Design Journal

Runzhi Yang, Fred Zhang, Katrina Kerrick, Adam Finger

Contents

Milestone 1

Choosing a combination architecture of load-store and accumulator

Load and store is going to make our processor run much faster because it needs less interaction with the memory. However, only 16 bits are available for one instruction, so there are not enough space for three register addresses if we want to have many registers. Therefore, we decided to have only two register addresses for the R-type and store the value back to the first one. This is an accumulator processor with multiple registers. It has an register to accumulate on and another register to specify another operand the address of the operand in the main memory.
This way, we would have one or two register address per instruction so we are able to have 16 different registers and more freedom and less dependence on memory.

Register types

We decided to have $ra and $sp so we can do function calls and return fast.

We decide to combine $t and $a because they use the same convention except when calling a function. They are both temporaries and destroyed after a function call. Therefore, we decided to just use the first n t-registers as the argument registers. So for a function with n arguments, $t0~$tn would store those arguments at the beginning of the function call. If there are more than eight arguments, they would have to be stored on the stack.

Zero register

We decided to leave out a zero register because the only use for a zero register is comparison. The way that our branch command works, we do not need a zero register, only the zero immediate.

Leave a 3-bit unused space in R-type

In our R-type, we have 4-bit opcode, two 4-bit register and 1-bit ML, so we have 3 bits left. We are not sure what to do with them for now. We might consider to add a function code for shift or something else of use. However, since these things are extra, we decided to focus on more important parts of the project for now.

B-type

When we are considering how to do optional branches, we have a hard time to putting everything into a 16-bit instruction. We decided to take the same idea from the IA32 processor from CSSE132 and to utilize some flag bits to control branches. So whenever the ALU performs some calculations, we would update the flag bits accordingly. Right now we have three flags -- the negative flag, the zero flag, and the positive flag. Only one flag of N, Z and P can be 1. Those flags would be used in the b instruction, which has a condition code (CC). b would check the flag from previous executed results and decide whether it needs to branch or not. Each bit in CC would correspond to one flag.

Generally speaking, for each compare and branch, we need to first subtract two values and store flags. Then in the second instruction, we can compare the flag value and the condition code in the instruction and perform the branch if necessary. For the purpose of ease of programming, we would like to have 6 pseudo-instruction for branches -- beq, bne, blt, ble, bbl, bbe, and have our assembler to convert them into two real instructions

```
sub $ta,   $tb
b     CC   Label
```

CC stands for a 3-bit condition codes. From left to right, there is negative bit, zero bit, positive bit. There is a table available in Design document to indicate what CC is supposed to be for each pseudo-instruction.

Translation table from instruction to binary code

OPCODE	00	01	10	11
00	add	addi	sto	lui
01	sub	subi	cp	(blank)
10	and	andi	b	jal
11	or	ori	jr	j

Translation table from register name to register address

REGCODE	00	01	10	11
00	t0	t1	t2	t3

01	t4	t5	t6	t7
10	s0	s1	v0	v1
11	s2	at	ra	sp

*the operation codes go column then row

Explanation of tables

The instruction to binary table contains many instructions that are similar to MIPS instructions (i.e. add, addi, sub, and, j, etc). We would like to group instructions in the way above to simplify the ALU design. There are lots of instructions that ALU needs to perform and some of them are very similar. We can actually reuse many blocks to make our processor smaller. For example, add, addi, sub, subi all use adders, so they are grouped together.
There are also a few unique ones. One such instruction is cp which was included to add the ability to copy one register to another register. Another added instruction is b. The b instruction does branches for a pseudo instruction. The last different instruction is subi, which is included currently, but may be removed later to allow room for an additional instruction. One operation code, 0111, does not refer to an instruction currently, so far we have two ideas for this register:

1. Make it cpi to copy and immediate value into a register
2. Make it a shift instruction, which will take a lot of thinking to create in hardware.

The other table contains the codes for all of the registers. We used mostly the same conventions as MIPS for their names. So, the eight t registers are all temporary, the three s registers are guaranteed to be the same across procedure calls, the two v registers are used to return values from procedure calls, the at register is used during sudo instructions, the ra register is used to store the return address across procedure calls, and the sp register stores the stack pointer.

Should we have ‘push’ and ‘pop’?

After writing the assembly code for the example program, we realized that we are using a lot of sto with addi 4 and cp with subi 4. According to the design principle that we should make common case faster, we would probably want to make these two cases faster. Using two instruction takes 5 + 4 = 9 cycles for cp and 3 + 4 = 7 cycles for sto, but if we combine them into one instruction, it is possible to do cp in 5 cycles and sto in 4 cycles. However, this change might include more work in datapath and control designing later. We feel that it would be something that we would probably like to do first after the basic design is settled.
The modification will have backward compatibility, so we don’t have to redo our previous work. For the instruction format, we can use those previous instruction name with only one argument.

So r1 would remain the same, and the second data would be implies as popped out from the stack. We can use only one argument for sto, so by default, it simply pushes the register value onto the stack.
As for the binary encoding, we have 3-bit unused bit in R-type. Maybe we can make one of them a switch. So if the switch is on, we would simply perform pop or push with $bp and ignore 4 bits register address for r2.
We are not changing instruction specification for Milestone 1, but this would be something we might work on Milestone 2.

Ideas for exception handling

There is coprocessor registers in MIPS helping exception handling -- EPC, cause and status. EPC would stores the programmer counter where the exception happened. Cause stores the exception code and the flag bits when exception happened. Status stores the interrupt level, interrupt enable bit and any pending exception.
If we want to have our processor able to do exception handling, so sort of co processor like MIPS would have to exist. However, such co processor would require more instruction to move data in and out from the coprocessor, which would make our instruction set bigger.
To solve this problem without introducing new instructions, we considered several options:

1. Assign a few bytes in our memory to serve as ‘co processor’ and store EPC, cause and status when exception happened. So we have access to ‘coprocessor’ data with memory load and store instructions.
2. Still have a coprocessor, but forbid direct read and write with assembly instruction. When exception happens, EPC, cause and status are updated in the co processor, but a few of them are also put into t0, t1, etc as arguments of exception handlers. However, we need to backup those t registers into memory before load from coprocessor. The exception handler may update status, cause or other registers. Then after handler finishes its job, eret updates coprocessor registers with current t register values, restores original t0 and t1 from memory and jump to EPC.

Option 1 makes it easier to access and modify coprocessor register, but it’s pretty slow to have all coprocessor registers to memory.
Option 2 We need to do some extra data moving when exception happens and eret. Also, the more information we want our exception handler to know, the more restoration we need to do for t registers. Maybe we can stuff every information we need into one 16 bits register and only put one argument for exception handler.

Milestone 2

Instruction Set Changes

The instruction set saw many changes with this milestone. Many instructions were removed or changed into pseudo instructions. The only instruction that was removed entirely was subi. Subi wasn't necessary as a negative sign in front of the immediate in addi would serve the same purpose. This is the same as the addi (big) pseudo instruction, and we determined that it would save time in the Assembler and was intuitive enough to not need a subi instruction at all.
If an instruction was changed to be a pseudo instruction, it was to make room for the new set of instructions.
We decided to include push and pop in the instruction set because they both lower cycle time and they are used very commonly in code. Push and pop, however, need to be a regular instructions to see this benefit.
The cmp command compares two register values and sets the comparison code. The purpose of the comparison code is explained more in full under the next heading.
Cp and cpi copy values and put them into a destination register. This is necessary because our add instruction only takes two register parameters instead of three, so copying a value must become a real command instead of a pseudo command.
Xor became a command because we thought it would be useful for arithmetic. It wouldn't be too hard to add to the ALU and it could come in handy, so we added it.
Jalr (jump and link register) is a command we will possibly be adding that functions exactly like the MIPS jal command except that it jumps to a register instead of a label. This is so that we have a way to go to functions while setting the return address register. It is currently unclear whether we will add this command into the group of j-type commands or if it will be standalone.
J, jal, and eret became pseudo instructions. The only real jump instruction now is jr. We decided that the J-type instruction should see a few changes to make room for what type of jump is being implemented. It now looks like this:

opcode	register	jop	condition code

The opcode will indicate what type of instruction it is. Jop indicates what type of jump instruction it is (j, jal, eret, etc). Register indicates what register to look in for the jump address. 0 is an unused bit. Condition code is a three bit code that indicates when the jump should be made according to the comparison code.
Ori, andi, xori, clear, and addi (big) were added to the pseudo instruction set to make arithmetic easier. They were simple to add and cost us nothing.

Condition Code

Condition codes are very important to our instructions. The condition code is set when the instruction cmp is called.

```
cmp    $r1, $r2
```

The comparison code is three bits, with each bit representing a special value. Only one of the three bits will be 1 at a time! The other two will be zero. For example:
The first bit represents if $r1 < $r2. It is set to one if this is true, and produces a code of 100. If $r1 == $r2, then the code is 010. If $r1 > $r2, then the code is 001. While this could be done in two bits, doing it like this allows any command checking the condition code to check for not equals (101), greater than or equals (011), and less than or equals (110), which is three more values than if we had just used two bits.
The condition code is anded with the last three bits of each R-type, B-type, and J-type instruction. If the result of this and is zero, the line of code is skipped. The way this works is this:
Let $r1 = 1 and $r2 = 2

```
cmp    $r1, $r2
⇒ 100 because 1 < 2
add    $r3, $r4, 110
⇒ the add completes because 1 <= 2
```

Different registers are used for the add command because the registers used for the comparison aren't necessarily what's going to be changed in the next line.
The condition code is anded with the condition code from the instruction. If they are zero, the instruction does not complete because the statement (1 < 2, etc) isn't what the command was looking for. If the anded code is not zero, then there was a match somewhere (1 <= 2 is true because 1 < 2 or 1 == 2).
If a condition code is not specified in the command, it is set to the default of 111, which executes the command no matter the CPU's condition code.

Explanation of Load Memory Bit (LM)

LM bit enable our processor to load data from the memory and perform operation on it immediately. With the help of LM, memory read takes much less than a instruction. In MIPS, lw is one instruction and takes multiple cycles to perform. However, our processor can load from memory with an extra cycle.

As for array, access and write. Both MIPS and our processor needs to move the array pointer step by step, but our processor combine read memory and another instruction, saves a few clock cycles per memory read.

However, as for struct, MIPS has the advantage of access memory with a fixed offset. However, our processor would need to perform an addi to access specified data. This is kind of a drawback without offset immediate as in MIPS.

The actual reason of leaving out offset immediate area is that we only have 16-bit instruction. If we specifies two register in lw, there are only 4-bit left for the immediate. 16 Word offset is way too tiny. MIPS has 32-bit instruction, so it has plenty of spaces for immediate.

The Stack Pointer Register

The rtl design become pretty tough with $sp within the register file. For push and pop, we need to read or write the memory while updating $sp, but we need to use the interface of register and perform similar operation every time. We decided to create a totally separate register for $sp with easy interface for incrementing and decrementing, it it going to make push and pop rtl design so much easier and a few cycles fast.
The stack pointer register $sp is no longer accessible by the user because it does not need to be accessed. After the commands push and pop were added, the stack pointer does not need to be available to the user. It is no longer one of the sixteen main registers, and has been replaced with register s3.

Decision about exceptions and interrupt

Though we have plan eret for our processor, when it comes to design the rtl for eret, things get too complex to be straight out. Previous plan for restoring $t0 and put exception register as argument to exception handler will enable us to handle exception and return back to the program. However, after reconsidering the project requirement, we realized that exception handler is not really required. We do need ‘interrupt’ to provide input, call a certain function and display the result. For those interrupts, our CPU is not actually restarting from where exception happens, but jumps to a new locations and starts to perform new task. Those input I/O interrupts always abort the previous program and start a fresh new one. Therefore, for our project, we can do not really need to handle the exception, but simply reset CPU into a fixed initial state (given exception handler).
We decide to design and implement our CPU without exceptions. Later on in the term, we can simply add a few entrance to reset CPU to a fix state to perform the I/O interrupts required.

Milestone 3

How to enable our processor to instantiate arrays and objects

After the last meeting, we realized that our processor can not do anything involving a pointer like an array or an object, because there is no way to get an address to stack space. We think that being able to do array and objects are very important.
If we let the stack pointer be visible again, we will make push and pop slower. Right now they are even faster than regular calculation instructions like add and and. Fred came up with the idea of two stacks -- data stack and procedure stack.

The procedure stack keeps track of return address, restored ‘s’ and ‘t’ registers, while the data stack keeps track of arrays and other data that needs a reference. No pointer is allowed to point to any address within the procedure stack.
There is one ‘dp’ register. This register keeps track of the data stack. We can add a big value to it to create a big chunk of memory for an array. If a programmer wants to dereference some variable and use its address, we will increment dp by certain amount, and store the variable there. All pointers in the program have to point somewhere within data stack or heap.

The advantage of two-stack design is that it is immune from buffer overflow attack because return address and any array will be in two different stacks and no one can modifier return address in procedure stack. Also, it keeps our processor fast. We no longer have to update value of stack pointer every time we push or pop a value. Also, for function without allocating local addressable memory, we can save two instructions for increment and decrement data pointer.
The disadvantage of two-stack design is that we are giving more pressure on operation system to organize memory since we know have an extra chunk of memory needed to be kept separate.

Changes to the RTL

We changed the RTL to be more like the ones shown to us in class. We adjusted how it was displayed from showing cycles in columns to showing them in rows, and we simplified all of the R-types to be one column. However, all other commands still needed their own columns in the RTL. We also simplified the RTL by removing all blank middle cells and moving up commands, even if it means we can’t name all of the cycle anymore. To save room in our Design Document, we didn’t name the cycles at all.

Changes to Branches and Jumps

Branches and jumps had a lot of redundancy, and only in special use cases were they different. Our group changed branches and jumps to remove this redundancy.
We decided that we only needed two commands: jump register (jr) and jump label (jl). Jr became an R-type instruction and is now storing the register that it receives in the second register slot instead of the first, like so:

jr	empty register	$r1	empty	condition code

Branches were removed entirely, to be replaced by jl. Jl jumps to a label if the condition code is satisfied, but the label is a sixteen bit address. This is done by using thirty-two bits of instructions for each jl/jal. Each time a jl is read into the instruction register, the control Unit signals that the next line needs to be read in as the address to jump to. Then the condition code and whether to set $ra is evaluated. B-type instructions were removed completely.

Jl vs jal is set using one bit, rather than the five that were used for jop before, and a reference to $ra is passed if the command is jal.

Moore Machine Vs Mealy Machine

We decide to implement a mealy state instead of moore machine, so the control signal output depend not only on the state, but also the input signals. With this method, one control state can provide different control signals and different instructions can perform different work at Cycle 2. Though we might need to wait control to produce those bits, these time will not be very long, but we can save a whole cycle. If we start branch at cycle 2, we can make jl, jal, lui, cpi, push and pop one cycle faster.

Creating Tests

To create tests, we used a table format with one column listing a name and another listing a description of the test. For the unit tests, we decided to specify only units which we are going to implement rather than trying to test the xilinx/resources provided Muxes and other components. Our process for these tests was to list all inputs and assumptions then check the output using specific, immediate values. For the testing of the datapath, we decided to implement tests for unique cycles for instructions. For example, we only had one test for cycle 1, since there is only one unique type for all instructions. For cycle 2 there are multiple types of cycles.

Milestone 4

Control Unit design

Control Unit is a finite state with multiple different outputs. We have decided that we are going to use a mealy machine instead of a moore machine, so each state can have different control signals output and each instruction can start performing different task in cycle 2.
With mealy machine design, we do not have to create a state for every instruction. If certain instructions are doing very similar things in a cycle, they can have the same state to simplify the state design. For example, cycle 3 of jal, j and jr all update PC to a branch address, but jal also needs to write PC into register file. Instead of making three different states, we can just combine them into one, and set RW depending on whether it is jal. Therefore, we are able to shrink the number of states to 9.
There are many options we have to implement such a FSM. We can either use a 4 flip flop and represent them with binary encoding. This way we can minimize the number of flip flop we have but also make transition logic more complex and less intuitive. In our project, the number of flip flop is not as important as ease of design and speed. We decided to use one hot design. We

have 9 flip flop, one for each state. At any time, only one of those 9 flip flop can contain 1. This way we can convert FSM transition into machine level logic directly and intuitively. Though we would need more flip flop, state transition is faster and easier to design. For those output, we can take advantage of lots of “don’t care”. For example, if RW is 0, we don’t care what RWSrc is because its output is going to be thrown away. With the help of that, we need less logic gates and be able to reduce transition time. Each control signal output has a table that specifies state, input value and corresponding output.

control Unit Xilinx implementation

Datapath is easy to implement with schematic because it is very intuitive to see blocks. However, it wouldn’t be easier to implement control Unit with schematic, because there are going to be overwhelming gates and the diagram would be super messy.
In contrast, Verilog is a very powerful and useful tool to design hardware. With behaviour level Verilog, we can describe control Unit’s behavior and ask Verilog to generate corresponding hardware for us. We can use “if”, “else”, “case” to “program” hardware. It would much more easier to implement. Fred actually wrote and tested a behavior-level Verilog of control Unit overnight. However, behavior level Verilog cannot take advantage of “don’t care” bit. It can generate hardware that meets the requirement, but it also meets many unrequired additional conditions. Verilog is going to produce more hardware to fix values that we don’t actually care.
In our Xilinx file, behavior level control Unit is “ControlUnit.v”, and there is another file “ControlUnitFast.v”. Instead of descripting circuit behavior with “always block”, we will implement another control Unit at register-transfer level. In other word, we will draw out K-Maps for each control signals, and find the optimal logic design manually and then implement them with continuous “assign” in Verilog. This way we can take full use of “don’t care bits” and make our control faster!

Testing of control Unit

The testing of control Unit can be a bit tricky, since there are so many cases of control Unit. It would be super hard to cover everything. Also, it is going to take a long time to write every cases. One alternative way is create a ControlUnitTester module, which will take state, input to controls and output corresponding control signals and next state. For control signals that can be don’t care, ControlUnitTester would output an extra bit to specify whether this signal is “don’t care”. ControlUnitTester is implemented simply by copying every states in FSM. As long as programmer do not make typo, it would be correct. ControlUnitTester is all logic gates without storage, and serves as a standard answer producer machine. **In test bench, we can run ControlUnit with specified input combination for multiply cycles and check its output and transition to ControlUnitTester.** The checking also cover the case of “don’t care”. In actual code, for example, “MSrc” is the output from ControlUnit, and is compared with the standard answer “MSrca” from ControlUnitTester. If they equal or “MSrca” indicates “don’t care”, then

“MSrce” is going to be 1 to indicate that “MSrc” pass the test in this case. If any control signal do not pass the test, “error” would be high. Sometimes, our FSM would transit to some impossible state, with all control signals to be “don’t care”. We do not want to miss those errors, so ControlUnitTester also output an “error” bit to signal that the combination of input and states is not supposed to happen.

ALU Design

Our current ALU is designed with Verilog with a very simple behavior level description. Overflow is checked after we have the answer. We are not sure whether Verilog would automatically generate carry-look-ahead for us. Our ALU might be using the slow carry-ripple adder. Also, the overflow checking is very slow, since it needs to wait until adder finishes its job.
This design is not perfect, but we decided to move on with it. If we have more time latter, we will work on redesigning it in a lower level with carry-look-ahead.

Exception and I/O handling

For this milestone, our datapath is not able to do I/O handling and exception yet. We will fully test the core part first and add exception handling later. All we would need to is to add a RESET control signal to set PC to a specified address and move stack pointer to its original position. Whenever an exception happens, for example overflow or memory access error, our CPU would abort the current process and restart from a known process. If there is an I/O interrupt, we will set PC to a specific handler to handle this kind of I/O, for example read data, call relPrime().

In order to fulfill the requirement of I/O, there will be another control and even FSM at top level outside control unit and datapath. These controls are not general purpose, solely designed for this project. We will start work on Exception and I/O handling in next milestone.

Fixing Integration Tests

After the last milestone, we realized the integration tests we designed did not fulfill their intended purpose. For this milestone we fixed this problem by rewriting the tests. We tried to test every connection between two components by sending through inputs and reading outputs. For instance, we made sure the pc register and adder would only increment the pc’s value when pc write was set to 1. We did not consider edge cases in the design of the tests because they were covered during our unit tests.

Milestone 5

System testing

We add four muxes into our datapath to form datapath Test. Those muxes enable us to directly load and check data in register and memory. For system testing, we don't need to rebuilt memory with coe file for each test cases. We can just load instructions code in test bench --

```
run(startPC, numInstru)
readMem(address, out)
readReg( address, out)
writeMem( address, out)
writeReg( address, out);
```

Those testing interface make system test much easier.

Get rid of state 9

Everything is almost complete. We are looking for what we can improve. As we go through what our datapath is doing for each cycle, we realized that write back cycle is basically doing nothing compared to others. ALUO is only used to write data back to register in our design, and it only takes a very short amount of time for ALUO data to reach RWD of register file. The write back cycle is basically waste of time and resources.

Therefore, we decided to get rid of state 9 and combine write back and execution cycle of R-type and addi. In other word, the output from ALU won't be stored for a cycle, but directly written into register file. This makes our R-type and addi only 3 cycles long without compromising any clock frequency.

This change does not take a lot of time, since all we need to do is to delete everything related to ALUO.

Synchronous Design:

We decide to modify memory and register file to make them synchronous. In other words, memory and register file only updates its output at clock edge. Once the data are updated, they aren't going to change within the same cycle period. Synchronous component can been seen as asynchronous component with its output connected to flip-flop, which only update at rising clock edge. Since register file is already synchronous, we no longer need to store its output in some register temporarily. So we get rid of register A and B. Besides that, we also make some big

modification in order to make memory synchronous and operate in only one cycle. Those changes including making instruction register asynchronous and un-clocked and add an delayed mux are discussed in the next section.

Memory modification

Memory is pretty complex and slow. We will use a block memory to save more resources for other component. However, block memory is slower. We designed our RTL assuming that we have an asynchronous memory unit in the first time. By the time we realized that asynchronous memory is going to take too much resources, and we won't be able to implement our CPU in the board, it is already 9th week. We don't want to make too much changes up to this point.

The first option we have is of course, wait memory for an extra cycle. This is going to introduce an extra cycle, that is basically doing nothing -- write MD into Instruction Register. Also, we would need to add new states and make changes to our control. Making huge design changes in a late time is not a very smart choice. This option involves a lot changes in control and will slow our processor by around ⅓ .

The second option is to add an inverter in front of the clock of memory. Therefore, when other components experience a falling clock edge, memory experiences a rising edge and output the read data. Therefore, we don't need to wait for another cycle for the rising clock edge, and memory access can still be done within one cycle. However, this is actually makes execution time even longer, since memory read only at the first half of the cycle. If we give memory two cycle, we only add one extra cycle per instruction, but if we only use half cycle to read from memory, our clock cycle is actually going to double. This option won't introduce extra design work, but will double the execution time.

The third option is to delay the memory clock in the second option even more. We can delay the rising edge of memory to be ¾ cycle later than others, so it will use most of the cycle to perform read and we only need to slow our processor by a smaller amount, ⅓ in this case. We can even delay the clock edge ⅞ cycle or even more, and reduce the clock cycle. The way we can divide cycle time is to increase clock frequency and used use a counter to create slower cycle and phase shift. For example, if we originally have 100MHz clock and we want to introduce a ⅞ cycle delay in memory. We can first set primary clock to 800MHz, which controls a 3-bit counter. The counter would go from 0, 1, 2, up to 7 and then back to 0 each cycle. When the counter is between 0 and, it output 1, otherwise it outputs 0. This way we can generate a 100MHz clock from a 1600MHz clock. For the ⅞ delayed clock, it output 1 when counter is at 7, 0, 1, 2. In this way, we can get two clock we a ⅛ cycle difference. This option does not involve any changes in control, but we would need extra flip flops for counter. The other drawback for this option is power. If we double the clock frequency, it consumes 4 times more energy.

The last option or what we eventually implement is to give memory a whole cycle and make modifications to components involved with MD. The Xilinx built-in block memory is synchronous. We may see it as an asynchronous memory with a temporary register at its output. Instruction register and mux for B are modified.

Instruction register is changed to a 16-bit latch, no longer controlled by the clock. When its EN is high, instruction register would update its data regardless of clock. Therefore, read instruction can go in and out of instruction register in one cycle. However, the control unit will need to be modified to accommodate this change. In the old design spill out IW = 1 in state 1, but now it needs to have IW = 1 in state 2 to update its data. We need to delay IW. This is easy to do, since all instruction transit from state 1 to state 2. We can easily make control unit output IW = 1 at state 2.

Besides instruction register, B register is also using data from MD. Similarly, we need to do some minor modification for B register. Since MD is read one cycle later, we need to delay LM for only cycle to select whether we need B to be Reg[r2D] or MD. This is also easily achieved with a delayed mux. Delay mux would delay the control signal by one cycle with a flip-flop. So we don't need to modify complex logic of LM in control unit.

We update the datapath diagram in the design document, and Xilinx implementation. The new design works well and pass all RTL tests.