2007

[FINAL REPORT]

A detailed report of the Beer Drinking Bearded Dragon's Five-Stage, Pipelined Processor

Table of Contents

Developer Roles

John Tooker	(Team Leader)	Datapath, Control Unit, ALU
Chet Henry		Assembler
Kiranbir Sodhia		I/O
Kyle Brown		Data Cache
Will Weston	(joined the group late)	Register Bank, Instruction Cache

Introduction

The Bear Drinking Bearded Dragons created a five-stage, fully pipelined processor. A data and instruction cache was implemented as well as branch and data hazard detection.

Data Path

The Data Path is the 'standard' five stage pipelined processor as outlined in FIGURE DATAPATH_1-5 (each is a stage of our processor). Each component was tested separately to ensure that it works correctly. This helped to debug the system as a whole as everything as put together.

The datapath makes use of 32-bit long instructions and data travels 8-bits wide. The 32-bit instruction length gave freedom to create whatever instructions we needed as well as represent all (and complete) values and addressed in the immediate section. The 8-bit wide data section sped up the calculations and allowed us to reduce the amount of hardware (expense) of the system, this was a necessity, as we started out with 32-bit wide data, but did not have enough room on the board for that register bank. 8-bits allows the numbers from -128 to 127 to be represented in our processor, we only need the number from ±10 for our multiplication algorithm, and up to around 60 for the memory address for the program counter.

Our processor can run at **11 MHz**, since we did not need to write data to memory in our algorithm. If we would have to write to memory, (and have a cache miss) the slowest our clock could go would be 6.25 MHz. We ran all of our tests at 2.56 MHz to be safe and due to time constraints.

The slowest stage of our five-stage processor (excluding the memory write-back miss) is the third stage. The total delay is 90.1ns (given by Quartus). This comes from a 37ns delay in the control unit (mostly due to calculating and storing information for the data-hazards), as well as getting information through the ALU – 60ns separately. 90.1ns → 11MHz.

We have created an input block that takes care of the formatting issue the Altera board has (the issue of active-low). This gives us pins we can use, as well as just stating what physical pins we need to connect our inputs too. See FIGURE INPUT_BLOCK.

MultiplyByTen is a block that takes a quicker clock and slows it down. We need this kind of component for two reasons: 1) our processor needs to run slower then the 25MHz clock that is provided on the board, and 2) for testing purposes. The memory block has built-in registers on the input and outputs, but we need values to pass between these in one 'regular' clock cycle. Therefore, the memory runs on a faster clock then the rest of the processor. See FIGURE MULTBYTEN.

ALU

The Arithmetic Logic Unit (ALU) is a 2's complement, integer based machine. Along with the basic adding, subtracting, logic functions it can also shift left and right logically as well as multiplication. All operations (except shifts) take two inputs, operate on them and give the output as an 8-bit number. Each input is 8-bits wide, except for the multiplication; we only take the lower four bits as the input to the multiplier. We do this for two reasons:

1) Multiplication of two 4-bit numbers can produce an 8-bit number

2) The numbers we need to multiply are in the range of -1 to 6 (or 11111111 to 00000110) both numbers are in their 2's complement form when we cut off the four most significant bits: 1111 to 0110.

The multiplication unit is the slowest part of our ALU; we justify this because our processor is a matrix multiplier. We are using the multiplication unit on our FPGA to handle our multiplication rather then using the 'normal' logic elements; that had a greater significance when we originally had our data be

32-bits wide. Multiplication of two n-bit number produces 2n-bit number. With the values we had to produce for our matrix multiplication, we found we could represent those with 4-bit number. Thus to speed up the ALU even more, we took the lower 4 bits from the data, multiplied those numbers and to get an 8-bit number as a result (as apposed to multiplying two 8-bit numbers and taking the lower 8 bits from the 16 bit result).

The ALU is directly controlled by the control unit. The control unit provides a signal that selects one of the six functions of the ALU (this is done with a multiplexor). The control unit will directly translate the opcode of an R-type instruction to the corresponding ALU operation, if the instruction is an I-type the control unit will automatically tell the ALU what it needs to do (if anything). The control unit also looks at the opcode and tells the ALU whether to add or subtract (they both use the same hardware).

The error detection/correction is fairly simple for our ALU. It was originally developed with an overflow detector for the addition and shift left logical, but this was later taken out since our processor does not handle interrupts. If the ALU select line is not one of the 6 operations the ALU will output "00000000."

The longest time it will take a signal to get through the ALU is 59.3ns.

Table ALU.1: ALU control mapping:

- 00000 - add
- 00001 - sub
- 00010 - mult
- 00100 - and
- 00101 - or
- 00110 - xor
- 01000 - sll
- 01001 - srl
- 11111 - addi (same as add)
- else: ALU will output "000...000"

Please see FIGURE ALU_TEST_32 for a demonstration of how our ALU worked with 32-bit wide data and overflow. See FIGURE ALU_TEST_8 for a demonstration of how our ALU works now.

Control Unit(s)

There are five control units controlling our processor: one control unit for each stage. There is only a little bit of redundancy in doing this. The control unit keeps track of data hazards, and this circuitry is partially repeated in each control unit. Separating the control units also speeds up the process of each stage of the pipeline. Our longest propagation delay is around 70ns and occurs in the third stage of our pipeline. This length is caused by both keeping track of data hazards: making sure the ALU has the right source for its information and the multiplication unit: which has been optimized and will be used for about 25% of our executed instructions.

See FIGURES CONTROL_1-5 for the waveform tests.

Control Unit 1 takes care of the difficult task of branch hazards; in fact, it directly tells the program counter its next value. The program counter will be the previous value plus one, unless the current instruction is a jump or the previous instruction was a branch that was taken. In the case of a jump, the PC will just get the next value from the opcode right away. This takes a little time, but is quicker then the third stage, so there is not problem. The delay through this control unit is 42ns.

Branches are handled in both the first and second stage. If there is a branch, it will be detected in the second stage. This information is sent to the control unit in the first stage, which will then decide whether or not to flush the pipeline or not. This is a simple process since the branch instruction will be allowed to continue, but no the current instruction. If a branch is taken, then the current instruction will be replaced with a 'null instruction' whose opcode is 32 zeros (indicating a 0 + 0 stored to the zero register, which cannot be written to). If a branch is not taken, we continue to execute like normal.

Control Unit 2 also keeps track of where the two numbers (that are being compared) come from. Normally they come from the rs and rt registers, but if the rs and rt registers have not been written to yet (data hazard) the control unit will pull these values from the ALU or the data memory output. The delay through this control unit is 42ns.

Each control unit uses registers to store and calculate data-hazards. For every instruction, the corresponding control unit will store the current destination register (if there is not one, zero gets stored). Also on each clock cycle that stored data moves to another register, and then to a third one (if necessary). Every time data is needed to be read from a register, the control unit checks each of these storage registers. If there is a match, data needs to be forwarded, and there is MUX that controls the source of the data in front of each component.

Control Unit 3 manages the data hazards for the third stage as well as telling the ALU exactly what to do. It also gives the ALU the immediate value if that is needed. (See the ALU section for more information on this process). The delay through this control unit is 37ns.

Control Unit 4 controls the data memory. If you want to store a word you just removed from memory, it will do that; otherwise it just stores the rt value (if it is a store word). The address for the memory is calculated in the third stage with rs as the offset. The delay through this control unit is 30ns.

Control 5 decides when to store data. The password is also pulled from here. This stage decides if you want to store the ALU result or the data memory output. It has a write enable line that feeds the register bank in the second stage, as well as if you want to store into the rd or rt register. The delay through this control unit is 27ns.

Register Bank

The Register file had thirty-two, thirty-two bit registers. It has three, five bit data access inputs and one five bit data read input. There is also a write enable input and a clock input. It had three, thirty-two bit outputs. One output is intended specifically for display the other two outputs are intended for operations.

The main parts of the register file are the thirty-two, thirty-two bit flip-flops, the decoder for data in and the three multiplexors for data out. The write enable enables data to be written to the flip-flops and the clock prevents data decay.

The register file has been tested for data decay, writing data and reading from all three data outputs. We have since changed the 32-bit flip-flops to 8 bit flip flops. See FIGURE REGISTER_32

Memory

The data cache is a 16-block, write-through cache. It has an 85% hit rate after the cache is filled. A 16-block design was chosen as the matrix multiplier utilizes 18 data points. FIGURE DATA CACHE

shows the cache misses. The address values are random to simulate a worst-case scenario. When fully loaded, the cache will only miss four times. However, if loaded correctly, it will only miss twice. Considering the same address is loaded more than once, an even worse case scenario occurs. A write-through design was chosen as it was simple to implement, and a delay-on-write was acceptable for the designers.

The instruction cache uses a direct mapped design for its two blocks. It uses 8-bit addresses and 32-bit data. There are a few reasons for having only two blocks. First, 16 addresses may be cached at any given time out of 256 total addresses. This means 1/16 of our instructions can be cached at any given time. This is a high fraction of cache over ROM. The problem is this creates a high amount of misses. However, the cache miss penalty for this design is about zero from the fast clock of our memory.

The block diagram of the instruction cache can be viewed in FIGURE INSTRUCTION CACHE 1. Two clocks exist in the design; a fast clock leading to the memory, and the slow clock leading to the cache logic. The address input leads to the memory where it outputs the selected data in about 50 ns. It also has a lead to the cache logic, where the cache determines if the requested data is stored. If the data is stored, the cache logic unit outputs the data, if not it writes the data for the selected address from the ROM. Finally, a multiplexer outputs the data either from the cache if it is cached, otherwise from the ROM.

As seen in the vector waveform in FIGURE INSTRUCTION CACHE 2, the data will be outputted from the cache if that data is in the cache. If not, the data will be written to the cache, and outputted from the memory. The instruction cache runs on a 400 ns clock and the memory runs on a 40 ns clock. It requires about 50 ns after the address is received to output the selected data from either the cache or the memory. It requires 160 ns for the cache to realize data is not stored, then to write the data, and finally to output the data.

VGA and I/O

As this project began, it was our intentions to attempt designing components that exceed the minimum specifications required. To follow this idea, we decided that it would be interesting to implement VGA output. Progressing through the semester, a simple controller was written that is able to send correct sync signals, as well as effectively display a superpixel, or a 10 pixel x 8 pixel area on the monitor.

While developing the VGA controller, one of the difficulties that we encountered was defining a generic font set. While creating the memory file to define a character was not difficult, the problem arose when referencing the correct range of memory when selecting a character to display. While this issue was examined, we decided to implement the concept of using a seven-segment display, and just using two push buttons to scroll through data.

In implementing the scrolling scheme with push buttons, the only changes from the VGA controller, was the decoding of the final output signal, and adding a unit to scroll through the registers.

The basic design consisted of nine 8-bit registers to store the nine elements of the result of the matrix multiplication. First two decoders were needed to direct the output to the correct register. They select the correct registers by decoding an 8-bit address. The first decoder sends the outputs to the register, while the second enables the specific register. Furthermore, the second decoder is enabled by a control signal for register writes.

Data from the registers are directed to a multiplexer. The multiplexer selects the output of the register that is selected by the user. Tying the select lines of the multiplexer to a counter does this.

The counter is a modification of a standard up-down 4-bit counter. After the up button is pushed, the counter increments. Likewise, after the down button is pressed, the counter decrements. As mentioned earlier, the outputs of the counter selects which registers' contents to show.

A waveform of a simulation can be viewed in FIGURE I/O 1.

As the deadline for the project inspection was approaching, time limited the completion of the VGA display. However, after a little research, it was found that the addition of a keyboard-controlled input was still possible.

A keyboard works quite similar to an electric telegraph, sending data in the form of Morse code. The communication is performed over a PS/2 port that is serial and bidirectional. The keyboard has a clock controlled by itself as well as the system. The configuration of this clock line with a data line controls whether to send data from the system to the keyboard or from the keyboard to the system.

As this project only required data from the keyboard to the system, both the clock and data line were held initially high. After a key was hit, the keyboard would take control of its clock, and would transmit a start bit. Following this, eight more bits would be sent, as well as a parity and stop bit. The waveform of this transmission can be viewed in FIGURE I/O 1. The byte sent between the start and parity bit would represent one character in the form of a scan code. This scan code would finally be decoded into its respective ASCII code.

Similar to the two pushbuttons, the keyboard was tied to the counter. Thus, when the "+" button on the keyboard was pressed, it would increment to the next element of the matrix, and "-" would decrement to the previous element. Since the byte was stored in a shift register, this allowed for an auto-scroll feature.

Finally, the keyboard was also tied to the multiplexer that selects the output. This allowed for the number buttons on the keyboard to select which element of the matrix to view, that is, button one would show the first element, button two would show the second element, and so on. The final design and results can be viewed in FIGURE I/O 2 AND FIGURE I/O 3 respectively.

Assembler

Input Format

The Assembler 1.0 Beta is able to handle all instructions in TABLE INSTRUCTION_SET.

The la and li are currently not working but will be addressed in later versions.

The j, beq, bne allow for use of labels. There are currently no restrictions on label names except that a label must not conflict with instruction names. If a label ends in ":" the ":" is parsed from the label. The label is then hashed using md5 function. This allows for easy label replacement after all address has been calculated and very few label conflicts. (Ie. count, count1, count2 are perfectly fine)

In the current beta version no comments are supported. In later versions the comments will be allowed with "#". Anything after "#" but before the new line will be considered comments and will be omitted from the machine code.

The input format of a given command is MIPS style however the $ before registers are not required and registers names must be an integer between zero and 31.

Technical Discussion

The assembler is implemented in php on our project website. (http://cycling/430/?q=node/28) Php allows for easy parsing and great portability. The Complexity of the assembler is currently O(n). It first runs through the code converting every possible value to its output. However in some cases like when a

jump or branch is before its respected label, the assembler is forced to run back through the code to update the jump or branch with the correct address. This process is O(n) where n is the number of times a branch references a previous label.

The instruction set is hashed for quick access. When the assembler comes upon a label it is placed into the instruction hash and is available throughout the rest of the computation.

Please See the Assembler Example in the FIGURES Section.

Stumbling Blocks

These are notes we have made as the project has developed:

- How long our instruction will be (16 bits, less architecture & faster, more instructions: about twice that of 32 bit) - (32 bits, longer, but fewer instructions, worth the extra hardware - plus just easier to work with)
- Having enough memory - Had to do part of it with the logic gates
- Having enough logical elements: our register file (32 bits x 32 registers x 3 logic elements/register) = 3047/3744 logical elements used just for the register file. Without the VGA or Cache, we were using 4910/3744 Logical Elements
- Getting the 'clock' to work correctly. I defined my input pin as a clock (in addition to assigning it to the clock - pin91 - on the board). This caused Quartus to realize that there would be a bigger delay as to need a slower clock then the provided 25.175MHz. I had created a function that would do this function, but Quartus would not recognize that function as such, thus I eventually had to remove calling my clock pin a clock pin. AHHH! Even this has not fixed our problem. I've tried putting in buffers, but this does not work (so fan-out is not the issue at all). - Well, after trying to 'slow down the clock' at various speeds I have come to the conclusion that some will generate fewer 'non-operational paths' but always above 75%. Each path (total of 40) comes from the Memory - 32 from Instruction, 8 from data.

Meeting Log (See online)

Wed 09-26-2007 We met after class and talked about having a 16 bit long instruction and about designating a register for comparisons, and then we could know branches right away! (in the fetch stage)

Mon 10-01-2007 Talked about 32 bit instruction length, format and instruction set. See Here for more info. Confirmed completion of Milestone 2.

Mon 10-22-2007 Decided to put the branch control in the 2nd cycle. (We'll still put jump recognition in the first cycle) Chet works on assembler; Kiranbir works on VGA; John works on control units and Data path; Kyle works on cache; Will works on Register Bank

Tue 11-27-2007 Kiranbir, Chet and John met to discuss deadlines, and worked out a solution if the VGA part of our project never gets working. Also worked on a plan to complete our project and prepared the report due 11-28-2007.

Conclusion

In the end, we had a pipelined, cache enhanced, hazard detecting processor that multiplied two 3x3, integer matrices. If we had to do it again, we would not have done much different. We know it would take a lot of time, so we could have done more sooner, but then our other classes would have suffered. All in all, we are very satisfied with out finished product.

Instruction	Type	Operation	Opcode
add	R	$rd = $rs + $rt	000000
sub	R	$rd = $rs - $rt	000001
mult	R	$rd = $rs * $rt	000010
and	R	$rd = $rs & $rt	000100
or	R	$rd = $rs | $rt	000101
xor	R	$rd = $rs XOR $rt	000110
prnt	R	$rs $rt $ru	000111
sll	R	$rd = $rs <-- shamt	001000
srl	R	$rd = $rs --> shamt	001001
gtpw	R	$rd <= dip switch input	001111
addi	I	$rt($rd) = $rs + immediate	011111
lw	I	$rt($rd) = memory[$rs+immediate]	010000
sw	I	$rt => memory[$rs+immediate]	010001
beq	I	if($rs=$rt): PC=PC+immediate	100000
bne	I	if($rs!=$rt): PC=PC+immediate	100001
j	J	PC = address	111111
Pseudo Code	**What you type**	**What it really is**	**Opcode**
la	la $rt($rd) address	addi $rt $0 immediate	*011111*
li	li $rt($rd) immediate	addi $rt $0 immediate	*011111*

TABLE INSTRUCTION_SET

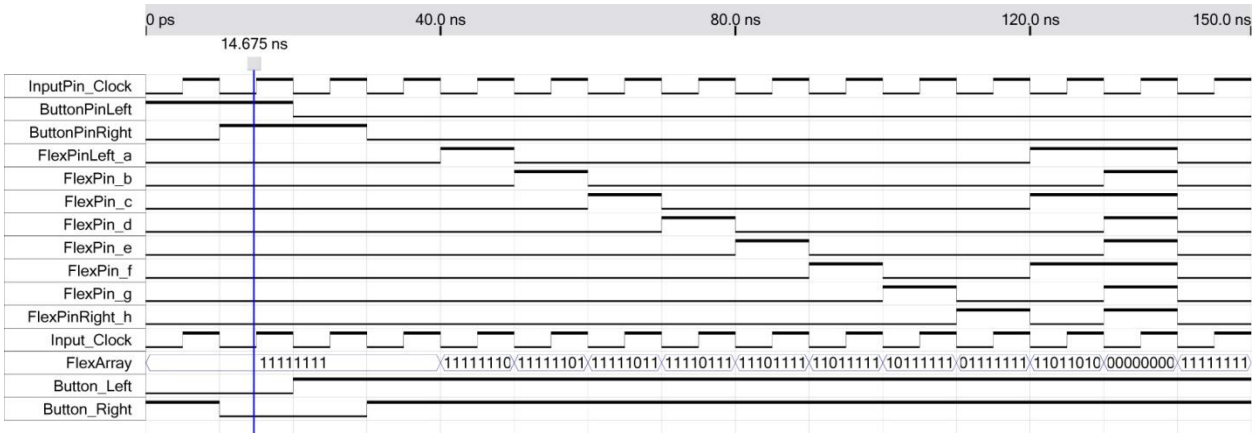

FIGURE INPUT_BLOCK

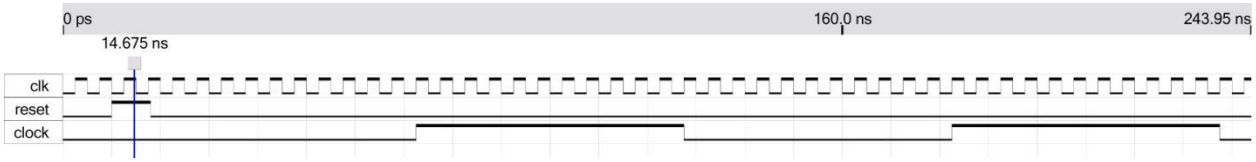

FIGURE MULTBYTEN

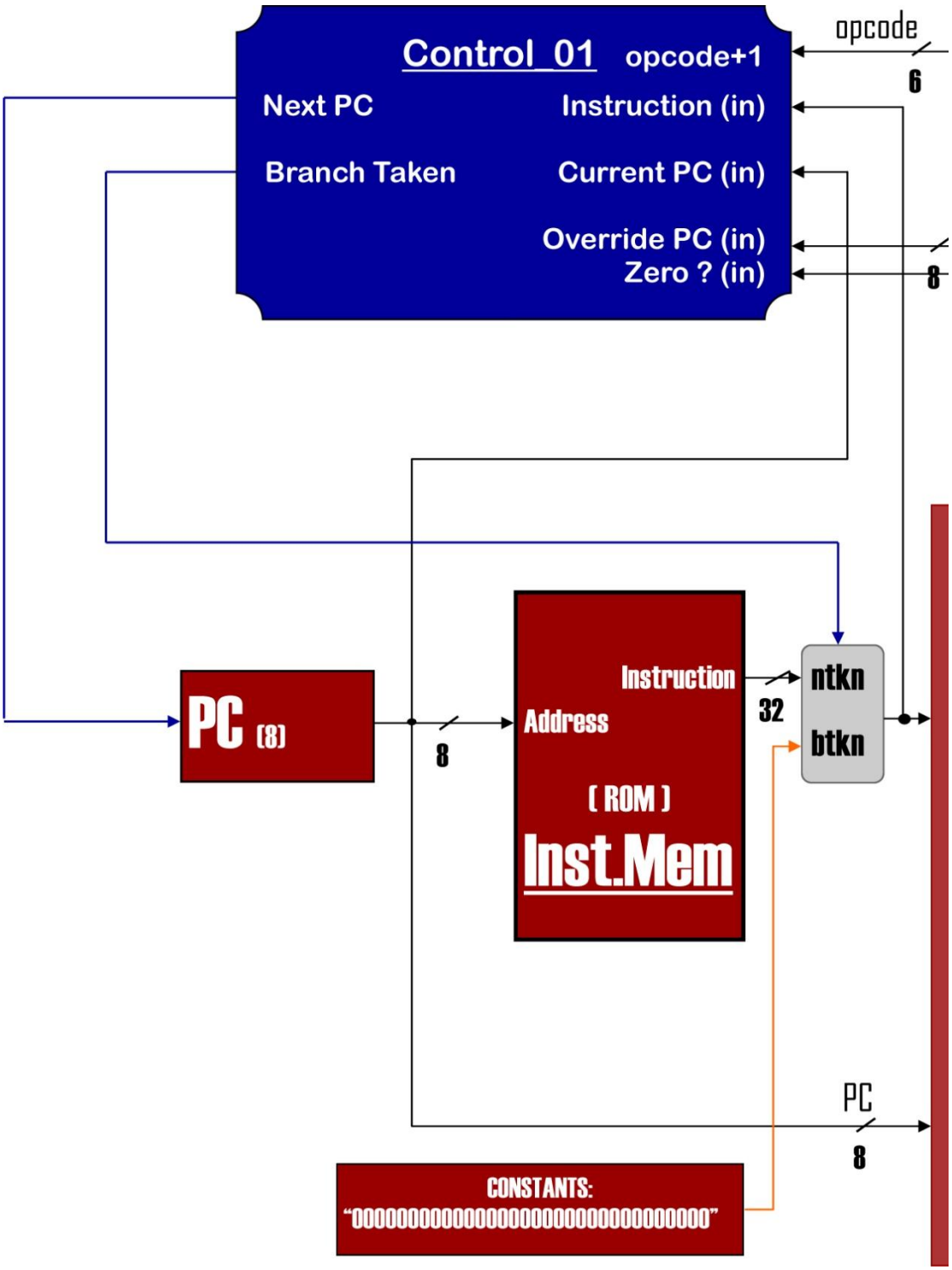

FIGURE DATAPATH_1

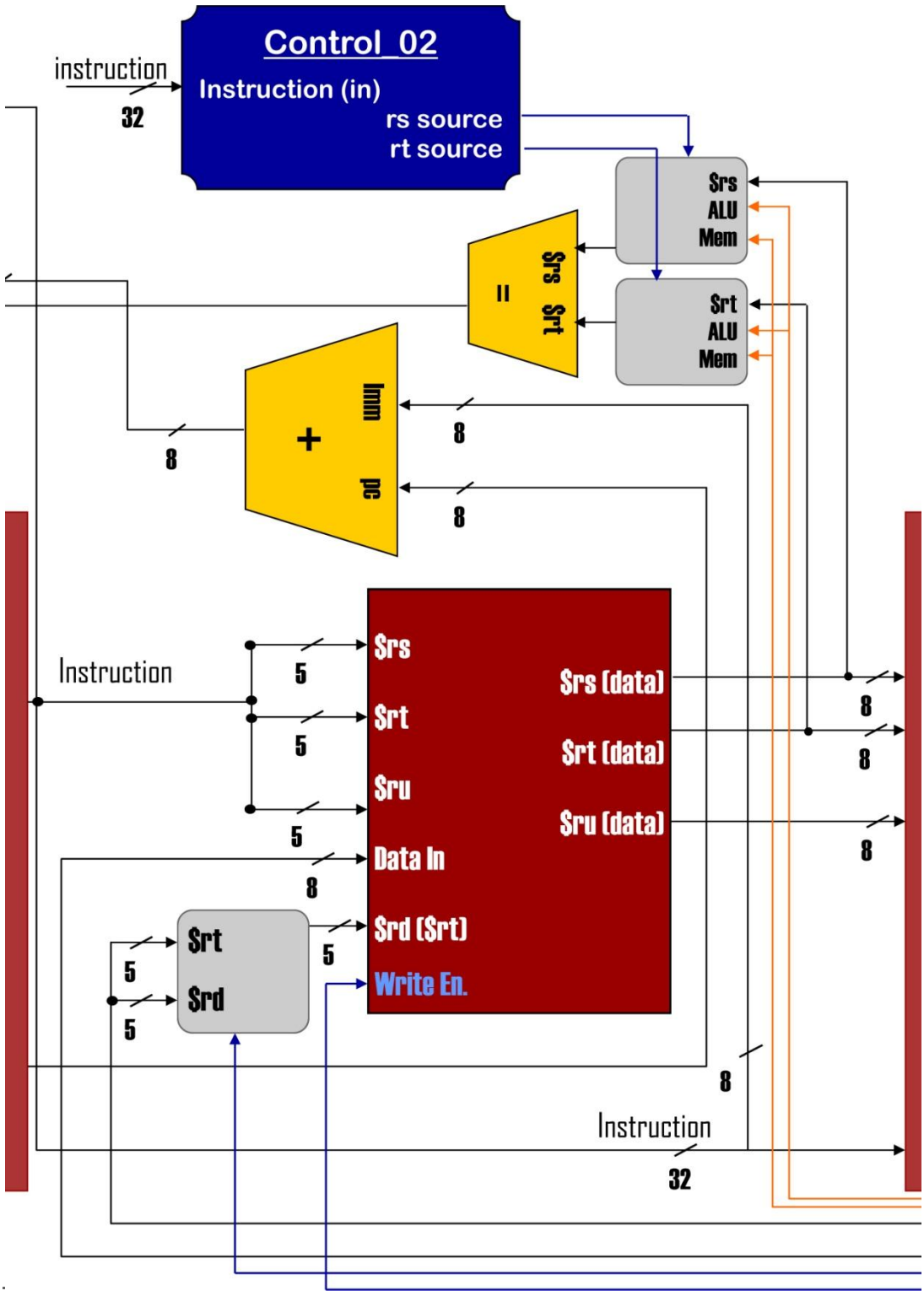

FIGURE DATAPATH_2

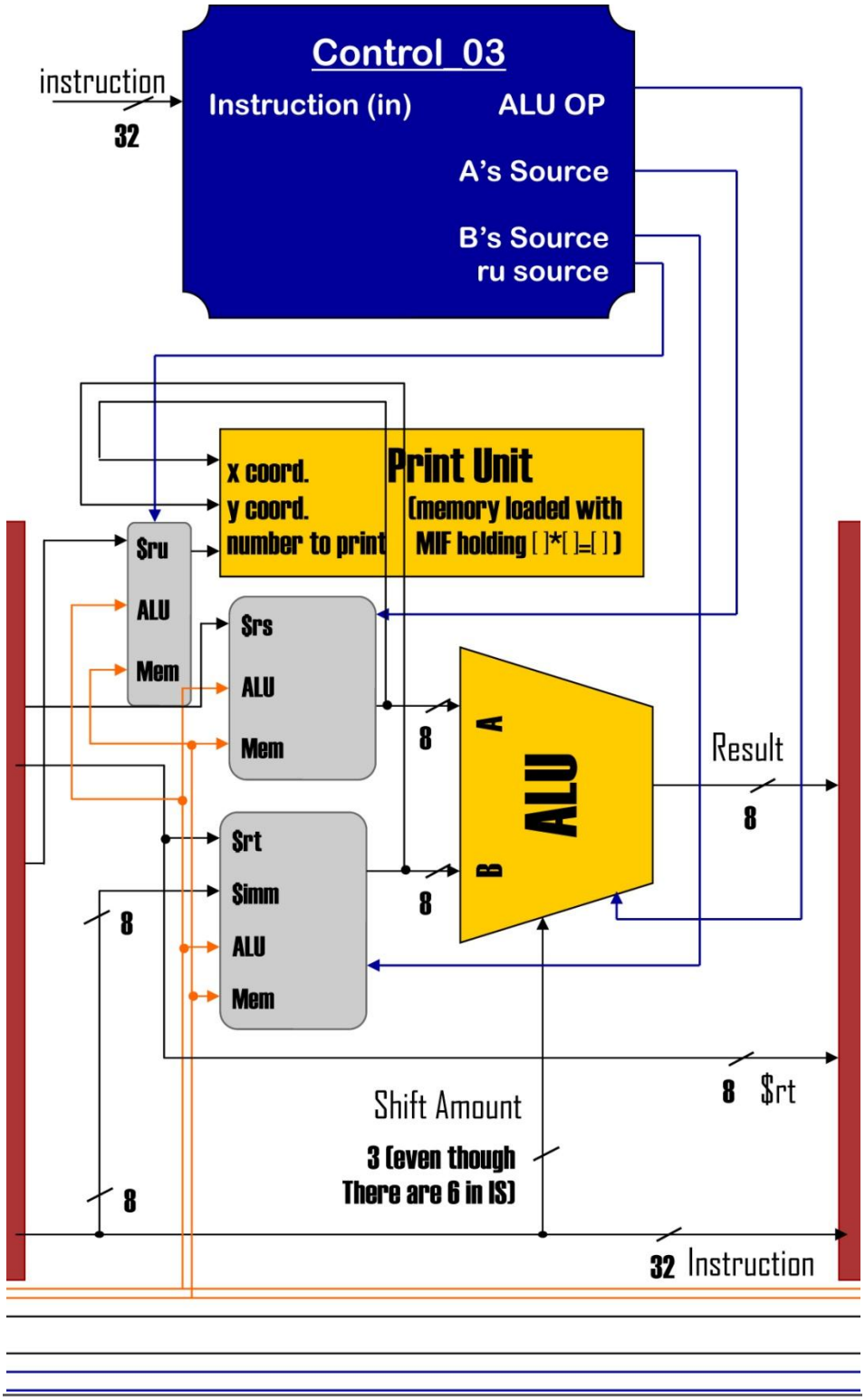

FIGURE DATAPATH_3

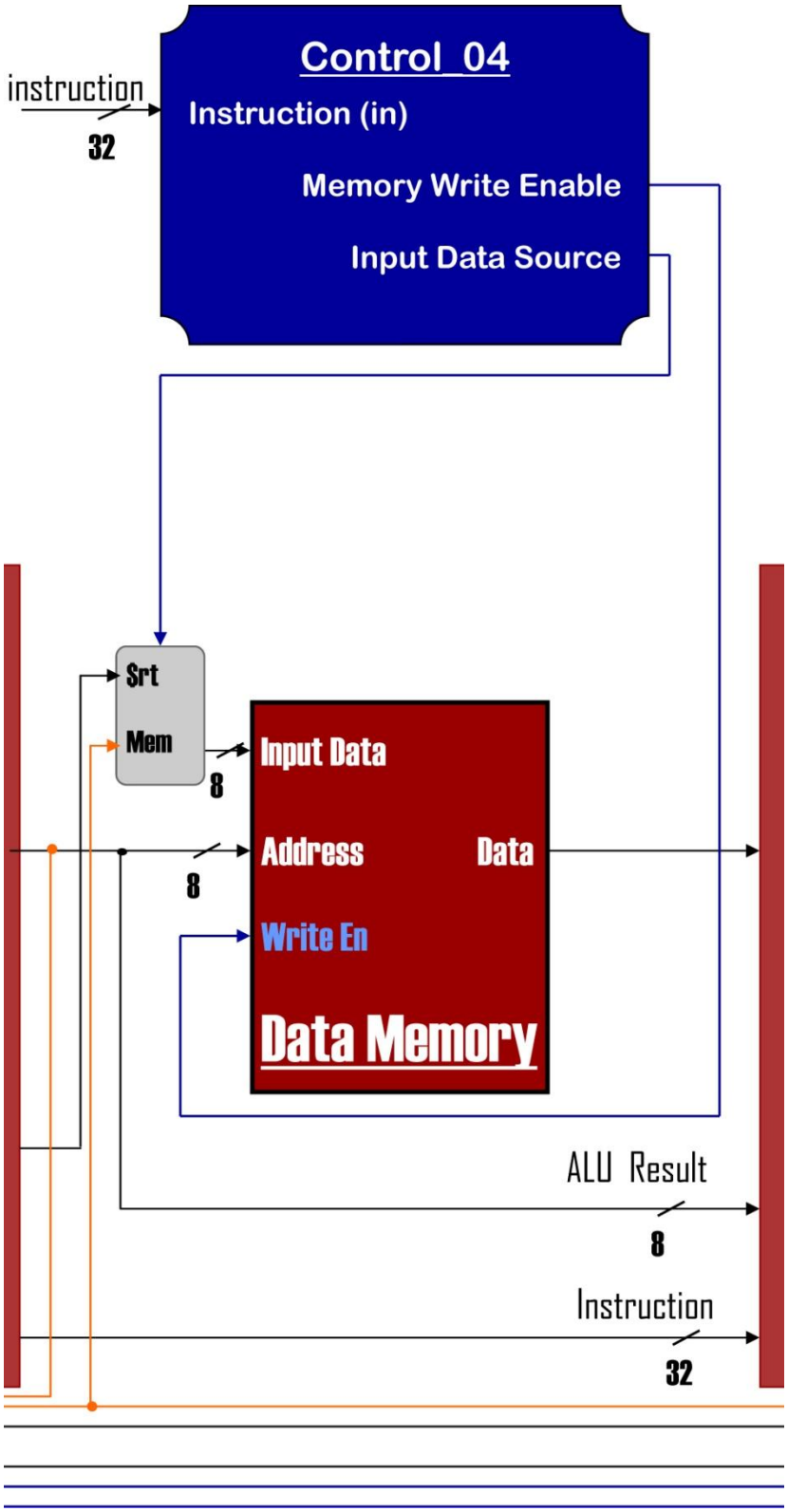

FIGURE DATAPATH_4

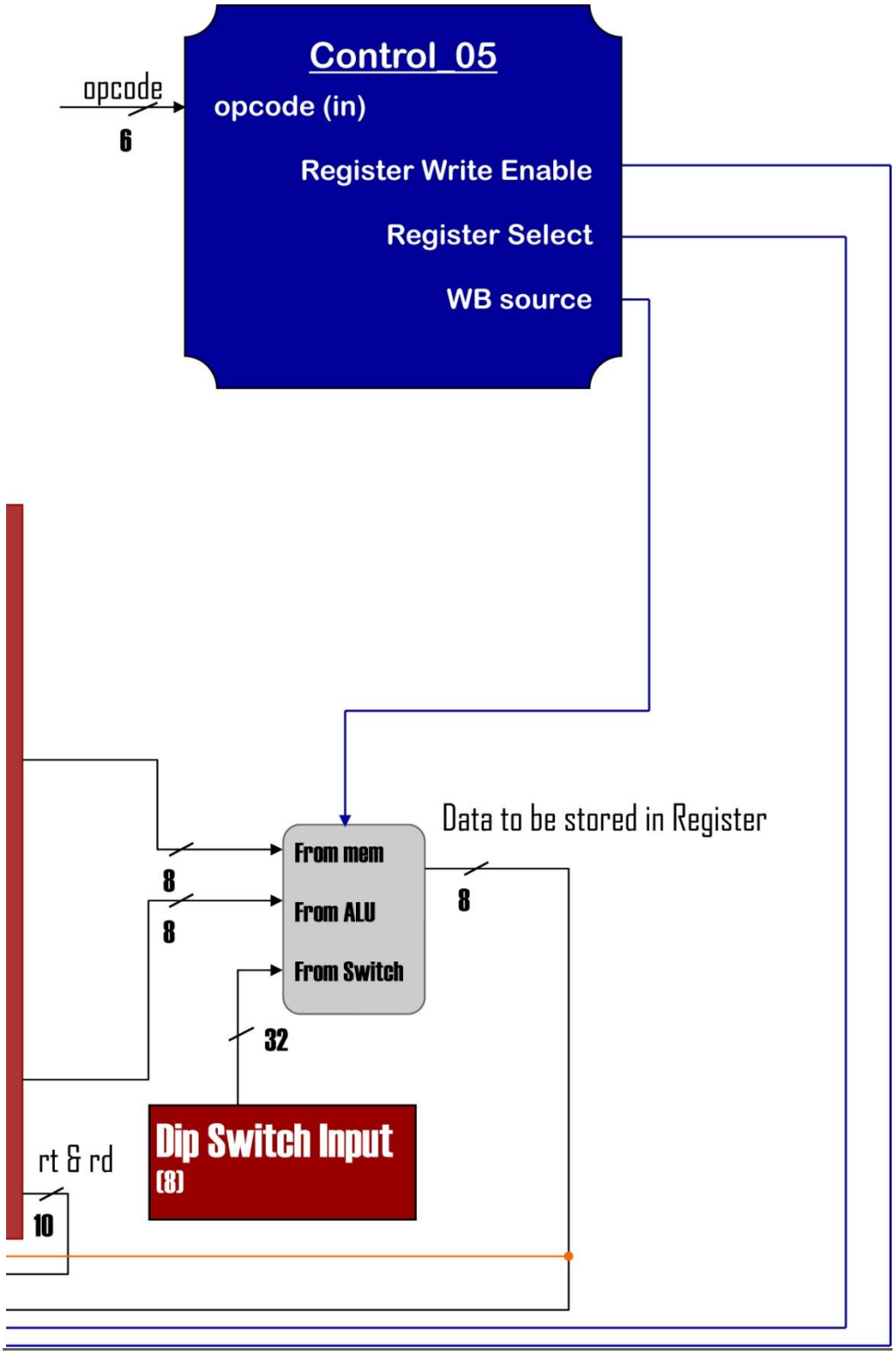

FIGURE DATAPATH_5

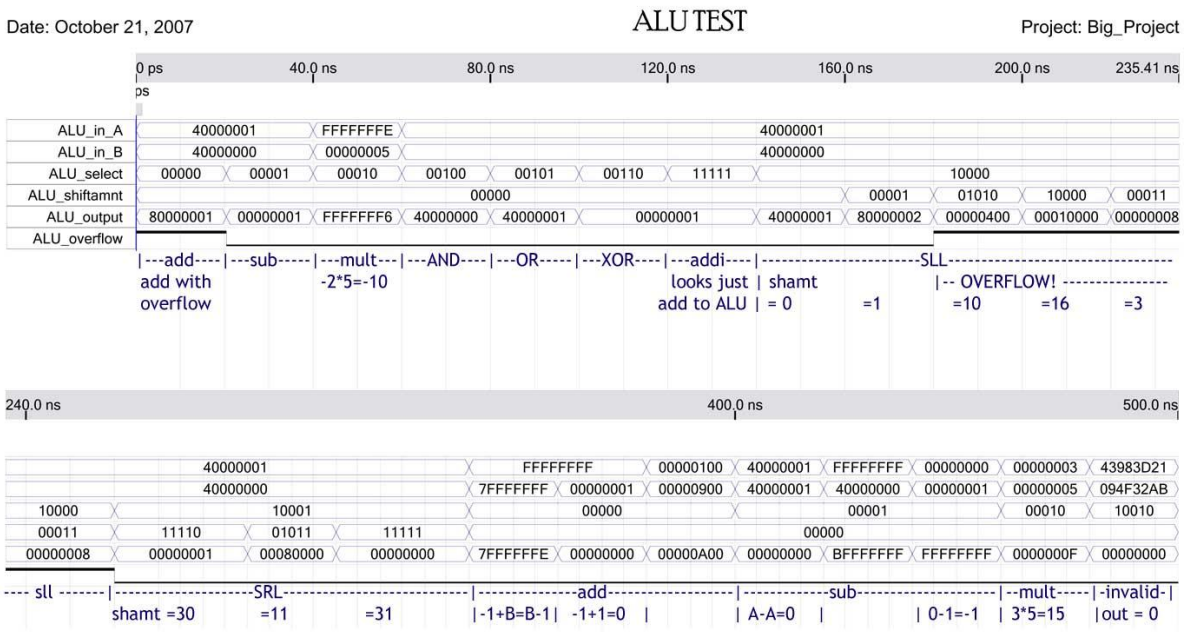

FIGURE ALU_TEST_32

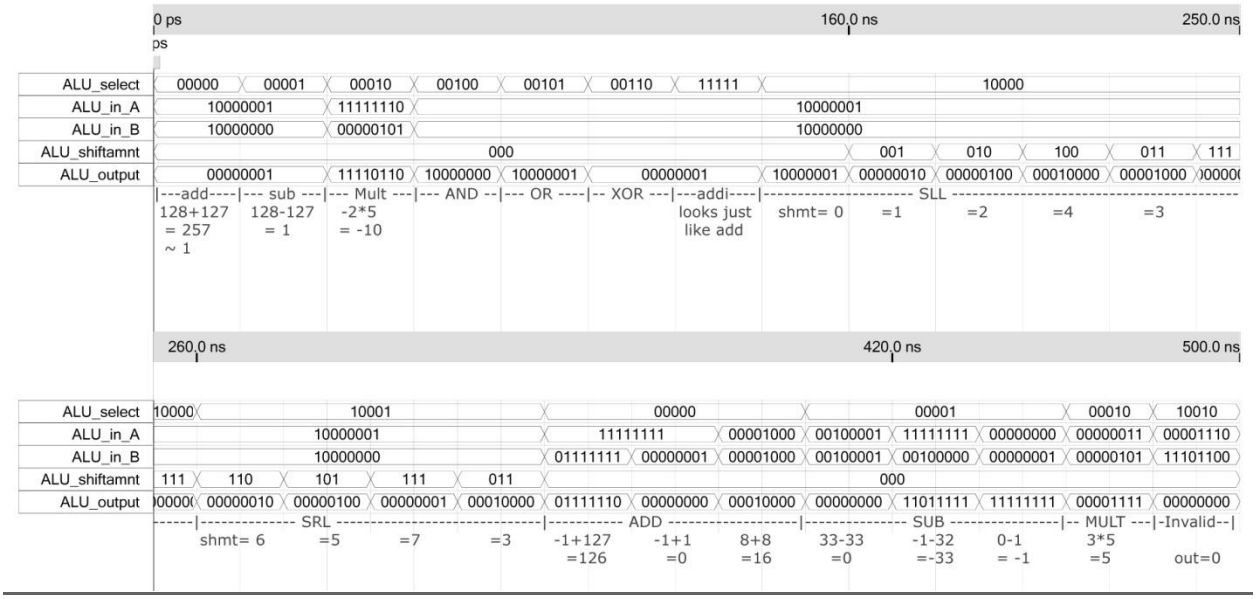

FIGURE ALU_TEST_8

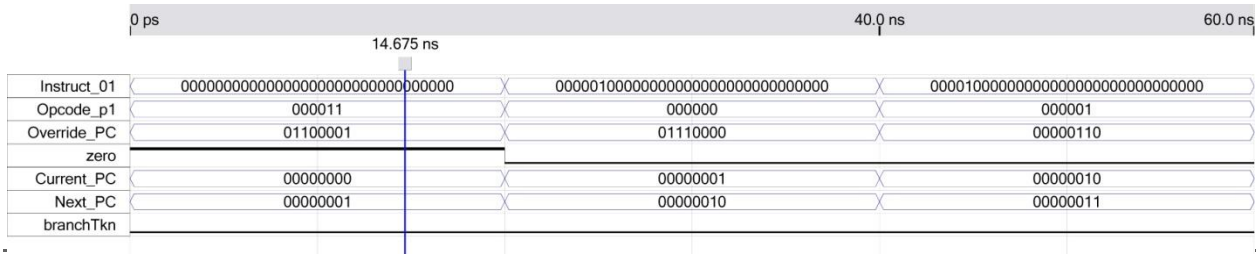

FIGURE CONTROL_1

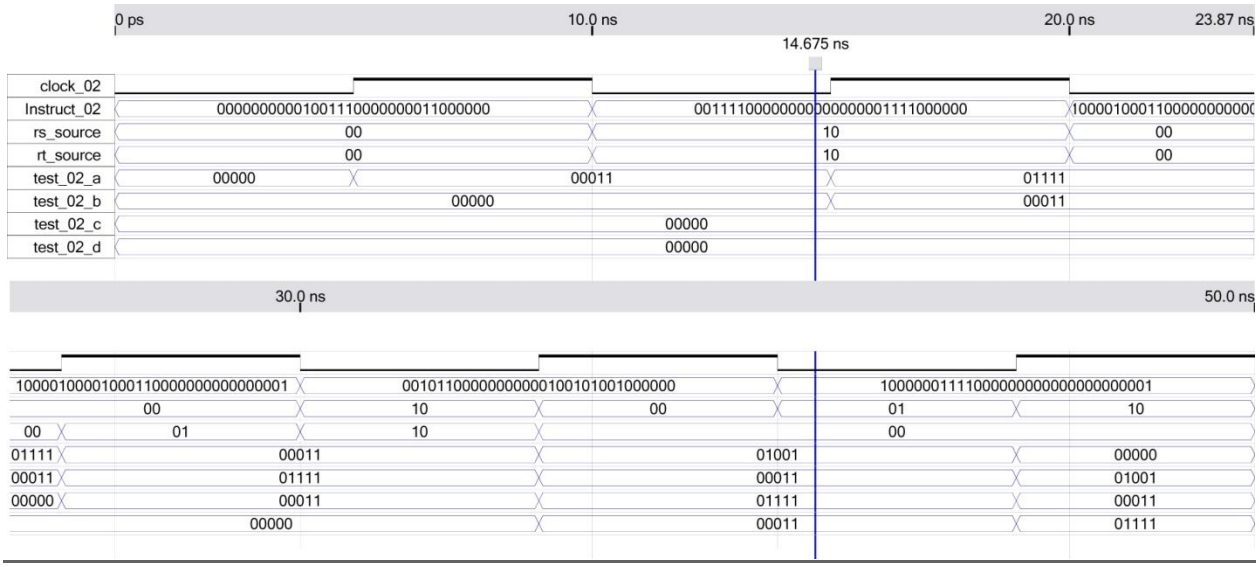

FIGURE CONTROL_2

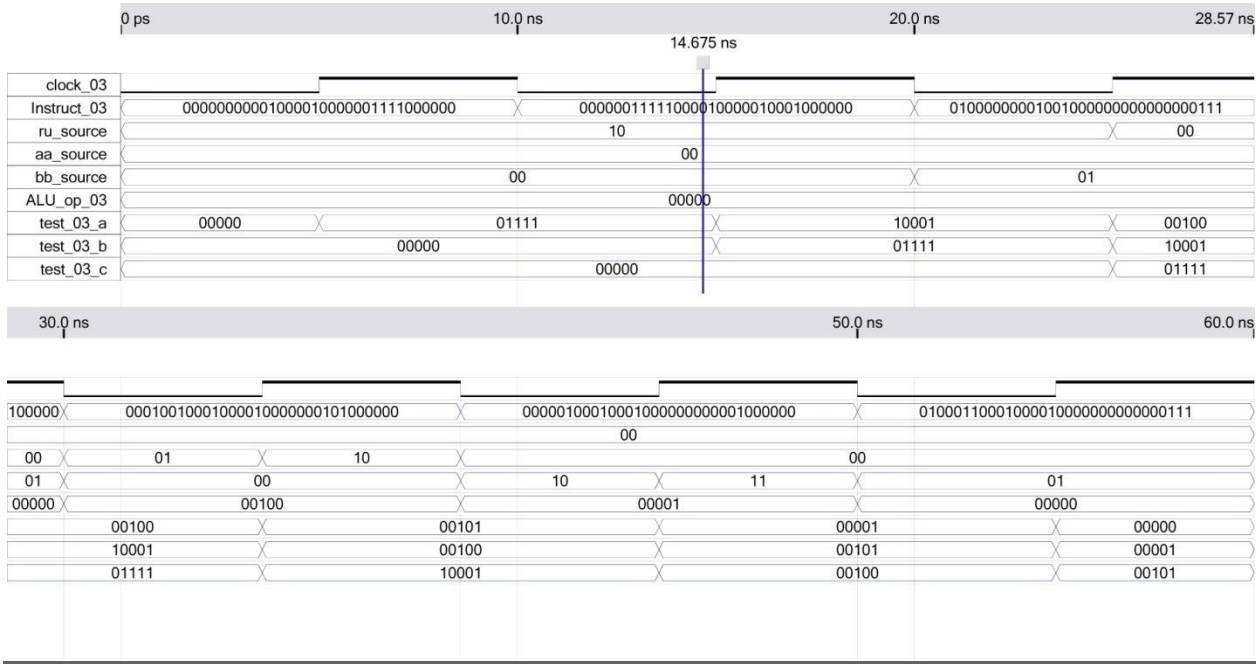

FIGURE CONTROL_3

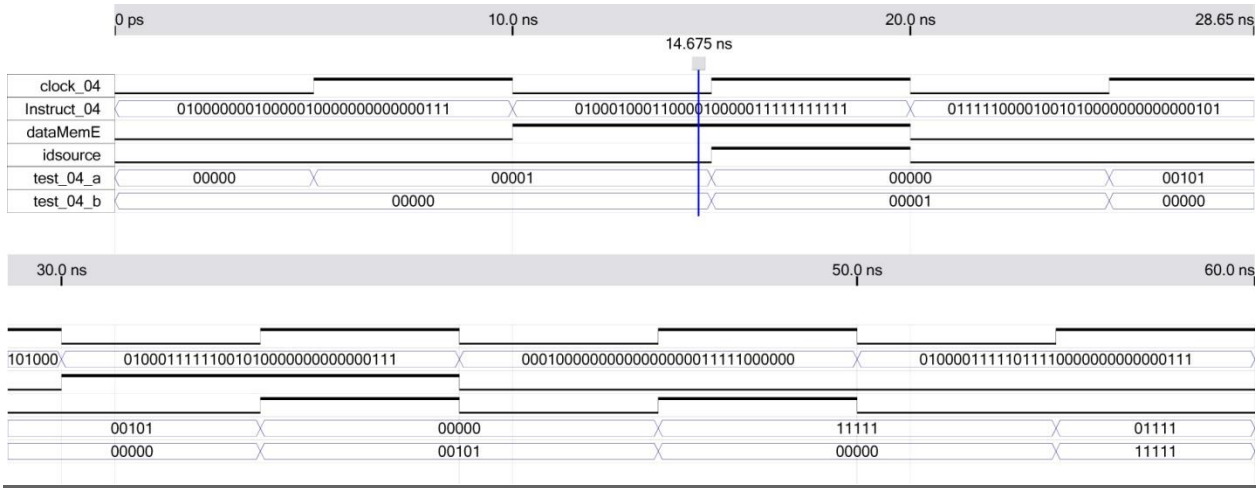

FIGURE CONTROL_4

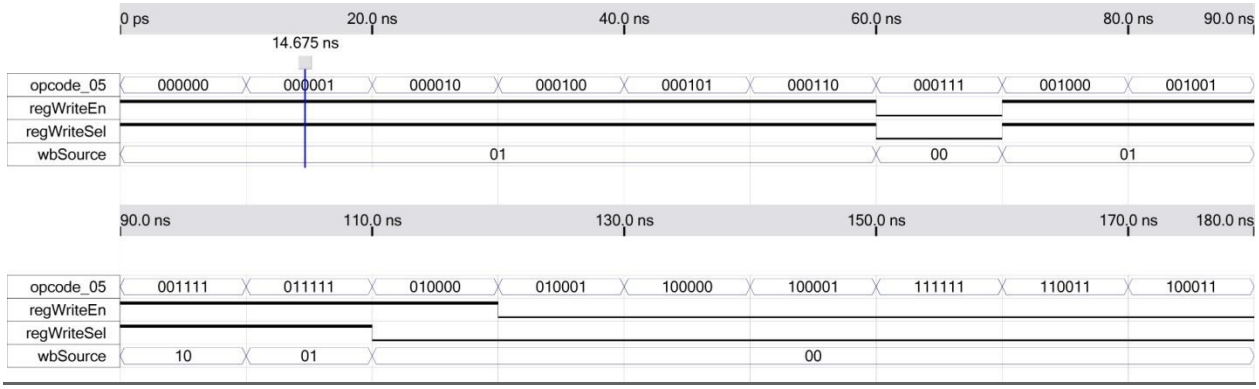

FIGURE CONTROL_5

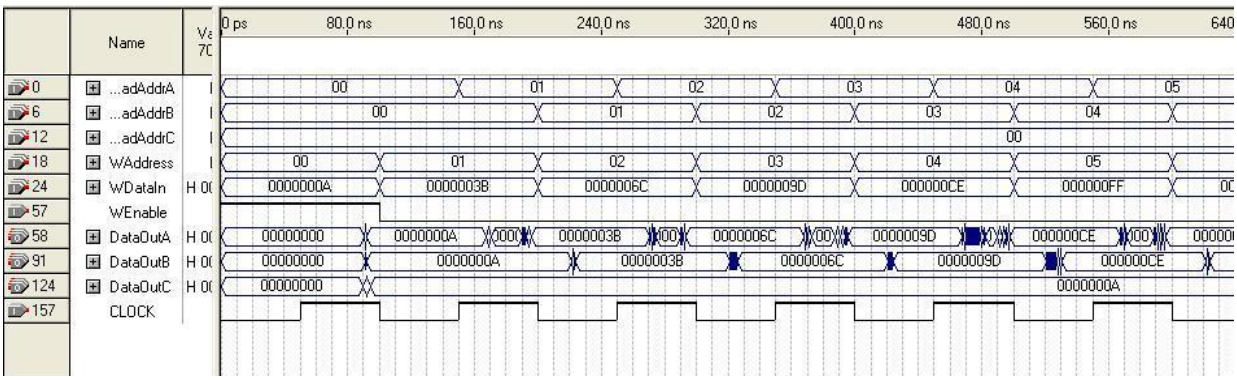

FIGURE REGISTER_32

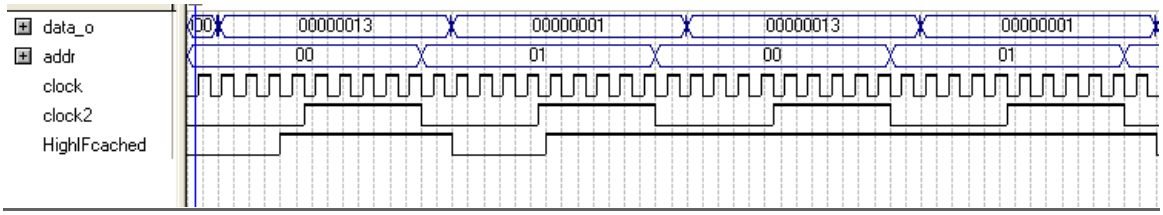

FIGURE DATA CACHE

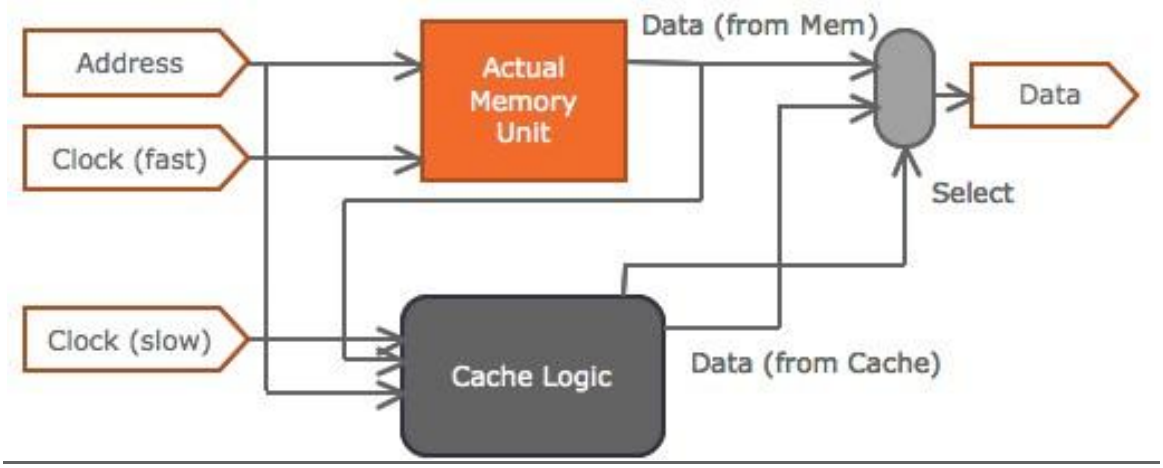

FIGURE INSTRUCTION CACHE 1

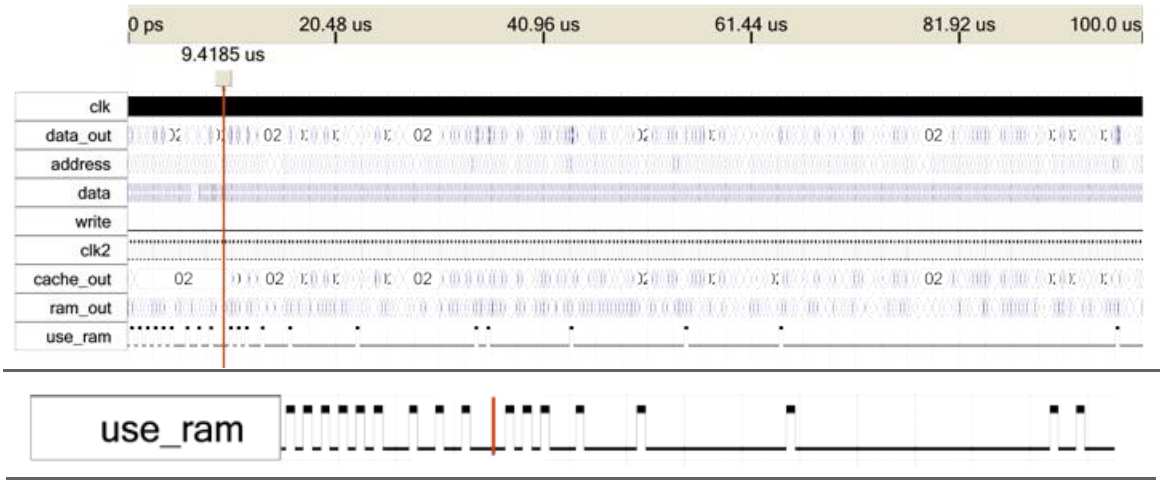

FIGURE INSTRUCTION CACHE 2

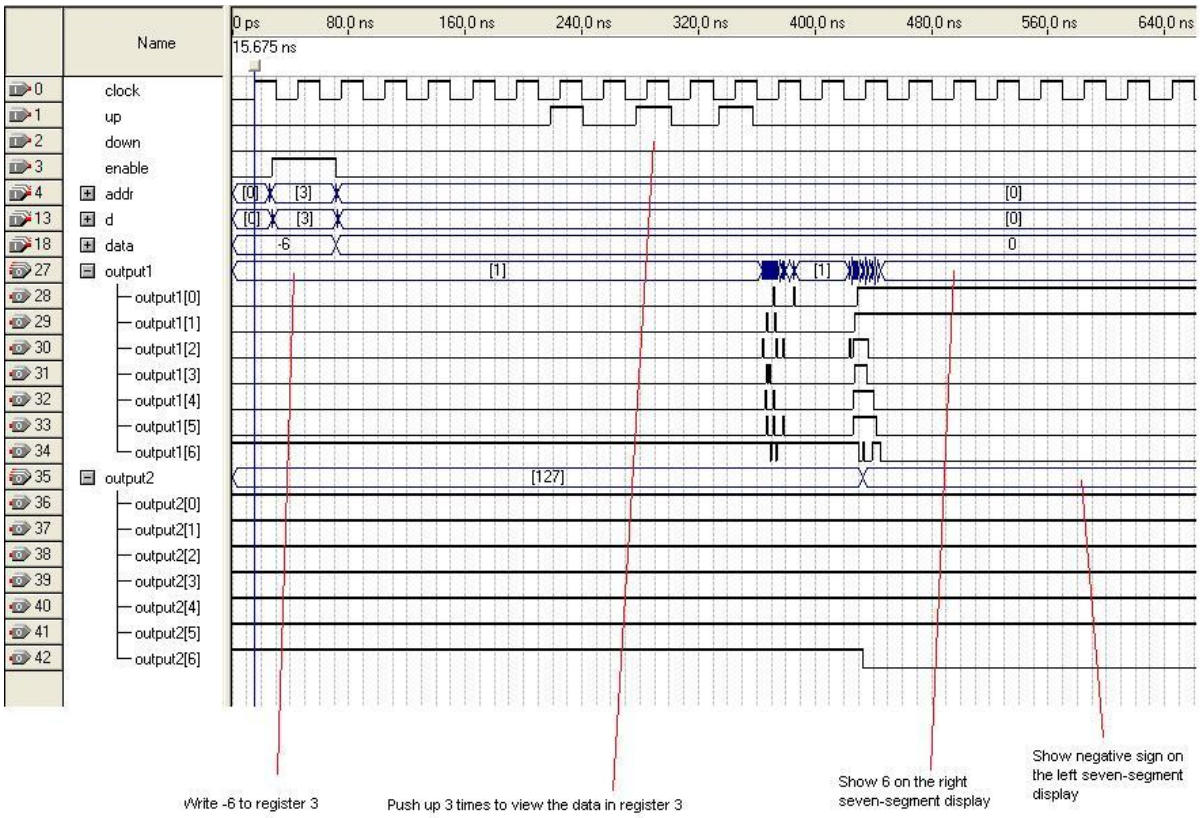

FIGURE I/O 1

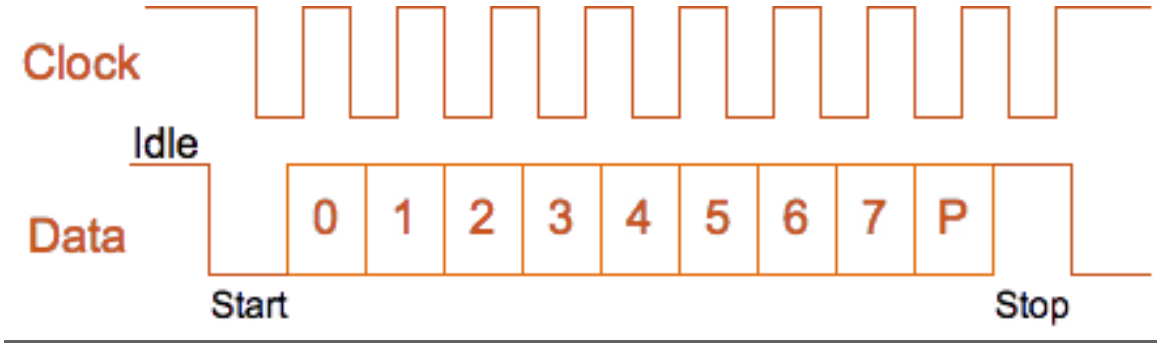

FIGURE I/O 2

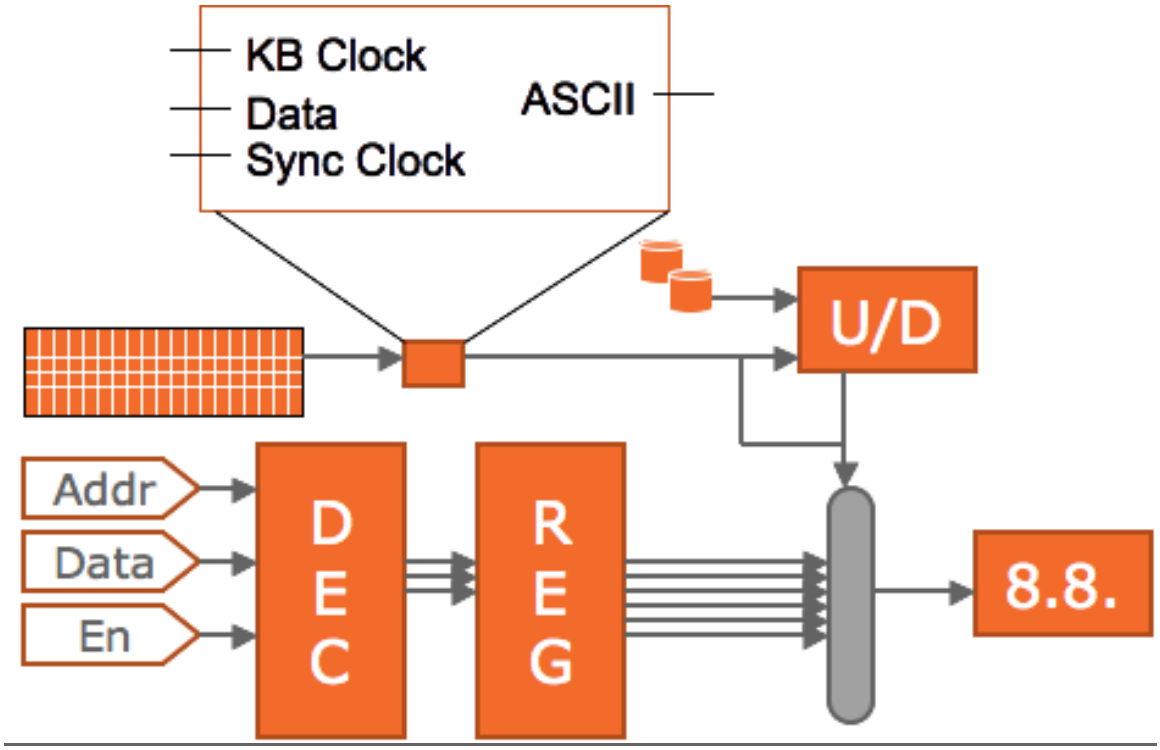

FIGURE I/O 3

Assembler – Example

The below example demonstrates the each major instruction type and its output:

INPUT:

```
beq $5 $6 count
j count
count:
add $5 $6 $7
print $5 $6 $7
sll $5 $6 5
gtpw $5
addi $5, $6, 7
lw $5 8($6)
beq $5 $6 count
j count
```

OUTPUT:

```
DEPTH = 256;
WIDTH = 32;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN
0:10000000101001100000000001000000
1:01111100000000000000000001000000
2:00000000110001110000000101000000
3:00100000110000000000000101000101
4:00111100000000000000000101000000
5:01111100110001010000000000000111
6:01000000110001010000000000001000
7:10000000101001100000000001000000
8:01111100000000000000000001000000
END
```