CME341 Part III: Designing a Microprocessor

Created by Eric Salt

```
REVISION HISTORY:
April 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.
```

Contents

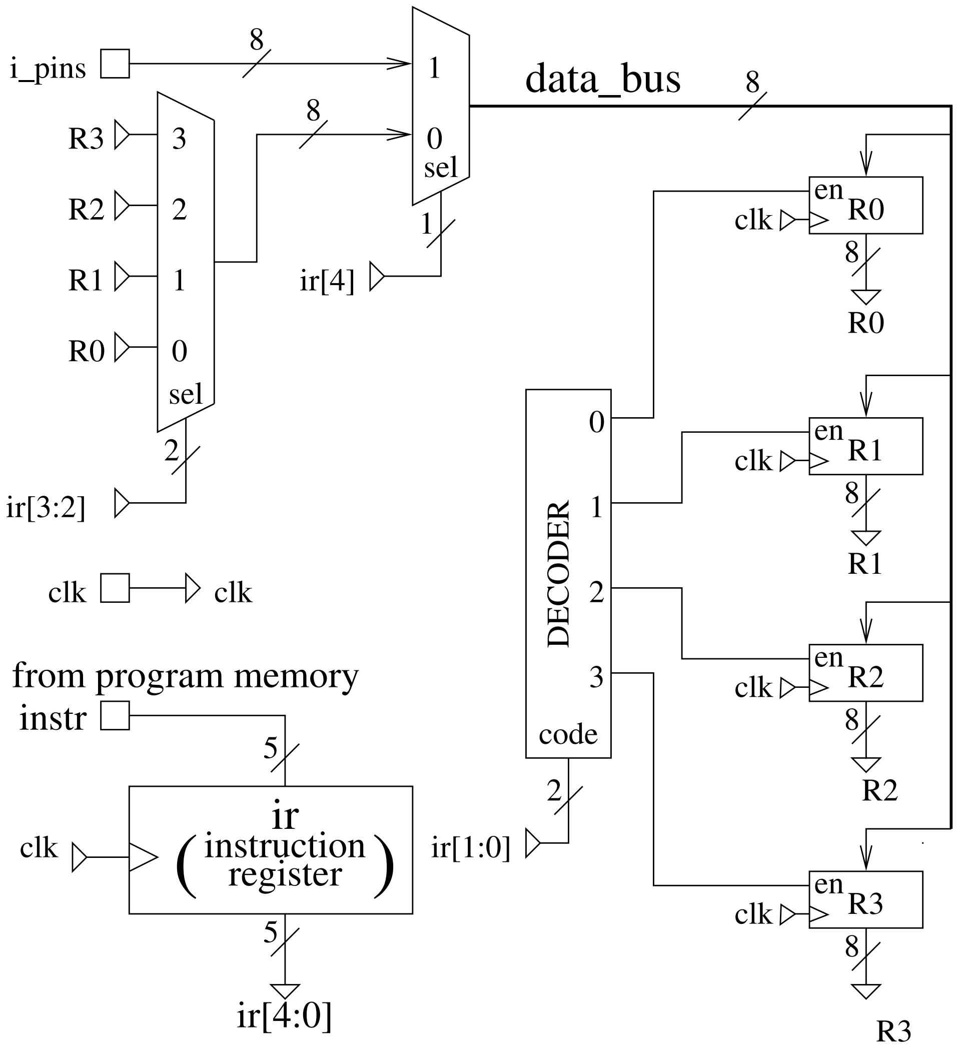
List of Figures

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 18. 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 a 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 19, 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 $r \neq 4'\mathrm{H}0$ (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 $PC + 8'\mathrm{H}01$.

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 21, 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 carry flag in the computational unit is 1'b0. The input `dont_jmp` is connected to the carry flag so the conditional jump is executed if and only if `dont_jmp` is not asserted, i.e. `dont_jmp == 1'b0`. If the conditional jump is to be executed then `pm_addr` must be { `jmp_addr`, 4'H0 }.

   If `dont_jmp == 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.

[1]Referring 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.

3. If inputs `jmp` and `jmp_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 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 circuity 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.

Insert notes from file `extra_notes_on_memory.pdf`

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 19.

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	r	when the ID is used in a source register field
100	o_reg	when the ID is used in a destination register field
101	m	
110	i	
111	dm	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 or 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 is 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_o 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 19 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-opps. The 8 ALU functions and associated codes are described below.

```
ALU function
  000 & ir[3]==0:  r=-x; i.e. two's complement - also used to
                         test for x==4'H0, i.e. set zero flag
                         if x==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; ones complement, which is also used to
                         test for x==4'HF, i.e. set zero flag
                         if x==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 ignor 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.

Insert notes from file `extra_notes_on_instruction_decoder.pdf`

1.2.7 Computational Unit

The essence of the operation of the computational unit is captured in the block diagram on page 22. 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.

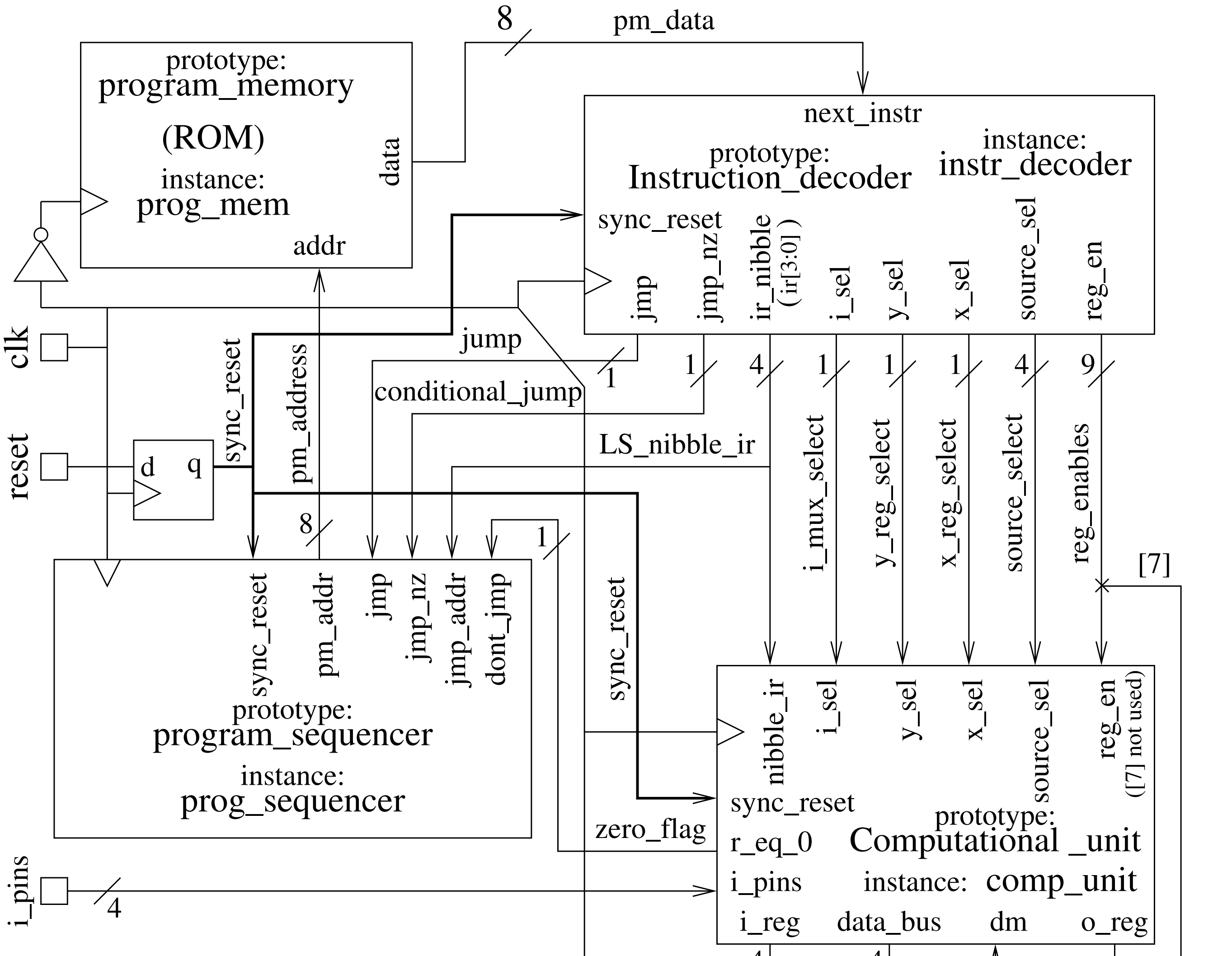
Data Memory

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.

1.2.8 Data Memory

Data memory is synchronous random access memory. Its inputs are registered (both data and address) and it 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



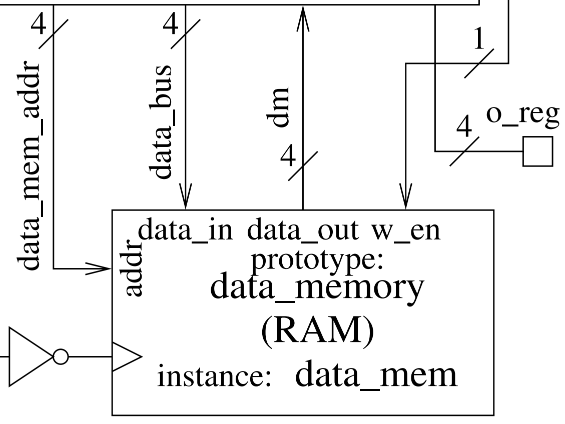
NOTES:

1. The pins, i.e. ☐, denote the signals in the port list of the top module and therefore represent FPGA pins.

2. All bocks 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 conection list with the association ".clk(~clk)".



Microprocessor page 1/5
Block Diagram of Microprocessor

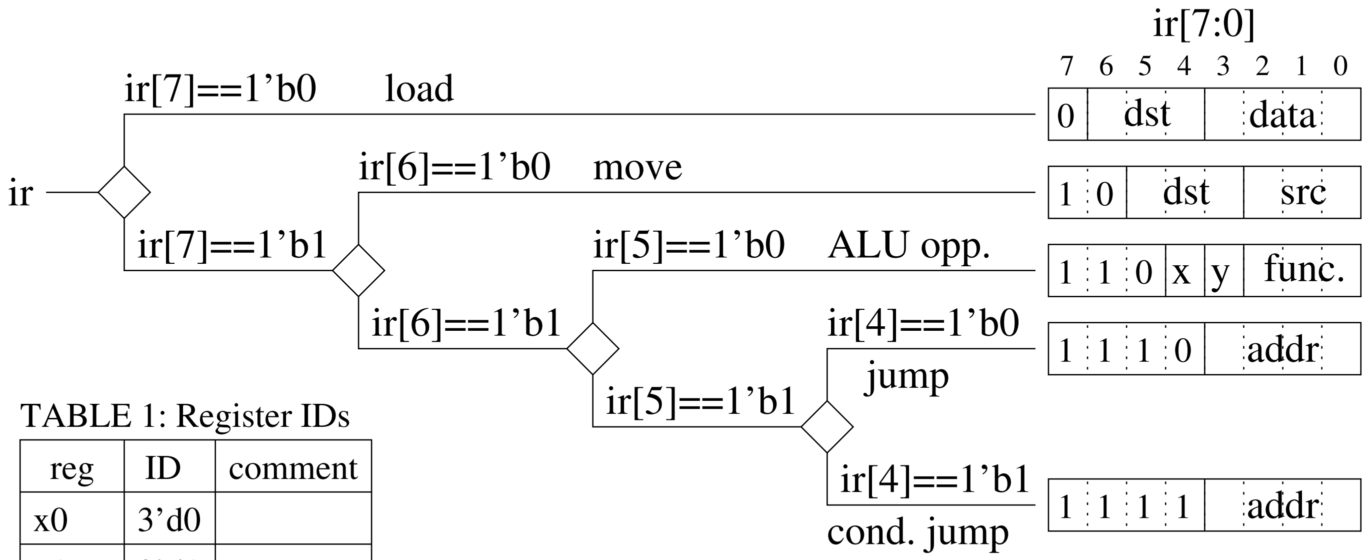


TABLE 1: Register IDs

reg	ID	comment
x0	3'd0	
x1	3'd1	
y0	3'd2	
y1	3'd3	
r	3'd4	src field
o_reg	3'd4	dst field
m	3'd5	
i	3'd6	
dm	3'd7	

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’’.

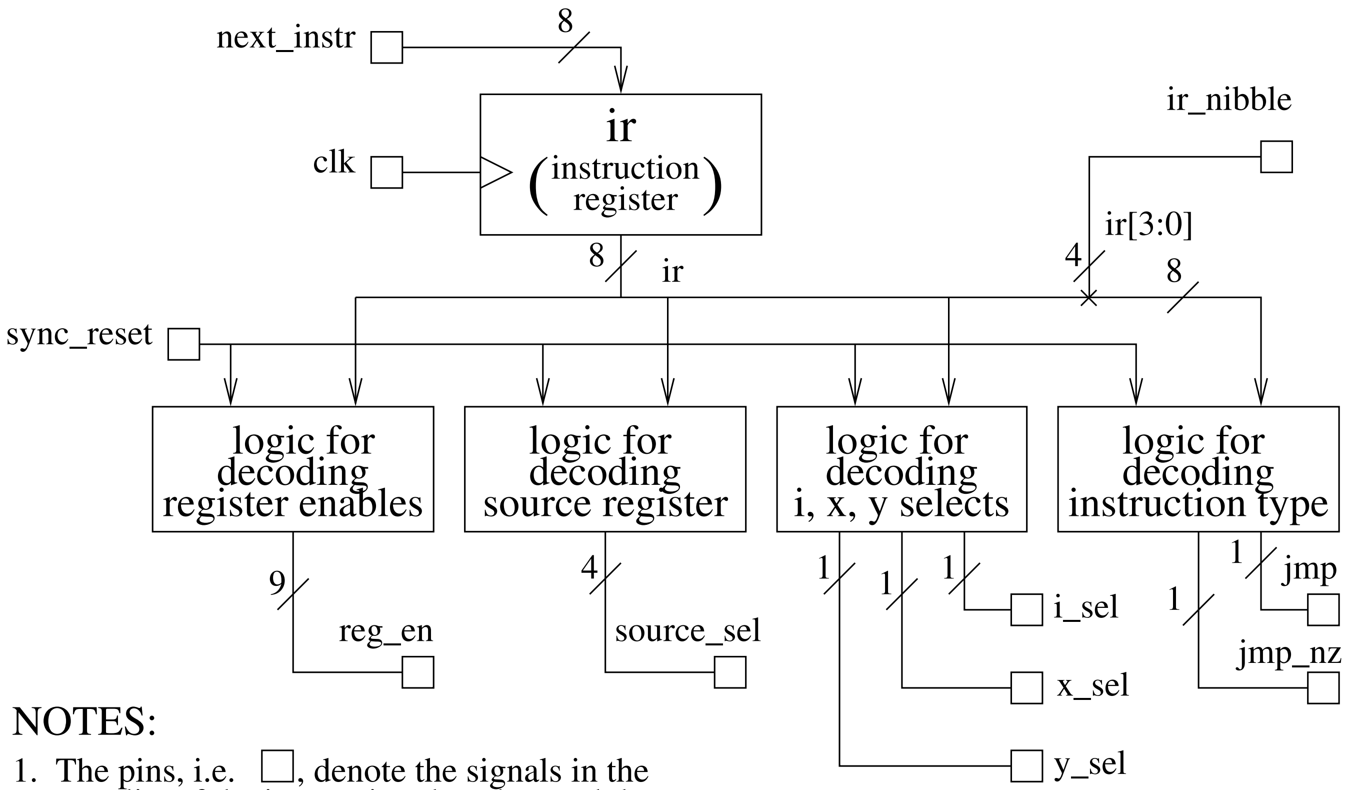
2. Move r to o_reg

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 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’’.

Microprocessor page 2/5
Instruction set for the Microprocessor


next_instr
8
clk
ir
(instruction register)
ir_nibble
ir[3:0]
4
8
8
ir
sync_reset
logic for decoding register enables
logic for decoding source register
logic for decoding i, x, y selects
logic for decoding instruction type
9
reg_en
4
source_sel
1
1
1
i_sel
x_sel
y_sel
1
1
jmp
jmp_nz

NOTES:

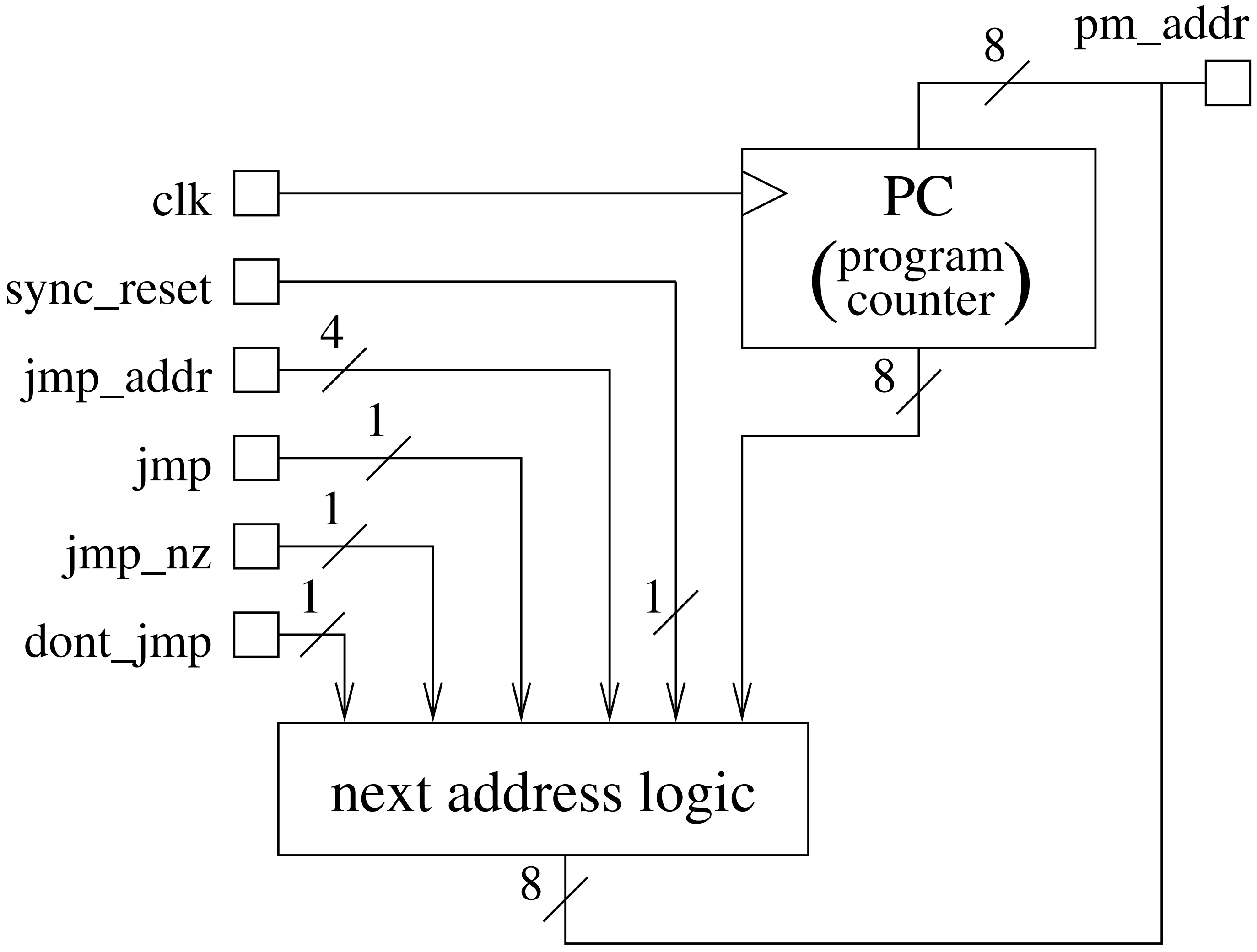
1. The pins, i.e. ☐, denote the signals in the port list of the instruction decoder module.

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]

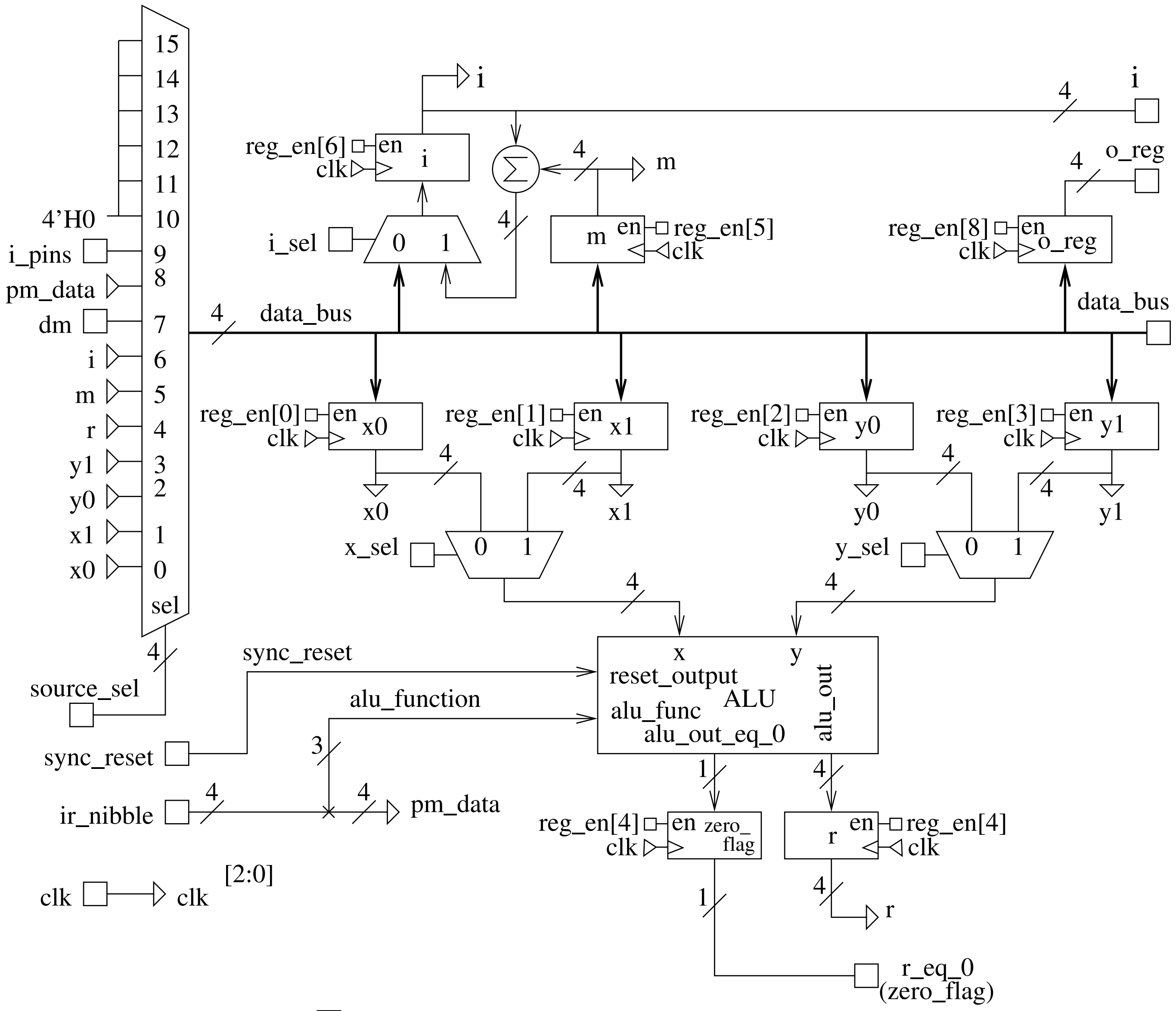
reg_en								
[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
o_reg	dm	i	m	r	y1	y0	x1	x0

Microprocessor page 3/5
Block Diagram of Instruction Decoder



NOTES: The pins, i.e. $\square$, 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

Microprocessor page 5/5
Block Diagram of Computational Unit

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[0],
                     ; i.e. o_reg gets 8,
                     ; i=i+m = 0+(-1) = F
8'H25   o_reg = r; o_reg gets 7
8'H26   jump 8'H30; jump to address 8'H30

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

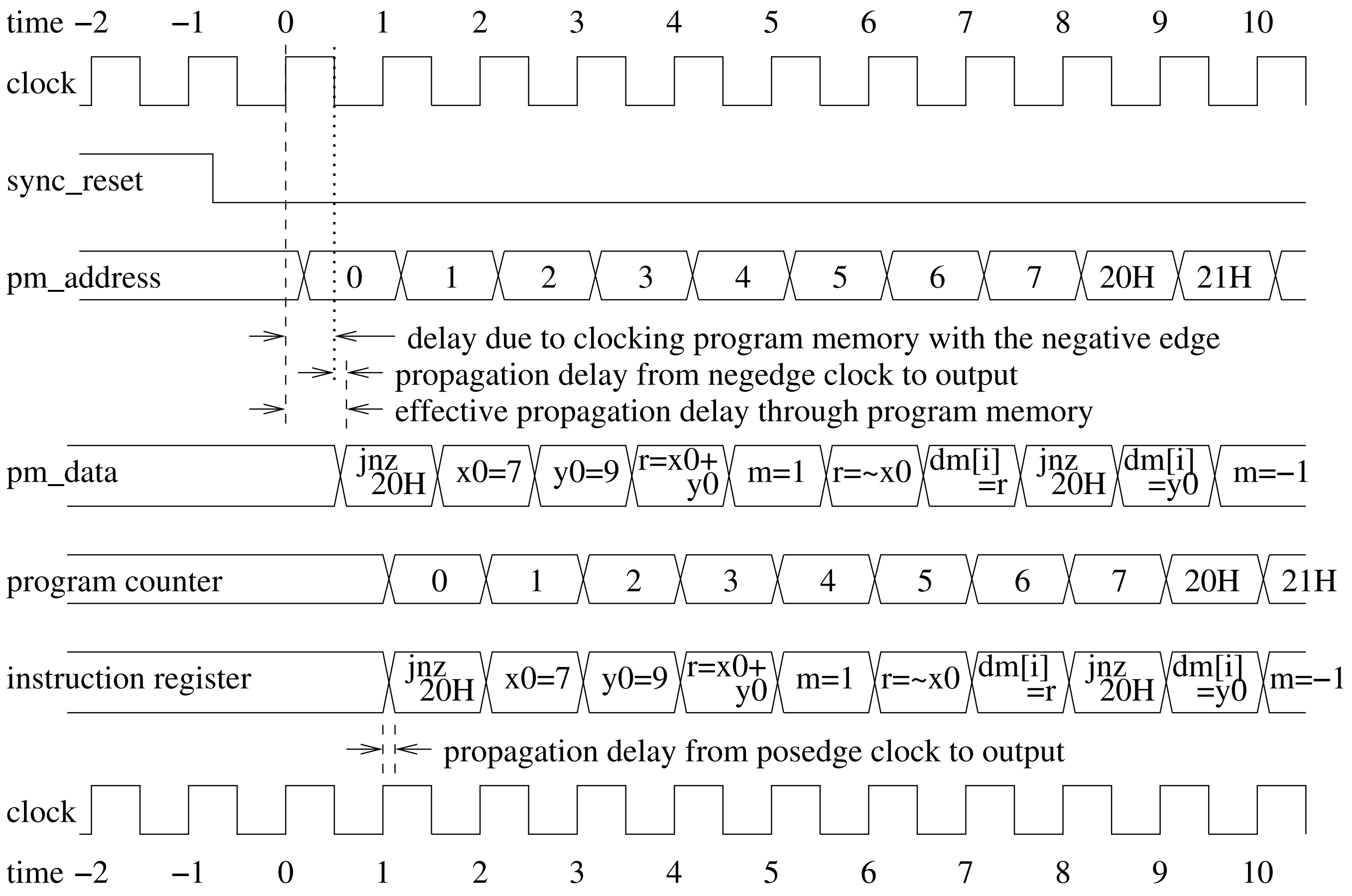

time −2 −1 0 1 2 3 4 5 6 7 8 9 10
clock
sync_reset
pm_address
0
1
2
3
4
5
6
7
20H
21H
delay due to clocking program memory with the negative edge
propagation delay from negedge clock to output
effective propagation delay through program memory
pm_data
jnz 20H
x0=7
y0=9
r=x0+ y0
m=1
r=~x0
dm[i] =r
jnz 20H
dm[i] =y0
m=−1
program counter
0
1
2
3
4
5
6
7
20H
21H
instruction register
jnz 20H
x0=7
y0=9
r=x0+ y0
m=1
r=~x0
dm[i] =r
jnz 20H
dm[i] =y0
m=−1
propagation delay from posedge clock to output
clock
time −2 −1 0 1 2 3 4 5 6 7 8 9 10

Figure 1: Timing for instructions flowing through program memory

Table 1: Template 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									
8’H02	y0 = 4’H9									
8’H03	r = x0 + y0									
8’H04	m = 1									
8’H05	r = ~x0	alu								
8’H06	dm[i] = r								8’b10_111_100	
8’H07	jnz 8’H20									
8’H20	dm[i] = y0			3’H2						
8’H21	m = −1	load								
8’H22	dm[i] = ipins									8’HB6
8’H23	i = dm[i]	move	3’H6							
8’H24	o_reg = dm[i]	move		3’H7						
8’H25	o_reg = r									
8’H26	jump 8’H30									
8’H27	jump 8’H30									

Table 2: Template for calculating `pm_address` from instruction sequence

NOTE: snyc_reset = 1’b0

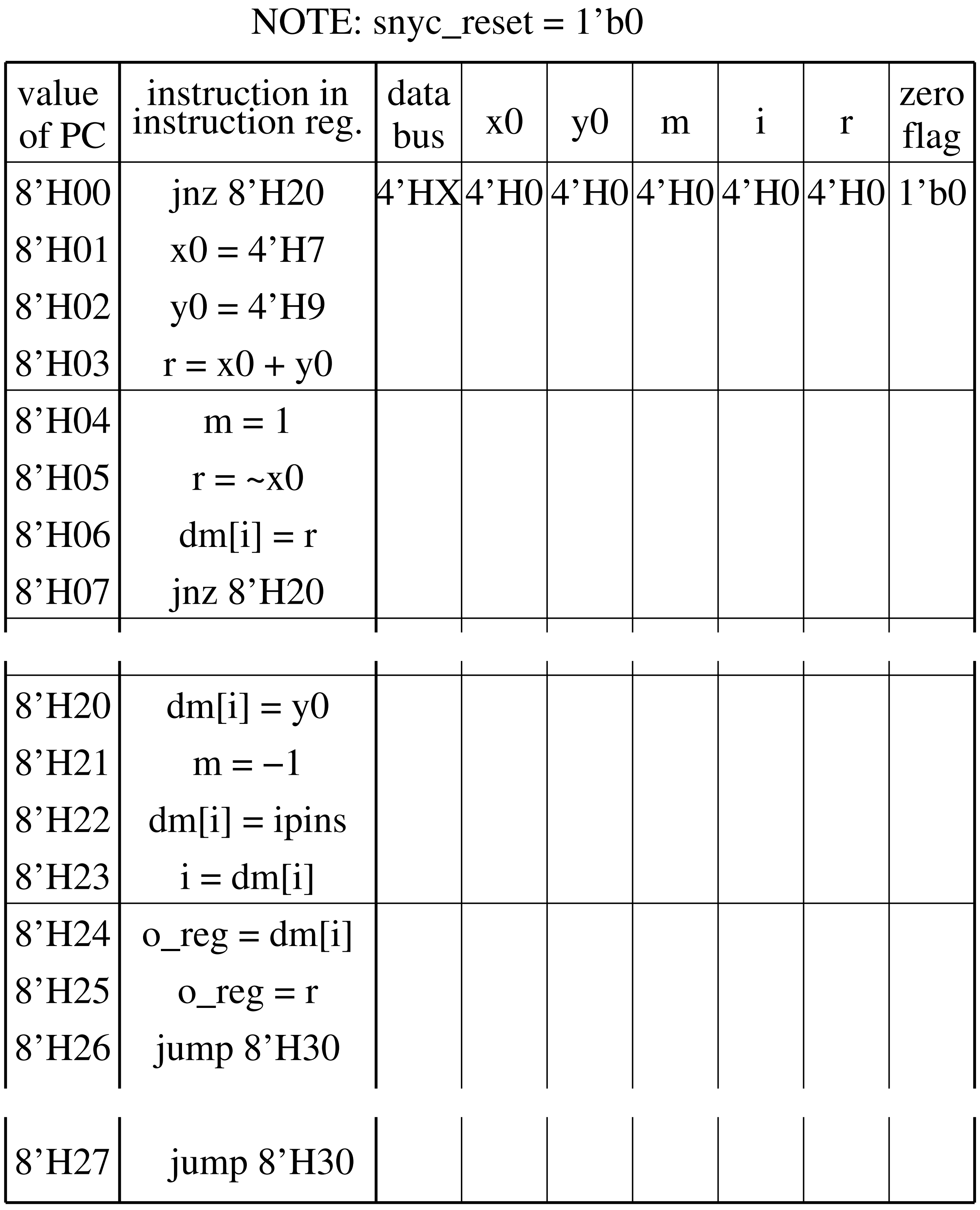
value of PC	instruction in instruction reg.	x0 reg	r reg	zero flag	pm_address
8’H00	jnz 8’H20	4’H0	4’H0	1’b1	
8’H01	x0 = 4’H7	4’H0			
8’H02	y0 = 4’H9	4’H7			
8’H03	r = x0 + y0				
8’H04	m = 1				
8’H05	r = ~x0				
8’H06	dm[i] = r				
8’H07	jnz 8’H20				
8’H20	dm[i] = y0				
8’H21	m = −1				
8’H22	dm[i]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

Table 3: Template for calculating *ir* outputs

NOTE: 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’b0
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’H27	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'H6	3'H6					8'b10_110_110	8'HB6
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'H27	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* outputs

NOTE: snyc_reset = 1'b0

value of PC	instruction in instruction reg.	source select	reg_enables (9 bits)	selects x	selects y	selects i	cond jump	jump
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'b0
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'b0
8'H05	r = ~x0	4'HX			4'H1			
8'H06	dm[i] = r	4'H8					4'H8	1'b1
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'H8		
8'H25	o_reg = r	4'H8				4'H7		
8'H26	jump 8'H30	4'HX						
8'H27	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 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/0 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.

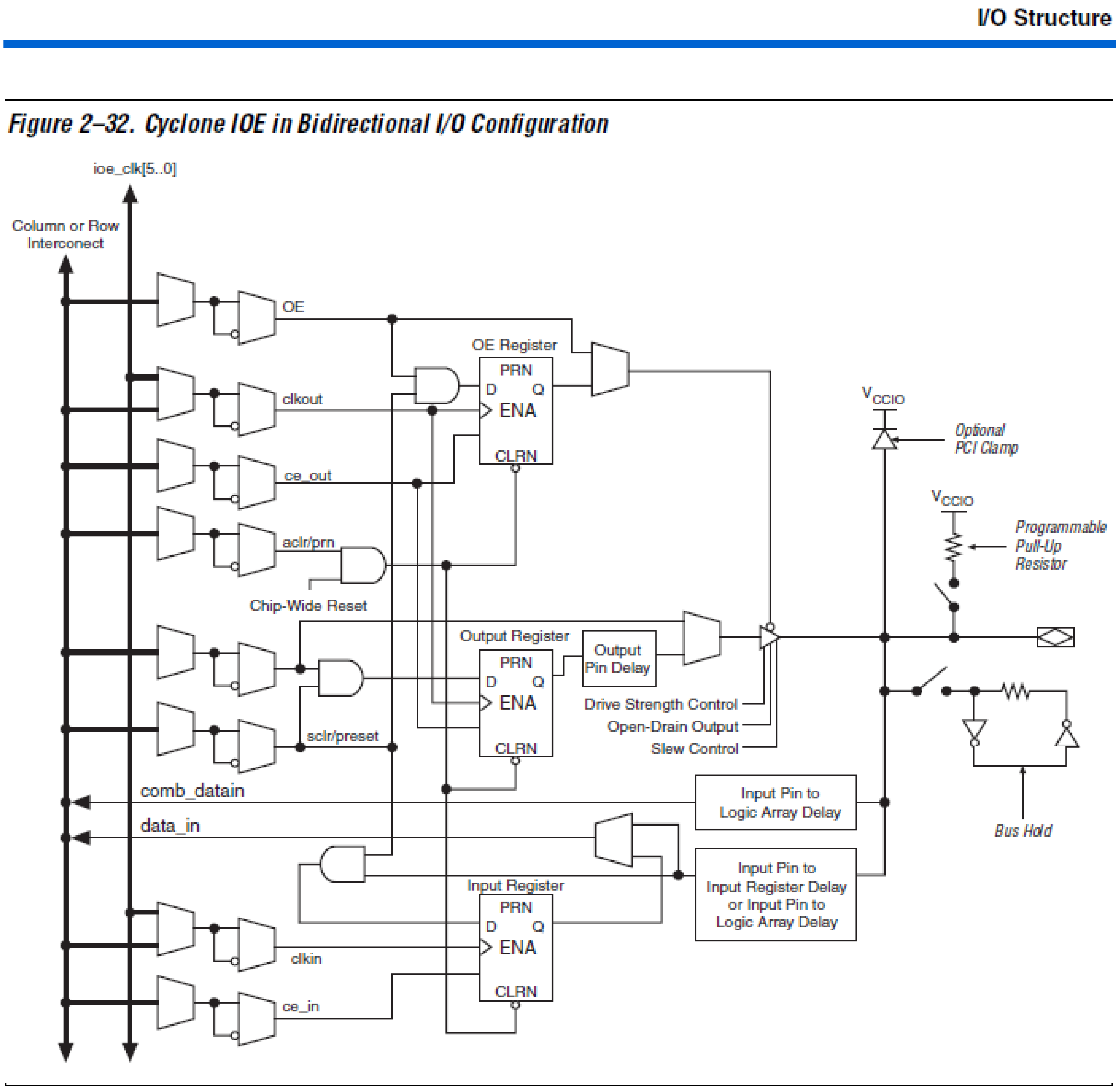


Figure 2: Schematic diagram for a Cyclone I/O element (IOE)

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

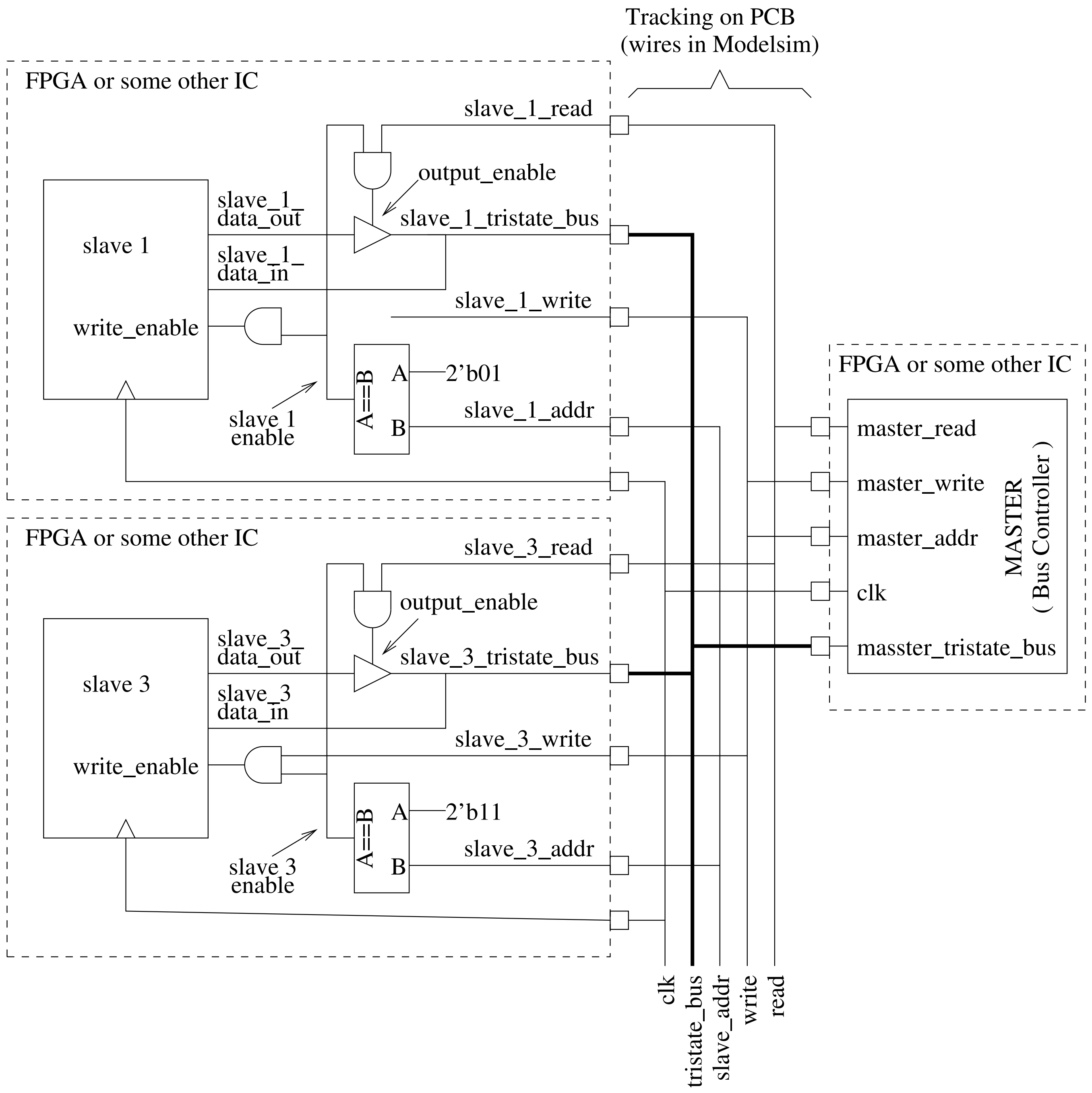


Figure 3: A block diagram of devices on a PCB connected with a tristate bus

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 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

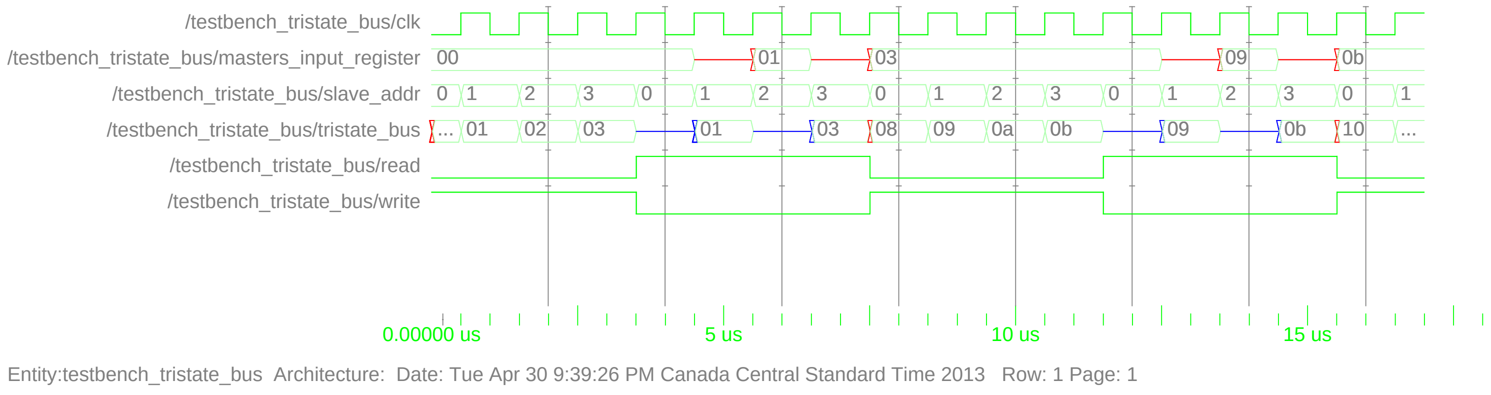


Figure 4: Test bench waveforms for the system in Figure 3

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

3 Foundation for Exam Questions

3.1 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.

The address for the last instruction could be given relative to the address for the first instruction. The relative address is the number of instructions in the loop minus 1.

The information is held in three registers called:
`start_addr, loop_length` and `loop_count`. `start_addr` holds the address for the first instruction in the loop. `loop_length` (which really should be called `loop_length_minus_1`) holds the address for the last instruction in the loop relative to `start_addr`, which is the number of instructions in the loop minus 1. `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

        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;
```

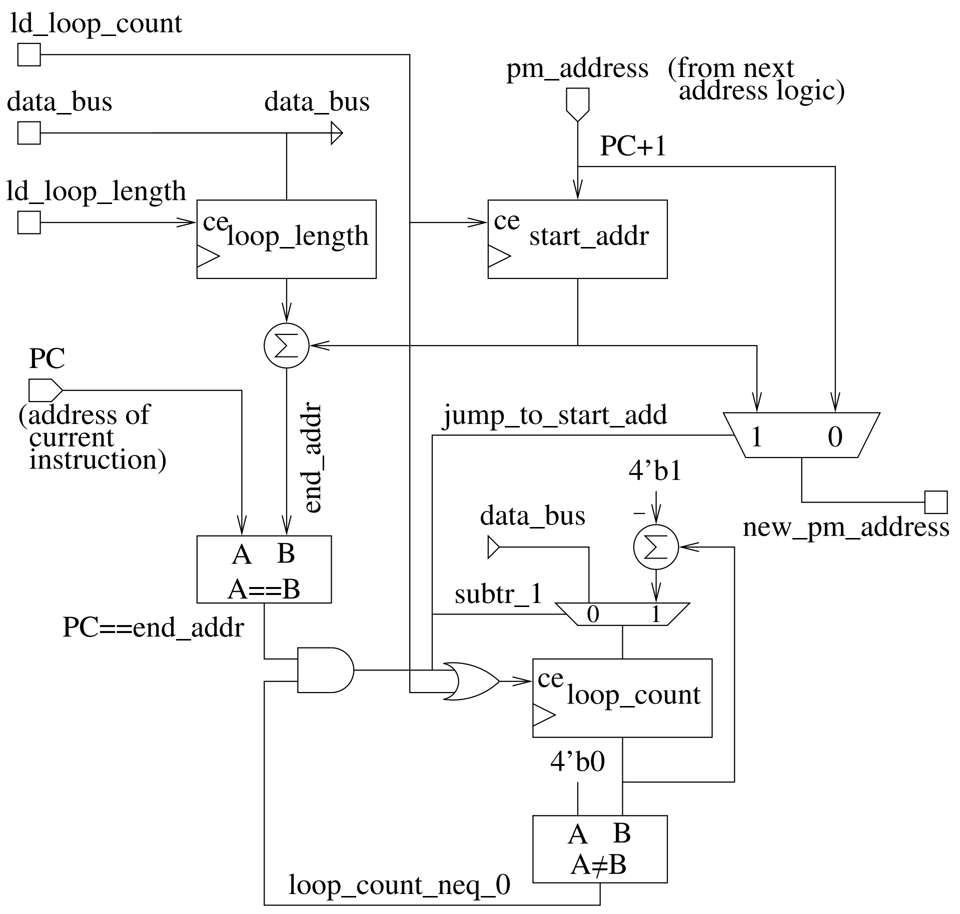


Figure 5: A block diagram of the hardware that supports a simple overhead loop

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 in port list of the program sequencer would have to be expanded to include the data bus as well as two 1-bit signals from the instruction decoder called `ld_loop_length` and ld_loop_count.

The instruction decoder 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. One way is to convert the "load x1" and "load y1" instructions.

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

```
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
```

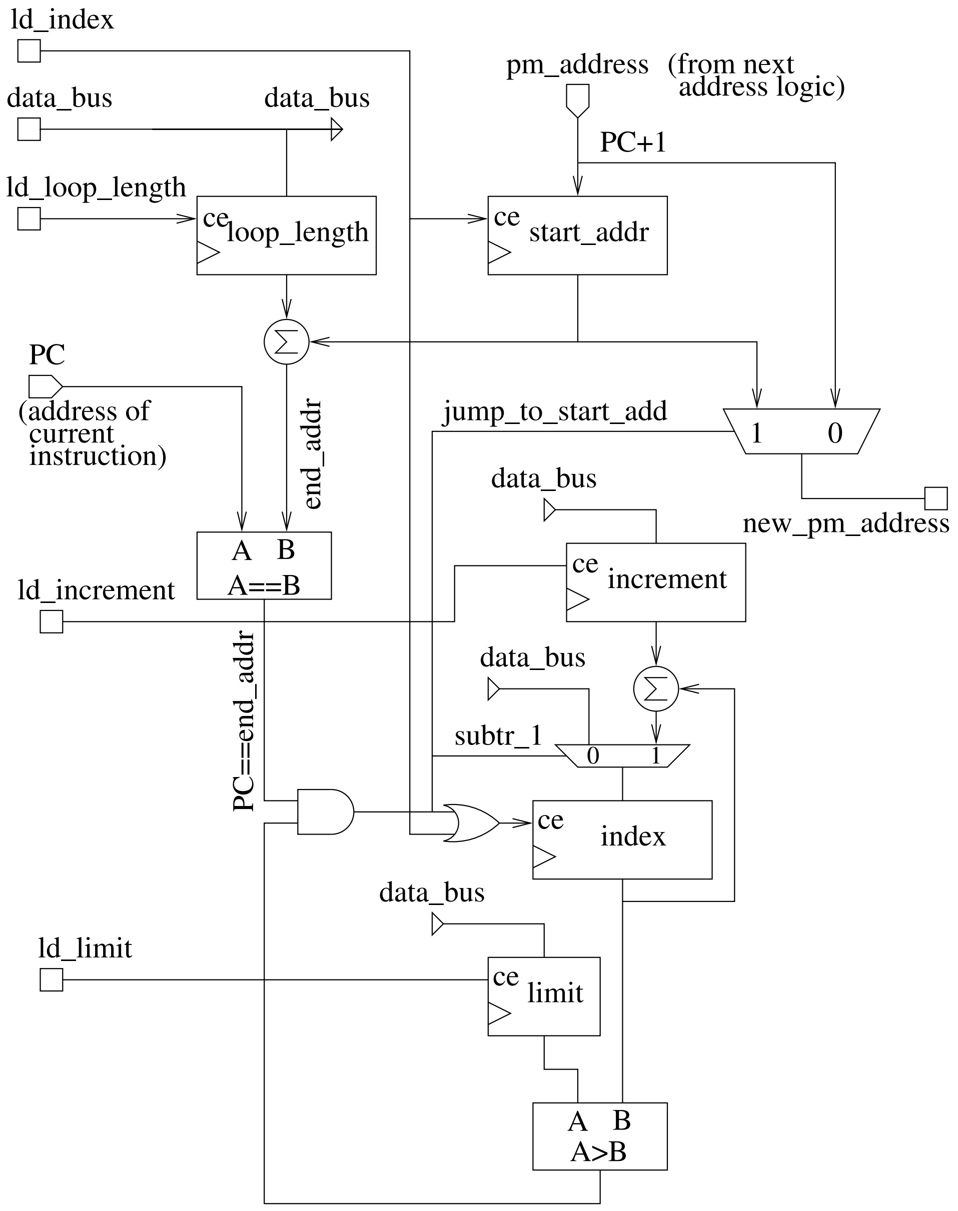


Figure 6: A block diagram of the program-sequencer hardware that supports a simple For-Loop

```
      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 written whenever x is written and with the same value.

This means `shadow_x` is updated with a `mov shadow_x, dm;` instruction and also with a move to or load dm instruction providing the *i* register contains `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 stays in sequence falling through the bottom of the loop to the instruction that follows the last instruction in the loop.

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".

3.2 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 (commonly 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, which is the address of the first instruction in the ISR, is called the interrupt vector. A microprocessor may have more than 1 interrupt pins. 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 (for this class the register will be called `return_addr`) to save the return address. The `return_addr` register is loaded at the time of the interruption with the value that `pm_address` would have had if the hardware interrupt signal was not asserted. For example, if the interrupt signal is asserted while a load, move or ALU instruction is in the instruction decoder, then `return_addr` would be 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 was asserted while a "jump" instruction was in the instruction decoder, then the `return_addr` register would be loaded with the target address of the jump instruction. It would be loaded on the positive edge of the clock that would have executed 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 must be to the interrupt vector. The jump instruction that was interrupted is executed on the return from the interruption, when `pm_address` is made equal to `return_addr`.

Microprocessors with external interrupt capability also must 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 an argument so no-op instructions NOPC8 and NOPD8 can be used.

The hardware interface between the external interrupt pin and the program sequence is shown in Figure 7. 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.

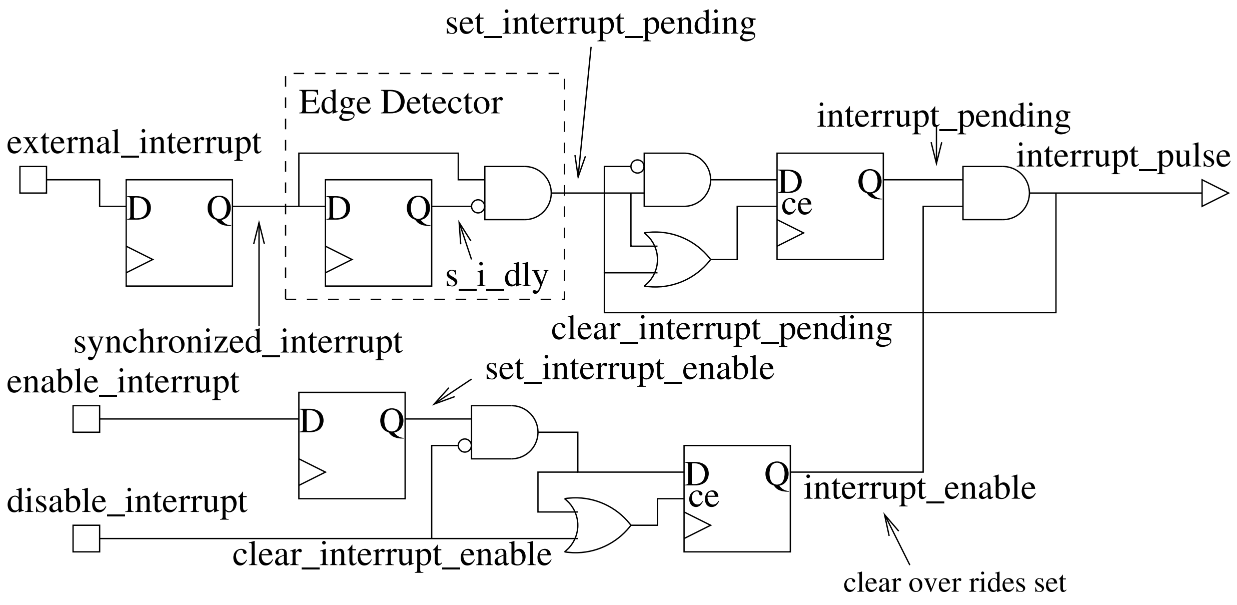
The external interrupt signal is synchronized to the clock using a flip/flop. It is processed by the edge detector to produce a pulse one clock pulse wide each time the synchronized interrupt signal makes a low to high transition.

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)
```



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 opperation 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 7: The interface between the program sequencer and the external interrupt signal

```
synchronized_interrupt = external_interrupt;

always @ (posedge clk)
s_i_dly = synchronized_interrupt;

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;
```

3.3 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 capture the value in the PC every time the memory location in question is written. If the captured value of the PC is inside or outside the address range of interest (i.e. if the statement `address_of_first_instr` $\leq$ `captured_PC` $\leq$ `address_of_last_instr` is true or false) then a monitor interrupt is generated.

Break Immediately

The work "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`.

3.4 Hardware Support for Subroutines

A subroutine call could be considered a software interrupt or a "jump a to interrupt vector" instruction. It is a jump instruction, where the program sequencer has to be modified to store the address of the jump instruction + 1 is stored in registered called `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 Return From Subroutine (RTF) must be added to the instruction set. This instruction causes the program sequencer to make `pm_address = return_addr`.

Using a stack for the return address

The subroutine call feature becomes more powerful if a subroutine can be called from within another subroutine. The ability to do this is called nesting. Every time a subroutine is called the return address must be stored. If N calls are made before an RFS instruction is executed then N return addresses must be stored. The RTS instructions use the return address in the time reverse order in which they were stored.

For example, suppose the subroutine call order is main_program → subroutine_1 → subroutine_2 → subroutine_3. The return addresses stored at the times would be stored in the order return_addr_main_program, return_addr_subroutine_1, return_addr_subroutine_2. Since subroutine_3 is the one currently running, the first RFS instruction encountered would be in subroutine_3. It would return control to subroutine_2, which is the subroutine that called it, using return_addr_subroutine_2 . Then subroutine_2 would return control to subroutine_1, which is the subroutine that called it, using return_addr_subroutine_1. Finally subroutine_1 would return control to main_program, which is the subroutine that called it, using return_addr_main_program. Clearly the return addresses are used in the reverse time order in which they were stored.

The order in which the return addresses are stored and retrieved allows them to be stored on a last-in-first-out (LIFO) queue. Such queues are commonly called stacks. The act of placing data on and removing data from a stack is called pushing and popping. An instruction that places data on the stack would have a mnemonic like "push". If one of several possible signals could be placed on the stack the push instruction would have a argument to indicate which signal is to be pushed and therefore would have a mnemonic like "push signal_name". A "push signal_name" instruction could not be one of the four no-opp instructions (i.e. an instruction with machine code 8'HC8, 8'HD8, 8'HCF or 8'HDF) unless the argument indicating which signal was to be loaded was held in one of the microprocessor's registers.

A stack that stores the return addresses for jumps to subroutines does not respond to an explicit "push" instruction. The push is implied in the "Jump to SubRoutine" (JSR or JTS which comes from Jump To Subroutine) instruction. That is to say the instruction decoder makes a signal called `push` or push_return_address that indicates a JSR instruction is in the instruction register. The push would be executed (i.e. the return address written to the stack) on the same clock edge that executes the JSR instruction. The instruction that removes the newest of the data entries still stored on the stack has a mnemonic like "pop". Sometimes a "pop" instruction is combined with an instruction that moves the TOS to a register. The "move TOS to reg_name and pop" instruction would have a mnemonic like "pop reg_name ". If the TOS is an input to just one circuit, whether that circuit is a register or not, there is no need to specify "reg_name". In this case a simple "pop" instruction is all that is needed. Since there is no argument for a "pop" instruction, one of the no-opp instructions (machine codes 8'HC8, 8'HD8, 8'HCF or 8'HDF) could be used for it.

The "pop" instruction for a stack storing return addresses is implied in the RFS (Return From Subroutine) instruction. Of course the RFS instruction is more than just a pop. It also assigns `pm_address = TOS` while the RTS instruction is in the instruction register.

A stack can be build using a synchronous two port RAM as storage. One port is used for pushing and the other is used for reading the TOS (reading is usually part of the popping although the TOS could be read on one instruction and popped on the next). Additional circuitry is needed to control the addresses of the two ports so that the popping happens in the reverse order of the pushing.

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). This means it is the address of the "pop" port of the RAM. 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 (data entry on the Top Of the Stack)".

In is very important to understand that synchronous RAMs have registered inputs and that these registers introduce latency. This latency does not cause any problems for writing, but does 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. In the case of a stack, the address, which is `TOS_address`*, changes on both push and pop instructions. This means if the RAM is clocked with the system clock (say everything is clocked on the positive edge),* **a pop instruction can not immediately follow a push or pop instruction**.

This has a couple of implications. One is an RFS (pop) can not immediately follow a JSR (push), which means the first instruction in a subroutine can not be RFS. Another is a RFS (pop) can not immediately follow a RFS (pop). This means if a subroutine is called from within a subroutine, the JSR can not be followed by RFS since the last instruction in the subroutine that was called would be a RFS.

If a hardware stack was built with synchronous RAM in a microprocessor, then compiler/assembler for that microprocessor will make sure a pop doesn't follow a push or a pop. If the programmer has a RFS immediately following a JSR, the assembler will insert a NO-OPP between them.

The port list for the synchronous RAM is illustrated in block diagram form in Figure 8. The top port, which have names ending in '_a', is the push port and the bottom port, which have names ending in '_b', is the pop port.

The pop port address is connected to `TOS_address`, which is the output of the pointer register that holds the address for TOS. The output of the pop port, which is connected to TOS, is the data entry on the top of the stack. The RAM is not written through the pop port so the pop port write enable must be 1'b0.

The address for the push port is one higher than the pop port address, i.e. it is `TOS_address + 1`. The push port address is for the next data entry. The write enable for the push port is connected to a signal called push, which is 1'b1 if and only if a push or implied push instruction is in the instruction register. The data that is to be pushed on the stack, which for a return address stack would be the return address for the subroutine called, is connected to the data input of the push port. The output of the push port is never read so is not connected to anything.

In setting up a stack it is helpful to define a constant called `BOS_address_minus_1` or perhaps `BOS_ADDRESS_MINUS_1`). It defines the address of the first location in the stack, but does so by point to the location 1 below the location for the first data entry. It is defined this way to simply the circuit.

The constant `BOS_address_minus_1` could be set by the programmer, in which case it must held in a pointer register. Of course a new load or mov instruction must be added to the instruction set to write the constant to the `BOS_address_minus_1` pointer register. Alternatively, the constant could be defined in hardware. If it is defined in hardware there is be no need for a `BOS_address_minus_1` pointer register and no need for new instructions. If it was defined as a local parameter in Verilog HDL, one may capitalize all the letters, i.e. it

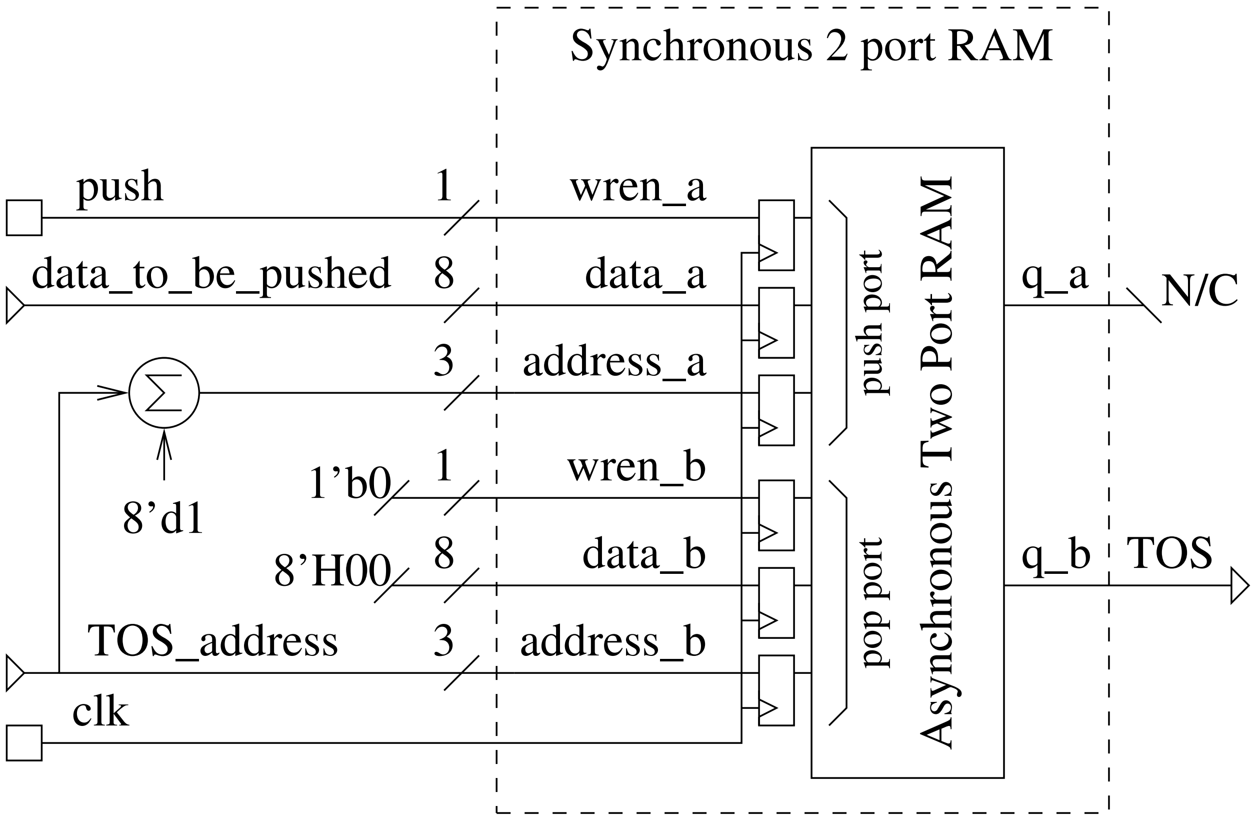


Figure 8: Connections that Convert an 8-word by 8-bits/word 2-port RAM to a stack

may be called BOS_ADDRESS_MINUS_1. The most likely value for BOS_ADDRESS_MINUS_1 would be -1, which would give the bottom of the stack an address of 0.

The Verilog HLD description of the logic that updates the pointer register called TOS_address is given below. The stack is assumed to have a depth of 8 words (i.e. 8 word RAM having 3 address bits). It uses the three signals called JTS, RFS and load_TOS_address defined as follows:

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.

load_TOS_address is 1'b1 if and only if a load pointer register TOS_address is in the instruction register.

```
always@ (posedge clk)
   if (load_TOS_address) // initialize TOS_address
      TOS_address = BOS_ADDRESS_MINUS_1;
   else if (JTS)  // push the return address
      TOS_address = TOS_address + 8'd1;
   else if (RFS) // pop the return address
      if (TOS_address == BOS_ADDRESS_MINUS_1) // underflow
           TOS_address = TOS_address; // don't allow pop
      else
           TOS_address = TOS_address-3'd1; // execute pop
   else
      TOS_address = TOS_address;
```

3.5 Extending the Address Range for Memory

The program sequencer has been designed to support a 256 word memory. The address range for memory can be extended in a variety of ways. One way would be to extend pm_address and the PC to 9 bits. Of course there would be some ripple effects. For one thing the jump instruction would have to be redefined since the way it is currently defined all jumps would to the lower 256 words.

Another method of extending memory is called paging. The expanded memory is divided into pages of 256 words. The length of the extended memory would be a multiple of 256 words. 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 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 extra most significant bits would come from a new register called the base register.

The base register contains the most significant address bits that are in excess of 8. In essence it holds the page number of the current page of memory. To change pages in memory requires a special instruction that writes the base register. A tranfer from one page to another would be accomplished by the pair of instructions given below.

```
    load base_reg, #4'H3;  base register will be loaded
                        ;  with 3 when the next
                        ;  instruction is executed
    jump 8'HA0;  base register is loaded with 4'H3
              ;  at the same time pm_address is
              ;  set to 8'HA0
```

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 `jump 8'HA0` instruction is loaded into the instruction register. The `jump 8'HA0` instruction will set `pm_address` to 8'HA0 at the same time the base register changes to 4'H3. The program sequencer will make `pm_address` { 4'H3, 8'HA0 }. This cause a jump to address 8'HA0 on Page 3 of memory on the next clock edge.

There is a "last_instruction_on_a_page" situation that must be considered. That is the situation where the last instruction on a page is executed i.e. any instruction where `PC == 8'HFF`. If that instruction is anything other than a successful jump, the next instruction in the sequence should come from the first instruction on the next page.

In this special case, if the instruction being executed is anything but a successful jump, `pm_address` must be made 8'H00 and the base register must be incremented. The problem is the base register in not incremented until the next clock edge so the increment, which is yet to come, will not influence the current value of `pm_address`. This problem is remedied by placing some logic between the base register and the program memory address lines that specify the page number. This logic is given below

```
always @ *
if (last_instruction_on_a_page)
     if (successful_jump)
        page_number = base_register;
     else
        page_number = base_register + 4'b1;
else
     page_number = base_register;
```

3.6 Variable Length Instruction Set

Yet to be done.

3.7 Wait on External Flag

This feature allows the microprocessor to read a slow external device that takes multiple clock periods to respond. For example, external memory usually runs much slower than the microprocessor. In this case an instruction that moves data from external memory to a register can not be done in one clock cycle. A memory read instruction would have to remain in the instruction decoder without executing until the memory raises a flag indicating "result ready".

The memory would have to know when a read was requested and lower the "result ready" flag until the result was ready. Of course once a result is ready it remains ready until the memory address changes. This means the memory could lower it flag each time the address changes and then raise it some predetermined time later. Another way has the micro issuing a "read request" when a memory read instruction is in the instruction decoder and making the "results ready" flag a delayed version of the "read request".

3.8 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.

3.9 Data Address Generators (DAGs) for Circular Buffers

A Data Address Generator (DAG) is a hardware circuit that controls a block of memory to make it act like an independent circular memory (also referred to as a circular buffer). The block of memory is addressed indirectly through a register in the DAG that would be called something like `buffer_pointer` (Any register used for indirect addressing is called a pointer register).

The following hardware registers must be added to the microprocessor.

1. **bottom_of_buffer_pointer**: A register that hold the memory address where the circular buffer starts.

2. **buffer_length_reg**: A register that holds the length of the circular buffer. That is to say the address for the top of the buffer is
`bottom_of_buffer_pointer + buffer_length_reg - 1`.

3. **buffer_pointer**: A register that holds the memory address of the next word in the memory buffer that will be either written or read. The values in **buffer_pointer** are restricted to being one of the addresses for the block of memory that makes up the circular buffer.

4. **increment_reg** : A register that holds the amount by which `buffer_pointer` will be incremented immediately after a word in the circular buffer is either written or read.

The operation or a DAG requires the following instructions be added to the instruction set.

1. Load `bottom_of_buffer_pointer`. This instruction simultaneously loads `buffer_pointer` with the same value. It defines the lowest address of the block of memory that makes up the circular buffer.
2. Load `buffer_length_reg`. This defines the length of the circular buffer.
3. Load `increment_reg`.
4. Load indirect `buffer_pointer`. This loads a constant into the memory location pointed to by the address in `buffer_pointer` and increments `buffer_pointer` by the amount in `increment_reg`.
5. Move indirect from `buffer_pointer` to a data register. This moves the data in the memory location with the address in `buffer_pointer` to a data register and increments `buffer_pointer` by the amount in `increment_reg`.
6. Move indirect from a data register to `buffer_pointer`.This moves the data in a data register to the memory location with the address in `buffer_pointer` and increments `buffer_pointer` by the amount in `increment_reg`.

Each time a memory location is written or read indirectly through `buffer_pointer` the contents of `buffer_pointer` (which is a memory address) is incremented by the amount in `increment_reg`. If the calculated value for the next buffer pointer, which is
`calculated_buffer_pointer = buffer_pointer + increment_reg`
points to a memory location outside of the circular buffer, i.e. if
`calculated_buffer_pointer` $\geq$ `bottom_of_buffer_pointer + buffer_length_reg`,
then
`buffer_pointer = calculated_buffer_pointer - buffer_length_reg`

3.10 Timers: Regular and Watchdog

Yet to be done.