address.

Building a Stack with Registers

A stack can be built from either registers or memory. If the stack is relatively small then building it with registers makes the most sense. A block diagram for a 4-word stack built from registers is shown in Figure 10. The part of the Verilog HDL for that 4-word stack is given below:

```
module stack (
input pop, push, clk, sync_reset,
input [7:0] stack_input,
output reg [7:0] top_of_stack
);
reg [7:0] reg_1, reg_2, reg_3, reg_4, top_of_stack;
always @ *
top_of_stack = reg_1;

always @ (posedge clk) // build reg 1 circuit
if (sync_reset == 1'b1)
reg_1 <= 8'H00;
else if (push == 1'b1)
```

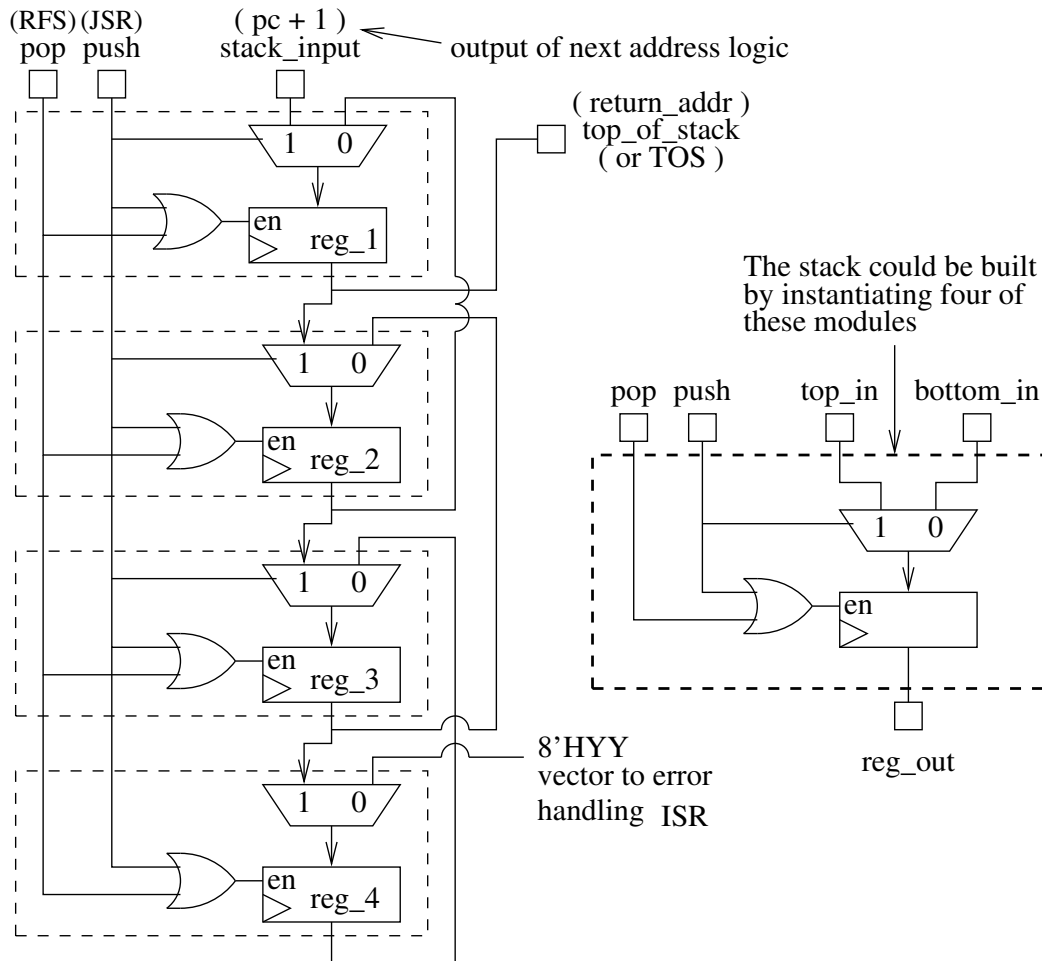


Figure 10: A circuit for an 4-word by 8-bits/word stack based Registers

```

    reg_1 <= stack_input;
else if (pop == 1'b1)
    reg_1 <= reg_2;
else
    reg_1 <= reg_1;

always @ (posedge clk) // build reg 2 circuit
if (sync_reset == 1'b1)
    reg_2 <= 8'H00;
else if (push == 1'b1)
    reg_2 <= reg_1;
else if (pop == 1'b1)
    reg_2 <= reg_3;
else
    reg_2 <= reg_2;

etcetera

\endmodule

```

Building a Stack with RAM

A stack can be built using a synchronous two port RAM as storage. One port is used for pushing and the other is used for reading the TOS. Additional circuitry is needed to control the addresses of the two ports. The circuitry must ensure that the output of the read port is always the TOS and that a push adds an entry to the stack and a pop removes one.

Central to a RAM based stack is a register which will be referred to as **TOS_address** (TOS stands for Top Of Stack). The **TOS_address** register holds the address for the data entry on the Top Of the Stack (TOS). Registers that hold program memory addresses are often referred to as pointer registers so **TOS_address** is more appropriately described as “the pointer register that holds the address for the TOS, where TOS is the data entry on the Top Of the Stack”. A register that holds the address of the TOS is often called **stack_pointer**, but in the stack implemented here it will be called **TOS_address**.

In is very important to understand that synchronous RAMs have registered inputs and that these registers introduce latency. The address input to the synchronous RAM is clocked into a register. The output of this register is the address for the asynchronous RAM (the actual memory that stores the data). Of course the data input to the synchronous RAM is also registered so that the data is stored in the asynchronous RAM at the associated address.

The output of the synchronous RAMs may or may not be registered. The synchronous RAM used in the implementation of the CME341 stack will not have registered outputs. This means the output of the synchronous RAM is that of the asynchronous RAM.

The latency created by the registering the inputs does not cause any problems for writing, but can cause problems for reading. To read the contents for a particular address on a clock edge, that address must have been written on a previous clock edge.

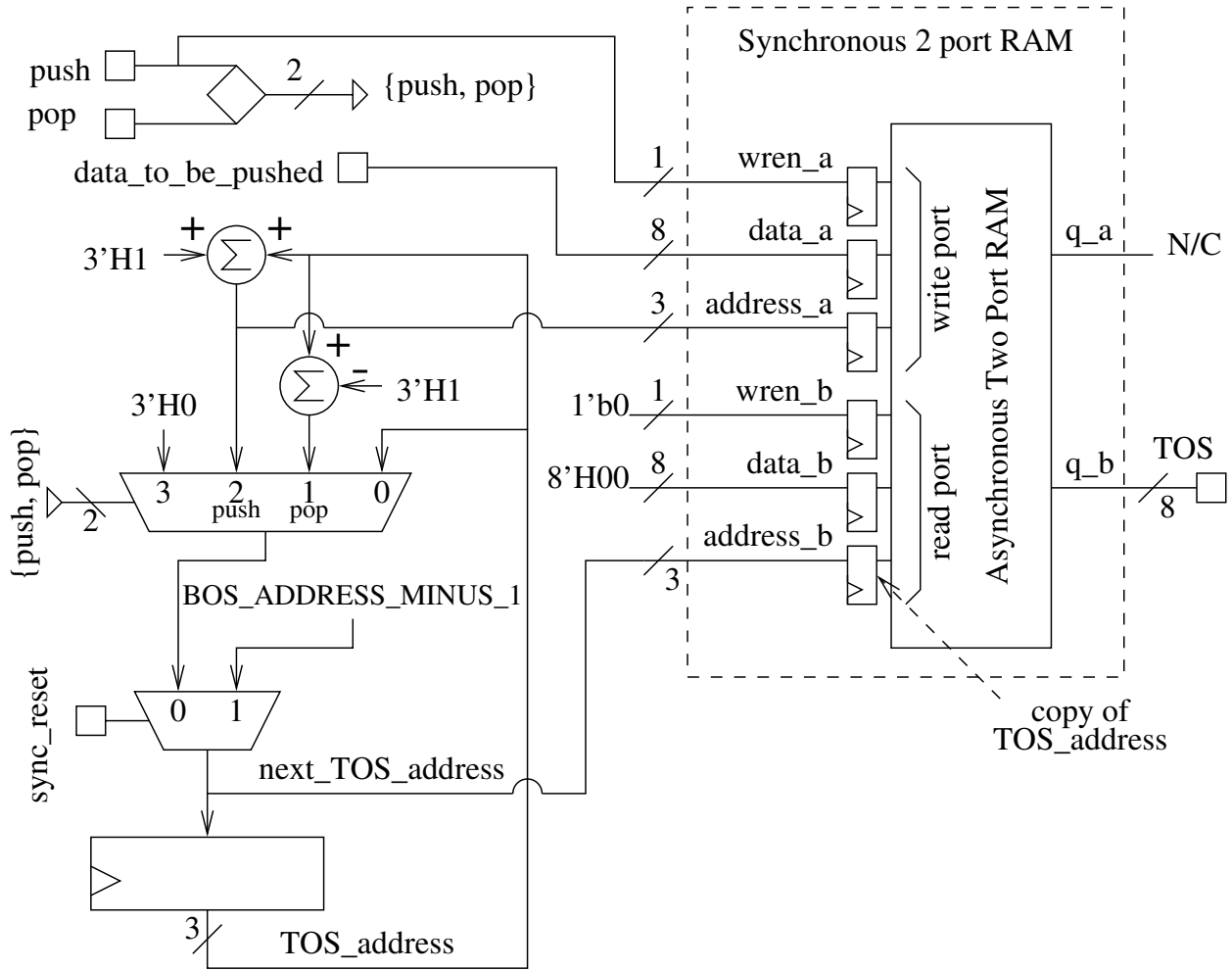


Figure 11: A circuit for an 8-word by 8-bits/word stack based on a 2-port synchronous RAM

*The implication of the latency is that the address for the read port must be calculated one clock period in advance. For this reason the address input to read port of a stack built with synchronous RAM could be aptly named **next_TOS_address**.*

A circuit that implements a stack using a two port synchronous RAM is given in Figure 11. The synchronous RAM is shown enclosed in a dashed line box. It is an asynchronous RAM with registered inputs. The inputs and outputs with names ending in ‘_a’ form the write port. The inputs and outputs with names ending in ‘_b’, form the read port. The contents of the asynchronous RAM is never read through the output of the write port so its output is not connected. Data is never written to the asynchronous RAM through the read port so the write enable for the read port is permanently set to 1'b0. Since no data is ever written to memory through the read port, its data input is set to 8'H0.

It is clear from Figure 11 that the address register for the read port has the same input as the register called **TOS_address**. Therefore, the input to the asynchronous RAM inside the synchronous RAM is identical to the output of **TOS_address**.

Since data is written to the stack when and only when “push” is active, the enable for the write port is connected to a signal called **push**. The signal **push** will be 1'b1 while a push or

JSR instruction (a JSR instruction implies a push) is in the instruction register. When the instruction in the instruction register is a JSR, the data to be written to (or pushed onto) the stack is the return address. The return address is always the address of the instruction that follows the JSR instruction, i.e. $PC+1$.

The logic that generates the addresses for the read and write ports is fairly straight forward. The address for the write port, which can also be called the push port, must be one higher than that of the address for the top of stack, i.e. the address for the write port must be `TOS_address + 1`. The address for the top of stack will only change on a push (JSR) or pop (RFS) instruction. The circuit must compute the address for the new TOS prior to the execution of the push or pop instruction. The `TOS_address` register is updated with the next TOS address on the same clock edge that executes the push or pop. The logic is straight forward:

1. While a push instruction is in the instruction register the next TOS address will be `TOS_address+1`.
2. While a pop instruction is in the instruction register the next TOS address will be `TOS_address-1`.
3. While any other instruction is in the instruction register the next TOS address will remain `TOS_address`.

In setting up a stack it is helpful to define a constant called `BOS_ADDRESS_MINUS_1` for the purpose to initializing `TOS_address`. `BOS_ADDRESS_MINUS_1` must be chosen so that it is one less than the address where the first data entry is to be stored. That is to say, the first push instruction stores data in location `BOS_ADDRESS_MINUS_1 + 1`. The most likely value for `BOS_ADDRESS_MINUS_1` in CME341 would be -1.

The microprocessor could be modified so that the initial value for `TOS_address` need not be specified at compile time. Initialization with `sync_reset` as shown in Figure 11 is unnecessary if the instruction set of the microprocessor is modified to include a `load_TOS_address` instruction. This instruction would allow `TOS_address` to be initialized from within the program and remove the need for initializing with `sync_reset`.

The Verilog HLD description of the circuit in Figure 11 is given below. The stack is assumed to have a depth of 8 words (i.e. 8 word RAM having 3 address bits).

JTS is 1'b1 if and only if a jump to subroutine instruction is in the instruction register.

RTS is 1'b1 if and only if a return from subroutine instruction is in the instruction register.

```
localparam BOS_ADDRESS_MINUS_1 ==-3'd1;

always @ *
    if (sync_reset) // initialize TOS_address
        next_TOS_address = BOS_ADDRESS_MINUS_1;
    else if (JTS) // push the return address
        next_TOS_address = TOS_address + 3'd1;
    else if (RFS) // pop the return address
```

```

        if (TOS_address == BOS_ADDRESS_MINUS_1) // underflow
            next_TOS_address = TOS_address; // don't allow pop
        else
            next_TOS_address = TOS_address-3'd1; // execute pop
    else
        next_TOS_address = TOS_address;

always @ (posege clk)
    TOS_address = next_TOS_address;

```

4.4.1 Example

Suppose the `load o_reg` instruction is changed to a jump to subroutine with mnemonic `jsr`. The 4-bit data field of the `load o_reg` instruction becomes the most significant four bits of the address of the target subroutine. For example, `jsr 8'H20` would have machine code `8'H42`.

Further suppose that the NOPC8 instruction is changed to be a return to interrupted routine with mnemonic **ret**.

Further suppose a stack is used to save the return address for jumps to subroutine.

Then the sequence of values for the top-of-stack, PM_address, PC and IR generated by the program that follows would be:

```
top-of-stack:  --  --  02  11  11  02  02  --  03  03  --  --  ...
PM_address:   01  10  20  21  11  12  02  20  21  03  00  01  ...
PC:           00  01  10  20  21  11  12  02  20  21  03  00  ...
IR:           10  41  42  12  C8  11  C8  42  12  C8  E0  10  ...
```

\end{verbatim}

```
\begin{verbatim}
```

[illegible]


```
        ;          second time it will be 8'H03.  
        ;          pop top of stack with the clock  
        ;          edge that executes ret
```

4.5 Building LIFO (stack) and FIFO Queues with FSMs

Queues can be built using a set of registers and a Finite State Machine (FSM) to organize the registers into either a LIFO or FIFO queue.

LIFO Queue or Stack

The current state in the FSM identifies the register that holds the top of the stack. In general terms a stack is a LIFO (last in first out) queue and the last entry is the top of the stack. The FSM could be implemented using either binary coded states or one flip/flop per state. The latter, which is referred to as one-hot state coding, will be used here.

The implementation of an N element stack requires an FSM with $N + 1$ states. The state diagram for a 4 element stack is shown at the top of Figure 12. The states are named so that the current state signifies the number of entries in the stack.

Operation is quite straight forward. Each time a push signal is encountered the FSM state advances one step in the progression: `empty` \rightarrow `depth_1` \rightarrow `depth_2` \rightarrow `depth_3` \rightarrow `depth_4`. If a push signal occurs while the FSM is in state `depth_4` it is ignored.

Each time a pop signal is encountered the FSM retreats one step in the progression: `depth_4` \rightarrow `depth_3` \rightarrow `depth_2` \rightarrow `depth_1` \rightarrow `empty`. If a pop signal occurs while the FSM is in state `empty` it is ignored.

The circuit that controls the four registers is shown at the bottom of Figure 12. The registers are numbered to coincide with the numbers of the states. The timing is such that if the current state is `empty` and a push signal is active, then on the next positive edge of the clock `reg_1` is loaded with the `input_to_stack` and the flip/flop representing `depth_1` set to 1'b1. It is pointed out that `reg_1` is not loaded if `depth_1` is entered with a pop.

The states of the FSM are used in the “4-input and-or” circuit shown at the bottom of Figure 12 to select the output of a register for the `top_of_stack`. For example, if `depth_1` is the current state it enables the “and” gate associated with `reg_1` to connect `reg_1` to `top_of_stack`.

FIFO Queue

A FIFO queue is a first-in-first-out. It is the type of queue found in banks and fast food restaurants. The fundamental difference between a FIFO and LIFO queue is that entrants to FIFO queue continue to advance in one direction until they get to the front of the queue and leave. Unlike the entrants to a LIFO queue, like the de facto queue formed in an elevator, where previous entrants move back to make room for the recent entrant and move forwards as the most recent entrant leaves.

Using the analogy of a bank or fast-food queue, one would think a FIFO queue is easily implemented as a shift register. Unfortunately, a shift register does not accurately implement such a queue. If someone enters an empty FIFO queue in a fast food restaurant they advance to the head of the queue without delay. However, in a shift register circuit there is delay in getting to the head of an empty queue. Advancement is one position at a time on a rising edge of a clock.

A FIFO queue can be implemented with a set of registers and a FSM. For purposes of analysis the FSM is best separated into two FSMs. The number of states required for each

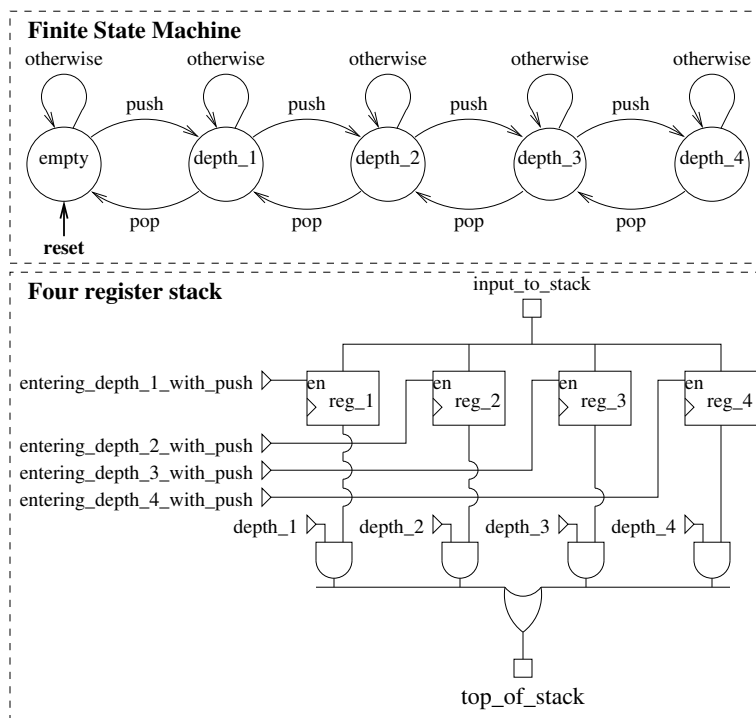


Figure 12: A Stack implemented with registers and a FSM

FSM is equal the number of elements in the queue. The two FSMs for a 4-element FIFO queue are shown at the top of Figure 13.

Of the two FSM shown Figure 13, one keeps track of the head of the queue and the other keeps track of the tail of the queue. A signal named `enter` is issued when `input_to_tail_of_queue` is to enter the tail of the queue on the next positive edge of the clock and a signal named `leave` is issued when the data at the head of the queue is to be extracted on the next positive edge of the clock.

The two FSMs are initialized to states `tail_4` and `head_4`. Both advance in a circular fashion on signals `enter` and `leave`.

The circuitry that surrounds the register is shown at the bottom of Figure 13. It is the same as the circuitry that controls the registers is a stack, which is shown at the bottom of Figure 12. Of course the signals are different in the circuit for the FIFO registers, but the operation is the same.

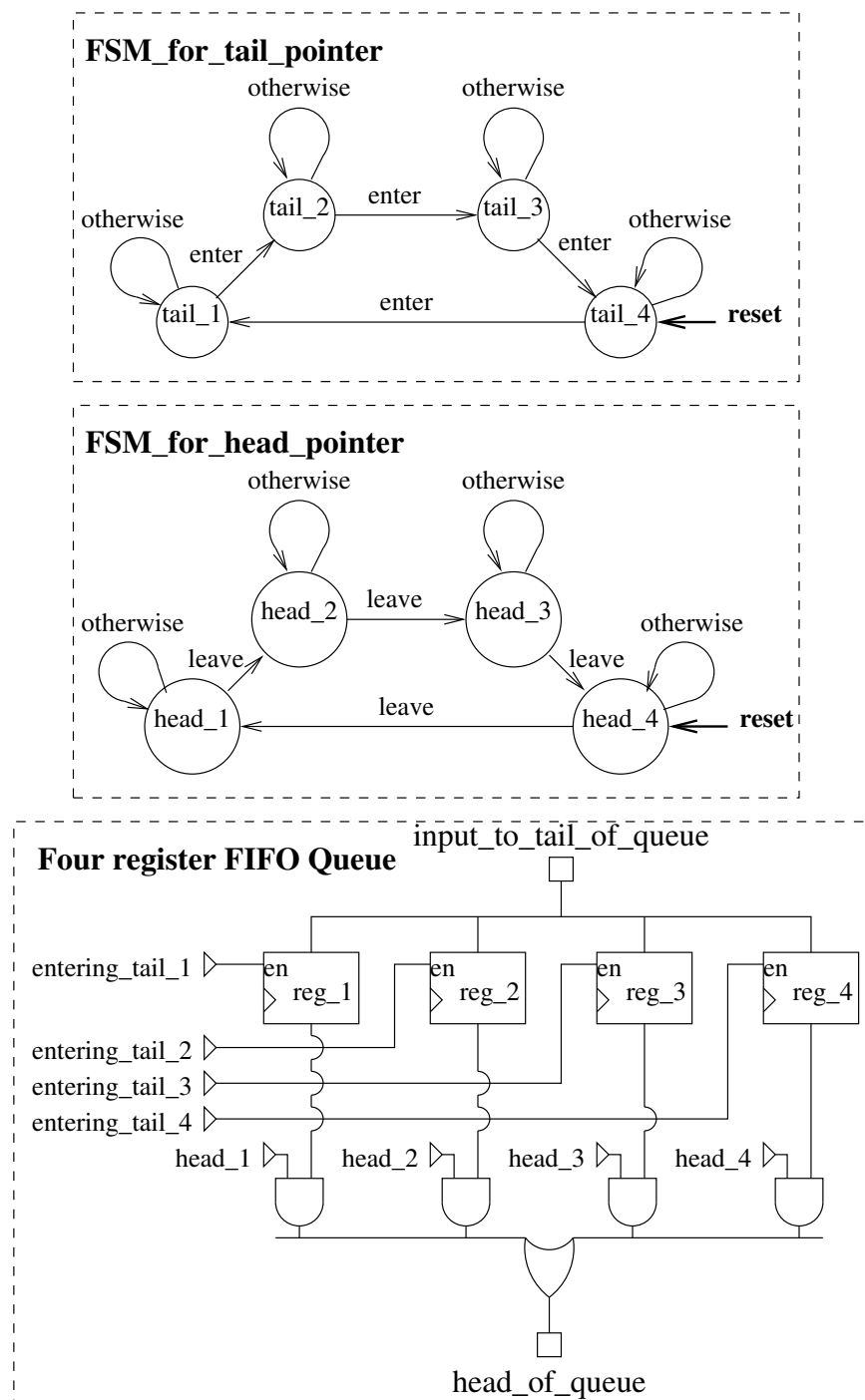


Figure 13: A FIFO Queue implemented with registers and a FSM

4.6 Extending the Address Range for Memory

The program sequencer has been designed to support a memory that has 8 address bits. It is possible to modify the program sequencer to accommodate a larger memory, that is a memory with more address bits. There are a variety of ways to do this. One way is to extend `pm_address` and the PC to 9 bits and modify the instruction set to work with the larger address space. The jump instruction would have to be redefined since it can only jump to the lower 256 words of memory with its current definition.

Another method of extending memory is called paging. The expanded memory is partitioned into pages. For the CME341 microprocessor a page would be 256 words. The program sequencer would provide the 8 least significant bits of the address without any modification. However, extra hardware is needed to provide the extra most significant bits. It will assumed for purposes of illustration that the address space for the extended program memory will be 12 bits, which is an extra 4 bits. Therefore, program memory can be viewed as 16 pages with 256 words per page.

The assembler would assemble the instructions one page at a time as if the current page was the entire memory. The program sequencer would have to be modified to provide extra address bits to program memory so the entire program memory could be addressed. The least significant 8 bits of the address would be `pm_address`. The newly added most significant bits would come from a new register called `base_reg`, which is short for base register. The base register holds the page number of the current page of memory.

It is not straight forward to execute a jump to another page. A jump from one page to a specific address in another page requires a pair of instructions: one to load the page register and the other to jump to the address within the new page. The latter could be accomplished with the existing `jmp` and `jmp_nz` instructions, but the former would require a new instruction. The new `load base_reg` instruction could be a re-tasked load instruction. For example, the `load o_reg` instruction could be redefined as the `load base_reg` instruction.

The pair of instructions would have to be executed one after the other as shown below:

```
load base_reg, #4'H3;  The base register will be loaded
                      ;  with 3 at the same time the next
                      ;  instruction is loaded into the ir
jmp 8'HA0;  base register changes to 4'H3
            ;  at the same time the jump instruction
            ;  enters the ir. The address logic makes
            ;  pm_address 8'HA0 and makes the address
            ;  for program memory {4'H3, 8'HA0}, which
            ;  causes a jump to 8'HA0 on page 3
```

The `load base_reg, #4'H3` instruction will change the most significant address bits, i.e. change the page to page number 4'H3, at the same time that the `jmp 8'HA0` instruction is loaded into the instruction register. The `jmp 8'HA0` instruction will set `pm_address` to 8'HA0 at the same time the base register changes to 4'H3. Therefore, the program sequencer will make the address for the program memory { 4'H3, 8'HA0 } during the time that `jmp 8'HA0` is in the instruction register. This will cause a jump to address 8'HA0 on Page 3 of memory on the clock edge that executes the `jmp 8'HA0` instruction.

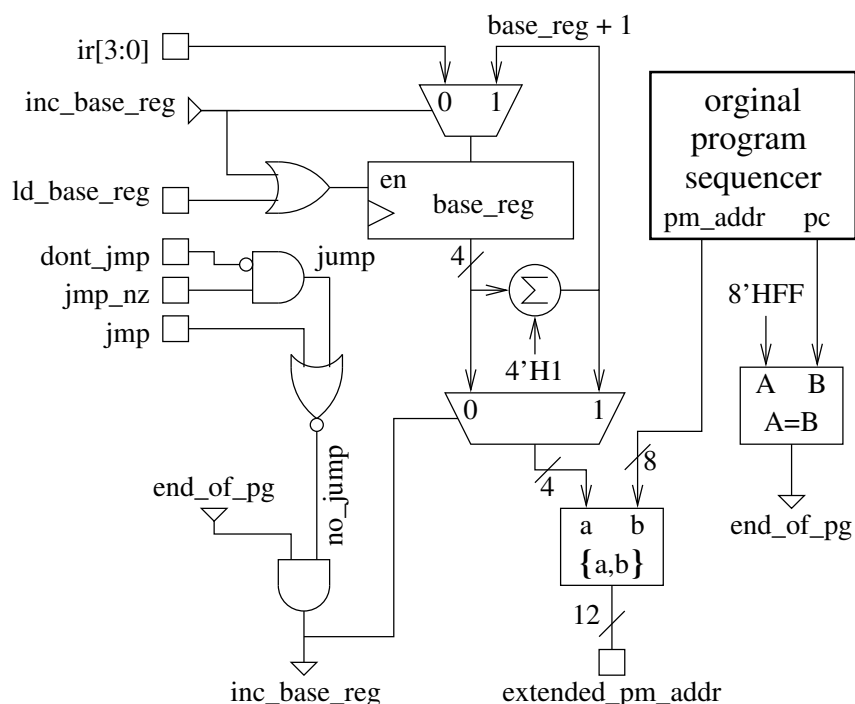


Figure 14: A block diagram of the hardware need to support paging of program memory.

There is a special case, which when the instruction in the instruction register is from the last memory location on a page. That is, the special case arises when $PC == 8'HFF$. Within this special there are two cases to be considered:

1. The instruction is either a `jmp` or a successful `jmp_nz`.
2. The instruction is anything else, which includes an unsuccessful `jmp_nz`.

If that instruction is either a `jmp` or a successful `jmp_nz`, then the jump is executed without changing `base_reg`. However, if the instruction is anything other than a successful jump, the next instruction should be from the first instruction on the next page of memory.

To get the next instruction from address 8'H00 on the next page of program memory, the extended program memory address must be { **base_reg+1**, **pm_addr** }. Of course **pm_addr** will be 8'H00. Also, at time the last instruction on the page is executed and the first instruction on the new page is loaded into the instruction register, the base register must be incremented so that the extended address points to the new page for the next instruction.

A block diagram of the hardware required to support paging is given in Figure 14. Notice that look-ahead logic (i.e. a data selector that selects the inputs to `base_reg`) is used to get the new page number immediately. On the clock edge that executes the last instruction on the old page the base register is incremented so the look-ahead is no longer needed.

```
always @ *
if (pc == 8'HFF) // last instruction of a page
    if(jmp == 1'b1)
        inc_base_reg <= 1'b0;
    else if ( (jmp_nz == 1'b1) && (dont_jmp=1'b0) ) // successful jmp
```

```

        inc_base_reg <= 1'b0;
    else
        inc_base_reg <= 1'b1;
else
    inc_base_reg <= 1'b1;

always @ posedge clk
    if (inc_base_reg == 1'b1)
        base_reg <= base_reg + 4'H1;
    else if (ld_base_reg == 1'b1)
        base_reg <= ir[3:0];
    else
        base_reg <= base_reg;

always @ *
    if (inc_base_reg == 1'b1)
        page_number = page_reg + 4'H1;
    else
        page_number = page_reg;

always @ *
    extended_pm_addr = { page_number, pm_addr};

```

4.7 Variable Length Instruction Set

All instructions in the CME341 microprocessor are 8 bits long and occupy one memory location in program memory. Many of the modern microprocessors have instructions of different lengths, meaning that some instructions occupy one location in program memory while others occupy two or three.

The instruction set for the CME341 microprocessor can be easily converted from a fixed to a multiple length instruction set. The first step is to modify the instruction decoder so that it has auxiliary instruction registers, say the additional 8-bit registers are called `aux_ir_1`, `aux_ir_2` etcetera.

Decoding and executing a multi-word instruction set requires the following:

1. The number of words in a multi-word instruction must be embedded in the first word. That information may be embedded implicitly or given explicitly. For example, if `NOPC8` is re-purposed to be the first word of a 4-word instruction then the length of the instruction is implicitly embedded in the first word. However, if `load o_reg,#4'H?` was re-purposed to be the first word of all of the multi-word instructions and the data field held the instruction length then the length of the instruction would be given explicitly in the first word.
2. The instruction decoder loads the second, third, fourth, etc. words that arrive from data memory into registers `aux_ir_1`, `aux_ir_2`, ect. until all words in the multi-word instruction have been loaded.

3. For instructions with two or more words none of the data registers are enabled until the last word of the instruction has been loaded into an auxiliary register.
4. As soon as the last word of the multi-word instruction is loaded into an auxiliary instruction register the instruction decoder decodes the multi-word instruction stored in the set of instruction registers: `ir`, `aux_ir_1`, `aux_ir_2` etcetera.
5. The instruction is executed with the clock edge that loads the first word of the next instruction into `ir`.

For example suppose the instruction set of the CME341 microprocessor is expanded to include a two word jump-to-subroutine (JSR) instruction. Suppose the first word of the instruction is no-operation 8'HC8 (NOPC8) and the second word is the 8-bit address of the subroutine. The instruction decoder must be modified to add two more outputs: one that signals the program sequence to jump to a subroutine and the second to give the target address. The one bit output will be called `JSR` and the 8-bit address will be called `SR_address`, short for subroutine address. The program sequencer must also be modified to include `JSR` and `SR_address` among its inputs.

The operation of the instruction decoder changes as follows. When the instruction in `ir` is `NOPC8` then the instruction decoder must ensure that the next clock edge loads the next word from program memory into auxiliary instruction register `aux_ir_1` while leaving `ir` unchanged.

Immediately after the last word (which is the second word) of the instruction is loaded into `aux_ir_1`, the instruction decoder asserts `JSR` and leaves it asserted until the arrival of the first word of the next instruction. The instruction decoder also must ensure `SR_address` is valid while `JSR` is high.

The operation of the program sequencer must be modified to jump to `SR_address` when `JSR==1`.

4.8 Wait on External Flag

This feature allows the microprocessor to synchronize with another microprocessor or to read a slow external device that takes multiple clock periods to respond.

A wait-on-external-flag instruction does not have an argument so a no-op instruction can be used for this purpose.

The function of this instruction is to stall the microprocessor until a flip/flop, which will be called `go_flag`, is synchronously set by an external signal. To stall the microprocessor the wait-on-external-flag instruction has to be repeatedly reloaded into the instruction register. This requires the cooperation of the program sequencer as it must freeze `pm_addr` at the address where the wait-on-external-flag instruction is stored.

The stall action stops as soon as `go_flag` goes high. The next instruction is loaded into the `ir` on the first positive clock edge that occurs while `go_flag` is set. The instruction decoder clears `go_flag` on the same clock edge that loads the next instruction in the `ir`.

The program sequencer must make `pm_addr = pc` while the micro is stalling. Otherwise the wait-on-external-flag instruction would not be repeatedly loaded into `ir`. Therefore, the program sequencer needs the instruction decoder to send a `stall` signal that is asserted while a wait-on-external-flag instruction is in `ir` and `go_flag` is low. As soon as `go_flag`

goes high **stall** goes low and the program sequencer will make `pm_addr = pc + 8'H01` so the next instruction will be loaded into `ir` on the next positive clock edge.

4.9 Co-processor

Co-Processors perform one or more complicated tasks independent of the internal operation of the microprocessor. For example a co-processor that calculates the square root of a number would be a circuit external to the microprocessor that could have its input connected to microprocessor output `o_reg` and its output connected to microprocessor input `i_pins`.

It may take several clock cycles for the co-processor to complete its assignment, e.g. compute the square root, which means `i_pins` can not be read and `o_reg` can not be written until the co-processor is finished.

The co-processor sets a `coprocessor_busy` flag as soon as it detects a change in its input and holds that flag high until the answer is valid. While that flag is high the microprocessor can **not** write to `o_reg` or read `i_pins`, but can execute any other instruction.

To handle a co-processor the microprocessor must be modified to include a one bit input called `coprocessor_busy`. If the `coprocessor_busy` flag is 1'b1, then the microprocessor can not write to `o_reg` or read from `i_pins`, but can execute any other instruction. If a “write to `o_reg`” or “read from `i_reg`” instruction is loaded into the instruction register when `coprocessor_busy` is high, then the execution of that instruction must be suspended until the `coprocessor_busy` flag goes low.

This is a special case of waiting on an external flag.

4.10 Data Address Generators (DAGs) for Circular Buffers

There are many algorithms that use a data structure known as a circular buffer. One such algorithm is the FFT. Buffers, which are vectors with each element in the vector being a word in memory, are referenced in high level language as something like `buff[i]`, where `i` is the index. Circular buffers are buffers whose index is computed with modulo arithmetic. That is, the reference `buff[modulo(i,buffer_length)]` converts a buffer to a circular buffer. Performing the modulo operation takes several assembly instructions if it is not done in a hardware circuit referred to as a Data Address Generator (DAG).

A Data Address Generator (DAG) does more than take the modulo of a number. For one thing it auto increments the index every time its block of memory is read or written. A DAG is better described as is a hardware circuit that controls a block of data memory to make it act like an independent block of circular memory.

The compiler of a high level language will allocate a block of memory, referred to as a buffer, to the DAG. It will provide the DAG with the following information in regards to that block of memory:

1. The address of the first word in the buffer (`buff_ptr` for short).
2. The length of the buffer (`buff_len` for short).
3. The initial value of the index.

The auto-increment feature is implemented much like it is in the microprocessor. The microprocessor uses registers `i` and `m` for that purpose. The equivalent registers in the DAG will be called `index` and `incr`, respectively.

NB: To simplify implementation of the DAG the increment, i.e. the value held in `incr`, is restricted to being positive or zero and must be less than `buff_len`. To get a negative increment of N , `incr` must be `buff_len - N`.

From the point of view of the microprocessor the DAG has created another completely separate data memory. Therefore, the instruction set has to be modified to write and read from this apparent second data memory. Instructions equivalent to ‘load `dm`,#H?’, ‘mov `dm`,`reg`’ and ‘mov `reg`,`dm`’ need to be added. These new instruction will be called ‘load `buff`,#H?’, ‘mov `buff`,`reg`’ and ‘mov `reg`,`buff`’.

These new instruction could be created by replacing one of the existing registers with `buff`. For example, `x0` could be removed and all instructions that read or write `x0` could read or write `buff`. Then the ‘load `x0`,#H?’, ‘mov `x0`,`reg`’ and ‘mov `reg`,`x0`’ instructions become ‘load `buff`,#H?’, ‘mov `buff`,`reg`’ and ‘mov `reg`,`buff`’.

Since this buffer is inside data memory its input and output are that of data memory. This means the instruction decoder must enable data memory for writes to `buff`. The instruction decoder would also have to change its `source_sel` for read `buff` instructions. The `data_bus` multiplexer has to select `dm` instead `x0`.

The following hardware registers would normally be part of a DAG.

1. **top_ptr**: A register that holds the address that points to the top of the buffer.
2. **buff_len**: A register that holds the length of the circular buffer.
3. **index**: A register that holds the offset between `top_ptr` and the address of the word in memory of interest.
4. **incr** : A register that holds the amount by which `index` will be incremented immediately after a word in the circular buffer is either written or read.

Additional “load” instructions are needed to load registers `buff_ptr`, `buff_len`, `incr` and `index`. However, to reduce implementation time so a DAG circuit can be implemented during an exam, some of (or all of) registers `buff_ptr`, `buff_len` and `incr` are eliminated by making them constants.

A block diagram of hardware for a DAG is shown in Figure 15.

4.11 Timers: Regular and Watchdog

Yet to be done.

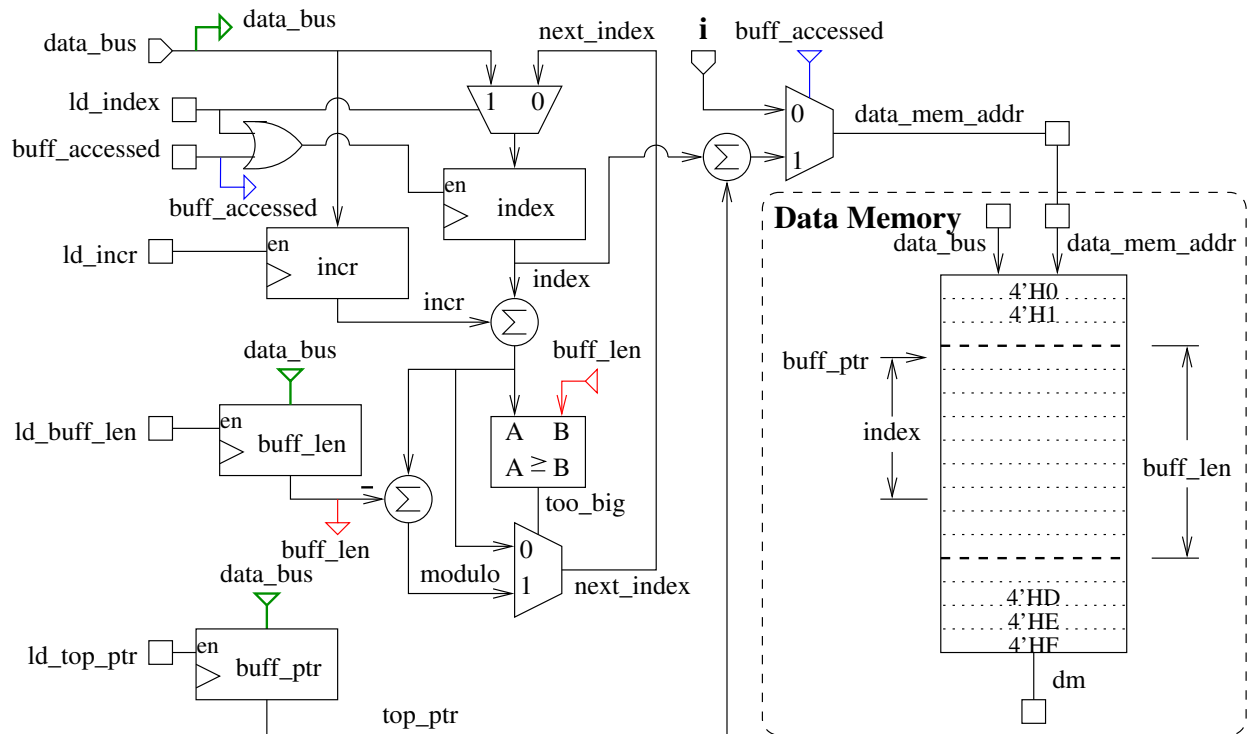


Figure 15: A block diagram for a Data Address Generator (DAG)