# 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.

**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.