

Final Project #1: Vending Machine Controller

Objective: To design a fully functional vending machine controller (VMC) in VHDL. Students will apply their digital design knowledge and various FSM principles to implement the hardware components necessary to realize a coin operated VMC based on a set of specifications. The machine will support four modes of operation: vending mode, program mode, display mode, and free mode. Consumers will use the vending and display mode, and vending machine operators the program, display, and free mode. Distinction between consumer and user will not be necessary in the context of this project.

1) Vending Machine Controller Overview

Inputs: clock, reset, hard_reset, start, set N, D, Q, funct, prod

Outputs: change0, change1, change2, runTotal0, runTotal1, runTotal2, total0, total1, total2, finished

States: idle, hardReset, program, display, vend, free

The vending machine controller (VMC) is a sequential, active-high circuit. The VMC may be set to a certain mode using the “**Start**” input, simultaneously applied with a “**funct**” (Function) input code. Accordingly, the start input signal will initiate the VMC to enter the “funct” specified mode of operation. The funct mode encodings are as follows:

- 0) Program Mode: program the price of a given product (operator)
- 1) Display Mode: display the price of a product (operator/user)
- 2) Vending Mode: purchase the selected product (user)
- 3) Free Mode: dispenses the selected product for free (operator)

When the VMC is set into the **vending** or **program** mode, the user specifies the product using input “prod”, and the controller thereafter accepts three types of coins: Nickels (N), Dimes (D) and Quarters (Q). The N, D, and Q coins each have their own input slot in the vending machine, and hence the coins may be input simultaneously or individually by the user. An input called “set” applied during program mode will signal to the VMC that the user has finished inserting coins.

The **outputs** of the VMC will present the user with an indication of how much a product costs (total0-2), the running total (runTotal0-2 i.e. coins inserted into the VMC), and change (change0-2) returned to the user. A “finished” signal will be output to the user to indicate that the operation has been successfully completed, and that the product has been dispensed (if applicable). For the sake of simplicity, we will allow our VMC to support up to four products only.

Program Mode

A product may be setup for dispensing in the VMC using program mode. That is, the product’s price may be programmed in the Program Mode indicating that the product exists. Program mode may be entered by asserting **start** and specifying **funct** = 0. Using the inputs **N**, **D**, and **Q**, the user will input a sequence of coins which will accumulate and stipulate the price of the specified product until **set** is asserted. An internal accumulator will tally the coins, where the accumulated total cost will be stored to an on-chip memory unit. The product number specified will represent the memory address for which the product’s price (total) will be stored. Therefore, each product is associated with a memory address, and the corresponding memory cell will store the price of the specified product. Since our VMC supports up to

four products, we will have an on-chip memory consisting of 4 memory cells, each n-bit wide. We will discuss the n-bit datawidth in subsequent sections.

Display Mode

Display mode may be entered by the VMC operator by asserting **start** and the **Display mode (funct = 1)**. In display mode, the operator or user may specify a product and view the price. In the context of the operator, they may confirm the correct price(s) was stored to memory for a given product(s).

Vending Mode

The vending mode may be entered by asserting the **start** signal and specifying the **Vending Mode (funct = 2)**. The user must then specifies the product for purchasing using the input **prod**. Thereafter in the next clock cycle, the user inserts coins into the **N**, **D**, and **Q** slot where a running total is displayed, along with the product's price on a separate display. Once enough money has been inserted, the product will be vended (signalled by the **finished** output), and will return any change if required; change will display on a 3rd display. The VMC will not vend the product if not enough coins have been inserted into the machine. In this case, the VMC will remain in Vend mode until enough coins have been inserted to purchase the product, or a soft reset is asserted.

Free Mode

Free mode may be entered when the user asserts input **start** and selects **funct = 3**. The program will vend the selected product for free, where **no coins** need to be inserted. All totals/change/runningTotal outputs will be placed into a **floating state** when free mode is entered, and a finished signal will be asserted a clock cycle after, lasting one cycle, signifying that the product was dispensed.

Soft and Hard Resets

There will be two types of resets supported by the VMC: 1) soft reset and 2) hard reset. The soft reset will allow the user/operator to return to the idle state, such that the user may switch out of the mode if they decided not to purchase the product, or complete the procedure required of a given state. In this case, all other information will remain intact, such as the product prices stored in on-chip memory.

In the case of a hard reset, the memory will also be wiped from the system, where the VMC will thereafter return to the idle state. In other words, a hard reset is a factory reset.

Accordingly, there will be two reset inputs supported by the VMC: **hard_reset**, and **reset** (i.e. soft reset). The soft reset should be implemented asynchronously, and the hard reset synchronously.

Logic Efficiency

To make our design as efficient as possible while requiring less memory storage requirements, we will store values as multiples of five. That is, if a quarter is inserted into the system, the VMC will interpret and/or accumulate/store the value as a 5, whereas a nickel will be stored/processed as 1, and a dime as 2. Accordingly, your VMC will require a technique to convert the inserted coins to this internal definition, and similarly an output function which transforms the internal representation back to the original number to display to the user (i.e. running total, the product's cost, and change due). In this case, the internal binary representation will also require further processing to be displayed on the set of seven-segment displays.

2) System Specifications

The vending machine will consist of 5 main components depending on your implementation: program_unit, sram, convert2bcd, vend_unit, and the FSM required to control the VMC's operational modes. You will be required to determine any specifications that have been purposely left out, such as port/signal datawidths and types. The rest of the implementation is up to you; you must have a justification for your design choices. The following outlines specifications for each unit:

A) Program Unit – implements the VMC's program mode

Inputs: clock, reset, hard reset, set, enable, product, NDQ

Outputs: data_mem, addr_out, done, wen

States: idle, adding, mem_writing

Other components: accumulator

State	Purpose/State Events
Idle	Output displays are set to zero values when unit is idle (i.e. not in use). Proceeds to the Adding state when enable = 1, simultaneously enabling the accumulator for the next cycle
Adding	User enters coins into NDQ on every cycle until the programmed price is reached. The accumulator unit keeps a tally of the coins inserted. When the user/operator asserts "set", this means that the accumulator reflects the product's correct price. Therefore, once set is asserted: 1) the value maintained by the accumulator is written to the data_mem output. This info is accompanied with the wen assertion (memory write enable), and the address of the product (addr_out). 2) The accumulator must be disabled from counting any further (note we are assuming NDQ will be "000" when set is asserted by the user). 3) The unit enters the mem_writing state
mem_writing	This state solely awaits 'x' cycles necessary for the product price to be successfully written to on-chip memory. The done signal is asserted and aligned with the successful write, and the idle state is entered in the next cycle.

Accumulator – hardware unit used to accumulate coin inputs

Inputs: clock, reset, en, N, D, Q

Outputs: accum_out

The accumulator is a sequential circuit, with an asynchronous reset. The circuit is enabled when input **en** is asserted. Upon every clock cycle, it accumulates the coins inserted into N, D, and Q, interpreted as multiples of five (specified in the previous section), and the accumulated total is output to **accum_out**. If the circuit is not enabled, the internal accumulator is set back to zero. The unit must be kept as **en = '1'** when accumulating to ensure that the data is not lost.

The unit should be able to represent a product that costs up to \$2.55, considering the representations in 5s that is internally maintained by the VMC. Determine your n-bit datawidth considering this assumption.

B) SRAM

We will expand our knowledge by learning how to create and interface IP (Intellectual Property) cores available in Quartus' IP Catalog into our designs. Specifically, we will use a 1-PORT RAM component for which we will store and read our product prices (required of the program and display mode). To generate a RAM IP core, carefully follow the steps provided in the Appendix. *Thoroughly test your RAM core through simulations to ensure you understand how to write and read data, and integrate the component within your design.*

C) Display Mode

Design choice: This mode may be implemented within the top-level vending machine, or as a separate component integrated into the VMC. When the user specifies display mode, the product specified is read from the on-chip memory. Accordingly, the product number is applied to the address input, and the read enable signal is asserted.

Once the data is read, the done signal must be asserted in sync with the read data output from the VMC system, displayed on the total0..3 output segment display. Hint: ensure you thoroughly test the memory unit before proceeding to code the display mode functionality.

D) Vending unit (vend_unit) –implements vending mode

Inputs: clock, reset, enable, price_in, NDQ

Outputs: change, insert_out, done

States: idle, calculating, convert

Other components: convert2bcd (accumulator optional)

The vend_unit implements vending mode functionality. The unit is enabled whenever enable = '1'. The price_in input provides the selected product's price which is read from memory, and therefore the FSM in the top-level (vending_machine) must ensure that all signals are synced when vend mode is activated, such that enable and price_in are aligned with respect to time. i.e. when vending_unit is enabled, price_in is reflective of the product's price.

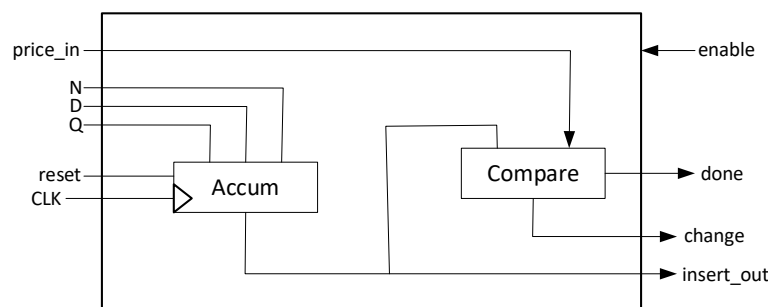


Fig. 1: Vending Unit Block Diagram (5x and bcd conversion not shown) [2]

As seen in the Fig. 1, the vend_unit consists of an accumulator (accum) and comparator (compare) circuit. If enable is asserted, or if coins still require processing once the vending machine pipeline is enabled, **the accumulator remains enabled until all coins have been processed**. Compare continuously compares the accumulator's output (total coins inserted) to the product's price on every cycle. When the accumulator is greater or equal to the product's price, then the change is dispensed until the user stops inputting coins to the VMC. This ensures that machine does not indirectly take money from the user as they insert coins

to purchase the product. Thus the vending unit must wait until $NDQ = 0$ to signal **done** and provide the user with the correct change, such that **done** is asserted concurrently to the correct **change** output.

As coins are inserted to the system, the **insert_out** output reflects the running total of the accumulator, displayed to the user, and **change** output reflects any change due. In the case of **vend_unit**, the **change** and running total (**insert_out**) output are represented as three 4bit Binary Coded Decimals (bcd), output on one 12bit port, representing the dollar value equivalents. Therefore, the outputs will need to be converted considering $5 \times \text{value}$ conversion, and thereafter bcd equivalent so that the values may be output and connected to the top-level's seven segment display. That is, the internal signals for change and running total will be multiplied individually by five and input to a component called **convert2bcd** which converts the **std_logic_input** binary value to its bcd equivalent. More details are provided below.

When either 1) soft/hard reset is asserted, or 2) the enable input is unasserted AND no coins have been further inserted to the system, all outputs from this unit display zero values.

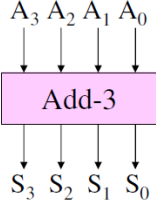
convert2bcd

The convert2bcd unit converts an 8bit binary number to its decimal equivalent, represented as three 4bit Binary Coded Decimals (BCD). Specifically, the unit convert2bcd converts a base 2 value to its base 10 equivalent, represented as 3 digits for input to a 7-segment display. The input to the convert2bcd will therefore be an 8bit value, for which the algorithm presented below is applied to the input, generating three 4bit values representing the hundreds, tens, and ones position for a decimal equivalent. These three bcd values will be output from the unit as one 12bit output and multiplexed within the top-level vending_machine for output display of the running total and change due.

Example: if the value $0xF6$ (11110110)₂ is input to the convert2bcd unit, the output will generate 246_{10} in response, represented as a 12b output ($0010\ 0100\ 0110$)₂. The ENTITY for the convert2bcd is as follows:

```
ENTITY convert2bcd IS
  PORT (binary : IN STD_LOGIC_VECTOR(7 DOWNTO 0) ;
        bcd : OUT STD_LOGIC_VECTOR(11 DOWNTO 0)) ;
END convert2bcd;
```

The convert2bcd comprises of various add3 components instantiated and connected to the inputs and outputs as illustrated below in Fig. 3. The add3 component implements the behaviour shown in Fig. 2.



A ₃	A ₂	A ₁	A ₀	S ₃	S ₂	S ₁	S ₀
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	x	x	x	x
1	0	1	1	x	x	x	x
1	1	0	0	x	x	x	x
1	1	0	1	x	x	x	x
1	1	1	0	x	x	x	x
1	1	1	1	x	x	x	x

Fig. 2: add3 Component Functionality [1]

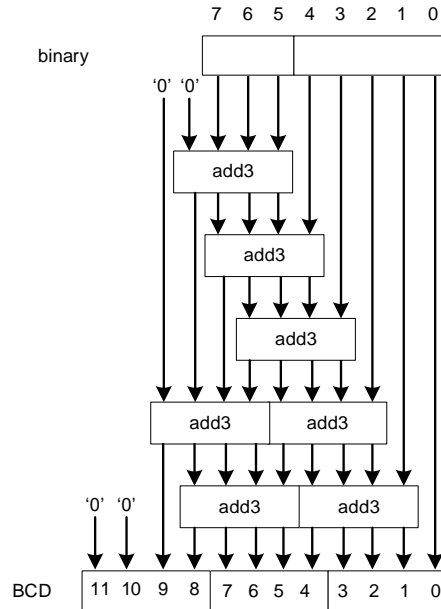


Fig. 3: convert2bcd Structural Diagram

Further elaboration on the methodology and background regarding this conversion algorithm for the interested reader is provided as supplemental reading material [1].

E) Free Mode - *implements free mode*

Likely the simplest to implement, and a good starting point. When the free state is entered, the mode simply sets all vending_machine outputs to the **floating state 'Z'** and dispenses the product specified by the user, represented by the assertion of the “finished” signal. Once asserted, the vending machine re-enters the idle state, where the finished signal is unasserted.

F) Vending Machine Controller Finite State Machine (VMC FSM) – *top level FSM, initiating and completing the VMC modes of operation*

Inputs: clock, reset, hard_reset, Start, set N, D, Q, funct, prod

Outputs: change0, change1, change2, runTotal0, runTotal1, runTotal2, total0, total1, total2, finished

States: idle, hardReset, program, display, vend, free

Components: sram, program_unit, vend_unit, convert2bcd

Your top-level entity, vending_machine, will contain all components specified above, which are instantiated and interconnected based on your flexible design choices and the specifications outlined above. The FSM implemented in the top-level VMC will exhibit Moore-like behaviour, with multiple possible outputs per state. The outputs for a given state will generate the appropriate signals to initiate any required data necessary to enable the corresponding unit. To recap the introduction, considering that the VMC is in an idle state, the condition to enter a functional state depends on the **start** signal and the 2bit **funct** input specified:

funct: 00: program, 01: display, 10: vend, 11: free

Note that the idle state and hardReset modes do not explicitly require a function code, however they are states within the FSM; the hard reset signal is an input to the system which triggers the hardReset state, and the idle state is implied if no other functionality or state is selected.

User/Operator Vending Machine operation selection and procedure:

- 1) Assert **start** and simultaneously specify the desired function at input **funct**. The given state will activate.
- 2) Next, specify the product with input **prod** while the vending machine initializes all necessary control signals required of the operation selected in (1), passing the product info to any necessary components.
- 3) Abide by the specifications provided in the document according to the mode selected. Recall:

Vend mode: Reads from memory to obtain the product's price. Insert coins to purchase product. The price is displayed to output, and the coins inserted are displayed to the running total output (likely delayed by a clock cycle when viewing the waveforms).

Program mode: Set is asserted once the coins entered by the operator is reflective of the products price; this is an accumulation of N, D, and Q coins inserted into the VMC. For the witty reader, the "coins inserted into the system" to program the machine will be returned to the user, however this functionality does not need to be implemented for the project. Consider the coins inserted as trick coins that the operator uses only to program the machine.

3) Suggestions & Comments:

To accommodate the required FSM behaviour specified by the vending machine, it is suggested that two separate processes are used in your top-level to a) enter a state, and b) initiate and wait for state completion. That is, the first process, implemented sequentially, should focus on entering a state considering the Start and Funct signals. The second process, also a sequential circuit, considers the state and asserts/deasserts the appropriate signals to enable their respective unit. For instance, when the vend mode is selected, a read request must be sent to the memory unit to obtain the product price and send the value to the vend_unit. The **price and time that the vend_unit is enabled must align**.

VMC Outputs:

As the outputs to the three sets of seven segment displays may come from various sources, these signals must be multiplexed in the top-level. For instance, both type of resets and the idle state must ensure that the output ports **change**, **total** and running total (**runTotal**) are set to zero, whereas the display and program modes will display values for total and runTotal, and the vending state will require all of the outputs to be displayed with their appropriate values. The display mode and program unit have not accounted for the multiple of 5 and bcd conversion required of its signals prior to being output. Therefore, the totals generated by these two states must first be converted to their actual values and passed to convert2bcd within the top-level circuit.

A word on resets:

A **soft reset** is simply used to allow the user to exit a given state. All products and prices remain intact.

A **hard reset** is considered a factory reset: all prices associated with the products are wiped from the system, and the system is placed in an idle state. Since the on-chip memory does not have an explicit

clear/reset port, the VMC *will have to clear out the memory manually during the hardReset state*. Accordingly, consider that this state may need to call yet another FSM to clear out the memory. The method you decide to invoke is up to you, however ensure you justify your reasoning for the hardware design.

Functional Verification

Use **testbenches and Modelsim to verify the functionality of all components** designed and your top-level vending machine. You may find it useful to use *wait until "signal" = '###'* to proceed to a new mode in your top-level testbench. Research other VHDL statements that may assist you in creating an efficient testbench, including other types of wait statements. Note that certain keywords, such as wait statements, are only used in testbenches as they are **not synthesizable logic** (i.e. can not be made into functioning circuit logic).

More Suggestions:

DO NOT design and build the vending machine as one large circuit. Implement one component at a time, as specified in this document, while thoroughly testing each unit to ensure proper functionality. Furthermore, ensure that you understand the protocol inherit of each of the components you have created. You will need to use this timing behaviour to integrate all the components and interface them appropriately within your design. Integrate one component at a time. Once the unit works, proceed to integrate the next into the top-level VMC.

4) Presentation/Deliverables/partners

You may work with a partner. Please sign up with your partner in Canvas > People > Project I. Give your group a name. If you do not work with a partner, this will be considered during grading.

An audio recorded powerpoint presentation must accompany your project, covering the following:

- A slide on each component/function created, discussing the design, VHDL, testing procedures and timing diagram. Provide a block diagram of your component's design, and a state diagram if applicable.
- A slide on the overall vending_machine design and how components are interconnected according to your hardware implementation. Also provide a state table and state diagram for your VMC, illustrating the control signals output from your Moore machine, and the transition conditions.
- Dedicate one slide which discusses how you've thoroughly verified the correct functionality of your design

Ensure you submit your presentation and Quartus project in one zip, uploaded to Canvas by the specified due date on the semester schedule.

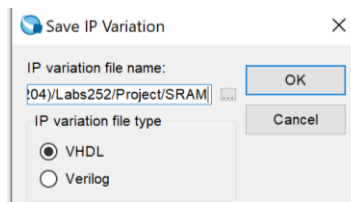
If you work in partners: Include an **individual report**, clearly outline and describing each of the contributions you have made to this project. Marks will be assigned accordingly to workload distribution.

References:

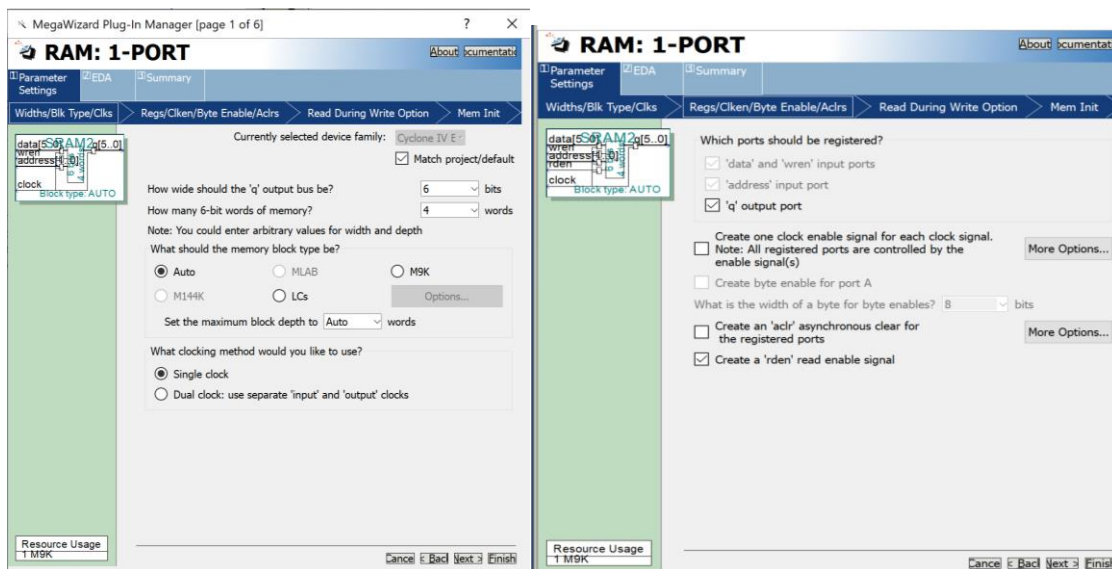
- [1] Gao, Shuli; Al-Khalili, D.; Chabini, N. "An Improved BCD Adder Using 6-LUT FPGAs", 2012
 [2] Lin, Calvin J, "A Coin Operated Vending Machine Controller," MEng Thesis, M.I.T., Boston, MA, 1999.

Appendix: Creating a 1-port RAM IP with Quartus' IP Catalog

In your Quartus project, select **Tools > IP Catalog**. A window will expand on the right. Under **Library** select **> Basic Functions > On-Chip memory > RAM: 1-Port**. Double click RAM: 1-Port. A window will pop up entitled *Save IP variation* as shown below. Ensure file type is specified as VHDL. Ensure the directory is pointing to your project folder, and at the end of the address, write SRAM as seen in the figure below. This will create a vhd file for you called SRAM, which you will integrate into your project once the steps below have been completed. **Click OK**.



Wait a moment. A new window will eventually appear in your task bar. Expand the window which will resemble the figure below. Specify **6 bits** for the width of the 'q' output, and **4 words** under "How many 6-bit words of memory?". Leave the other setting as defaulted. This will create a memory consisting of 4 memory cells, each of 6bit datawidths. Click **Next** at the bottom of the window.



On the next tab/page entitled **"Regs/Clock/Byte Enable/Aclrs"**, select **"Create a "rden" read enable signal"**. Click Next five more times, ensuring the default settings look the like provided figures on the next page.

On the last window, as shown below, click **Finish**. In your Files, you should now find a **.qip file**. If it does not automatically link to your project, Go to Project > Add/Remove Files in Project, and search for sram.qip. Add this to your project.

Once you return to your Quartus project, in the left panel project navigator, expand the sram.qip and double-click sram.vhd. Copy the entity included in this file to instantiate your sram into your vending machine design. Note that you may not actually view the internals of the ARCHITECTURE as this remains proprietary information of Intel. Hence the term IP core.

Ensure you thoroughly test this memory component to view the timing characteristics associated with reads and writes. You will need this information to properly interface the display and program mode respectively.

