Computer Systems and Architecture

Year Two, Semester One

Lecture One – An Introduction

What is Computer Architecture?

Computer Architecture can be studied a wide range of levels of detail. A hierarchy of levels at which computers are organised and designed can be summarised as:

1. High level language *
2. Operating System *
3. Instruction Set Architecture
4. General organising principles
5. Microarchitecture
6. Digital Logic
7. Semiconductor electronics

The Nature of Computer Programs

Computer Programs rely on a few core abilities - mainly interpreting input and output, performing logical and arithmetic operations, and assigning variables.

Programs contain an ordered set of instructions - which are usually executed sequentially unless otherwise stated – and the data to be manipulated. However, when represented in physical memory, instructions and data *cannot be told apart* and so **must** be kept logically separate. This separation is typically achieved by storing them in physically separate regions of memory, and transporting them using separate data-paths.

Requirements of a Computing Device

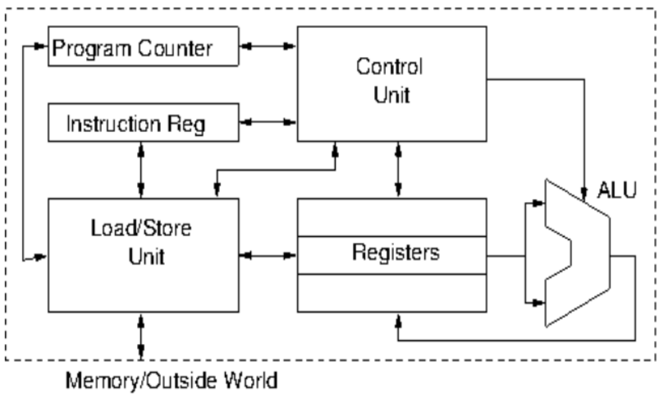
There are a few core requirements for a computing device to be useful. The device must be able to, at the very least:

- Load the program from some external device/memory (i.e. must have an interface with the outside world).
- Process instructions in the correct order - requires a mechanism to keep track of progress, local storage and decoding of instructions
- Access pieces of data in accordance with the program's instructions.
- Perform computations, using a calculation "engine".
- Take decisions according to the results of the computations – which requires mechanism for control.
- Send the results of the computations to some external device (i.e. again, must have an interface with the outside world).

A good starting point for learning how these requirements can be met is by studying the von Neumann Architecture.

The von Neumann Architecture

The diagram on the right shows the simplest possible representation of the von Neumann Architecture – a way of setting up and laying out the components of a computer's CPU.



It satisfies all of the requirements we listed previously:

- o The Load/Store unit can receive/send data from/to the outside world.
- o The Program Counter keeps track of progress through a program, by storing the memory locations of the instruction that should be exectuted next.
- o The Instruction register stores the instruction to be executed next.
- o The Control Unit decodes instructions and controls the operation of the other components, using a system of electrical command connections.
- o Each register is a small block of memory, each of which can store one item of data.
- o The Arithmetic Logic Unit (ALU) is responsible for carrying out calculations and comparisons.

This basic idea has had significant influence on the architectures that have come since.

Executing Programmes

Programmess typically run through four stages. These are:

1. **Load** – The programme is put somewhere accessible (main memory).
2. **Fetch** – Instructions are loaded from memory into the CPU.
3. **Decode** – The function of the instruction is determined, relevent data is fetched, and the components of the CPU are configured.
4. **Execute** – Perform the calculation.

We can step through this process, taking the von Neuman as the context.

Load

Put the programme in memory, and the address of the first instruction (known as the **entry point**) into the Program Counter.

Fetch

Fetch from memory the instruction at the address currently in the Program Counter, and update the Program Counter to point at the next instruction.

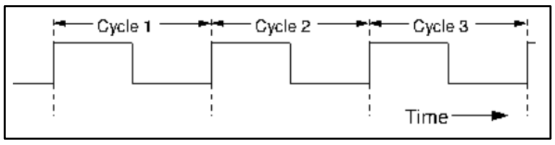
Decode

Determine the type of instruction. Then, calculate the memory address of any relevant pieces of data and fetch it if necessary.

Execute

Perform the calculation and store the result. Update the Program Counter if the instruction was a control instruction (such as a selection or iteration statement).

Cycles



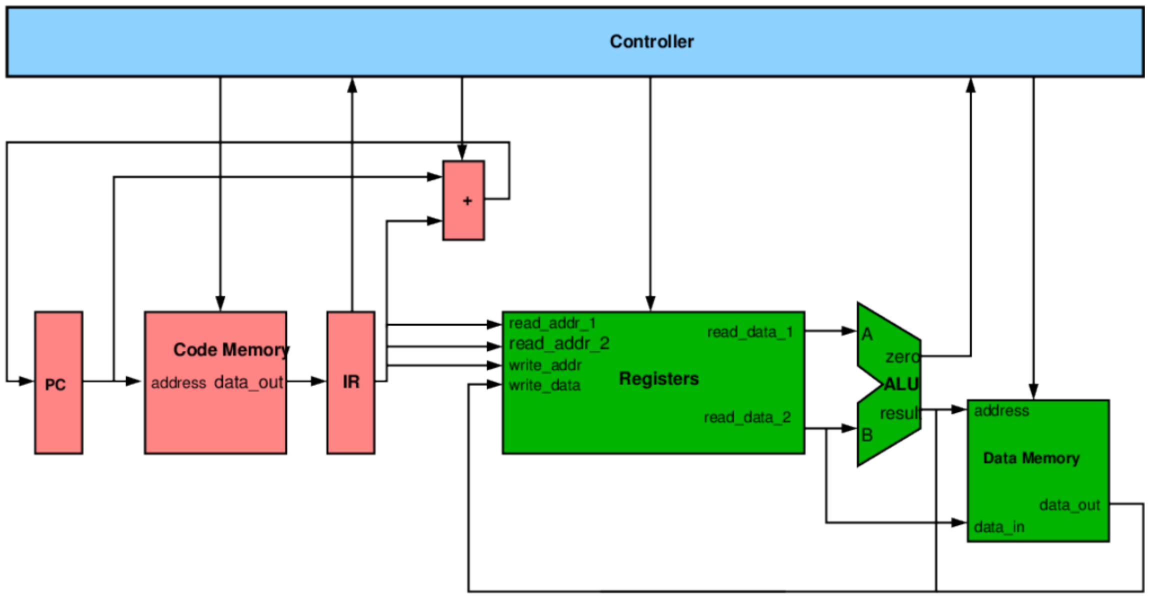
The widths of wires, resistance of contacts etc. will vary within a CPU, due to their extremely small size and high complexity. Individual components cannot be relied upon to carry data or perform comparisons at the *exact same* speed. Temperature also affects the performance of CPU components. This means that we cannot rely on important things happening within the CPU at the right speed or even in the correct order.

The solution to this problem is the **clock cycle**, which provides synchronicity to the systems within the CPU. One "tick" of the clock cycle is typically engineered to be at least the duration of a single operation under worst-case conditions.

The **instruction cycle** is synchronised by the clock cycle, but the two cycles are ***not the same thing*** - some instructions take multiple clock cycles to run, while others only take a single cycle. The clock cycle triggers successive stages of the instruction cycle.

Case Study: MIPS R4000

The MIPS R4000 architecture is similar to von Neumann's, but has separate interfaces for instructions and data. It is a modified version of the Harvard architecture. Below is a heavily simplified block diagram representing the MIPS architecture. We'll be going into far deeper detail as we progress through this module.

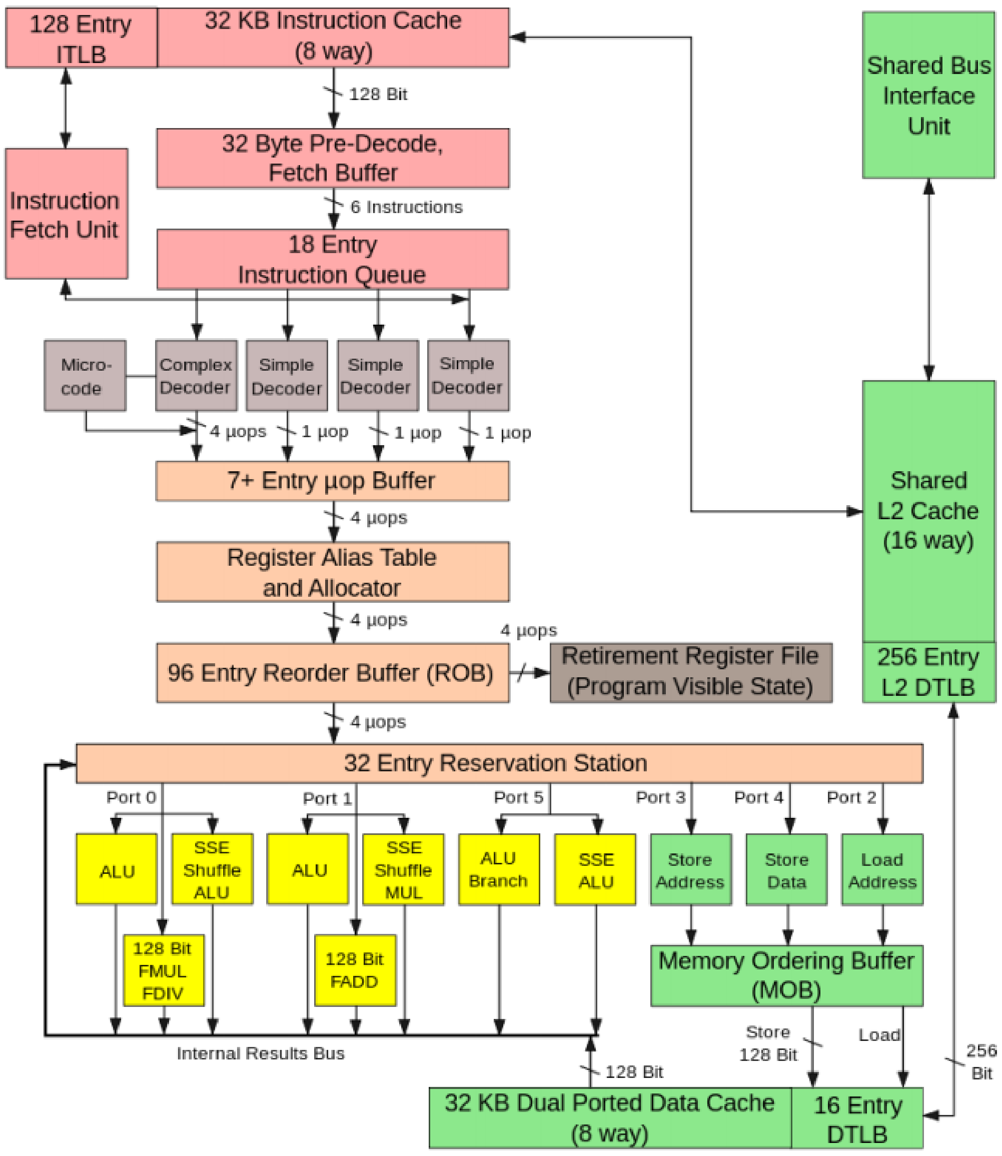


Case Study: Intel Core

Our second case study for this module will be the Intel Core 2 architecture, which is a *heavily* modified Harvard architecture. It the result of many decades of development in this area, and is therefore significantly more complex than MIPS.

The Intel Core architecture uses a huge range of performance enhancing techniques, such as caching, pipelining, and programme prediction.

A block diagram representing the Intel Core architecture in its simplest form is below.



Lecture Two – Instruction Sets and Assembly Language

An **instruction set** is the set of operations natively supported by a particular architecture. This set is the programmer's "view" of the machine, as it tells the programmer the exact operations that the architecture is capable of performing.

Each instruction has two representations:

1. Machine Code – Computer-readable binary format.
2. Assembly Language – Human Readable mnemonic for the machine code.

We will use the C language as our language for high-level language examples. This is because C code compiles directly into assembly language.

An Example

```
#include<stdio.h>

void main(){
    int a=1, b=2,c;
    c = a+b;
}
```

The C program on the right has a *very* simple task – adding two integers together and storing the result in a new integer variable.

If we compile this program, we'll get the x86 assembly language code below. We'll mainly be studying MIPS assembly language throughout this module but x86 is enough to give a general idea of the nature of assembly language.

```
main:
.LFB0:
        .cfi_startproc
        pushq  %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq   %rsp, %rbp
        .cfi_def_cfa_register 6
        movl   $1, -12(%rbp)
        movl   $2, -8(%rbp)
        movl   -8(%rbp), %eax
        movl   -12(%rbp), %edx
        addl   %edx, %eax
        movl   %eax, -4(%rbp)
        popq   %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

The first obvious characteristic of this code is its length. It takes many more steps to do achieve a goal in assembly language than it does in higher level languages. This is primarily because the amount of *data movement* required, with data being moved in and out of registers for assignment, calculations and keeping track of execution progress.

Types of Instruction Set

There are two main types of Instruction Set, although the lines between the two are very blurred in modern-day architectures.

Complex Instruction Set Computer (CISC)

This type was exhibited primarily by consumer-targeted CPU manufactures such as Intel and AMD. The instructions are considered "rich", allowing for high-level code to be compiled into fewer lines of assembly, but requiring complex hardware – thus making debugging and optimisation difficult.

Reduced Instruction Set Computer (RISC)

RISC machines use instructions that are far more simple than those used by a CISC machine – in that they performed one action, and one action only. They're used typically to refer to "load/store" architectures.

Modern CISC machines have RISC cores.

Programming with the Instruction Set

It can be said that if Functional Programming deals with "what computation you want", and if Imperative Programming deals with "what computation you want and how you want it done", Assembly language deals with:

- What
- How
- ***Where***

Everything must be defined explicitly, as assembly language translates directly into machine code. Resource management is also the responsibility of the programmer.

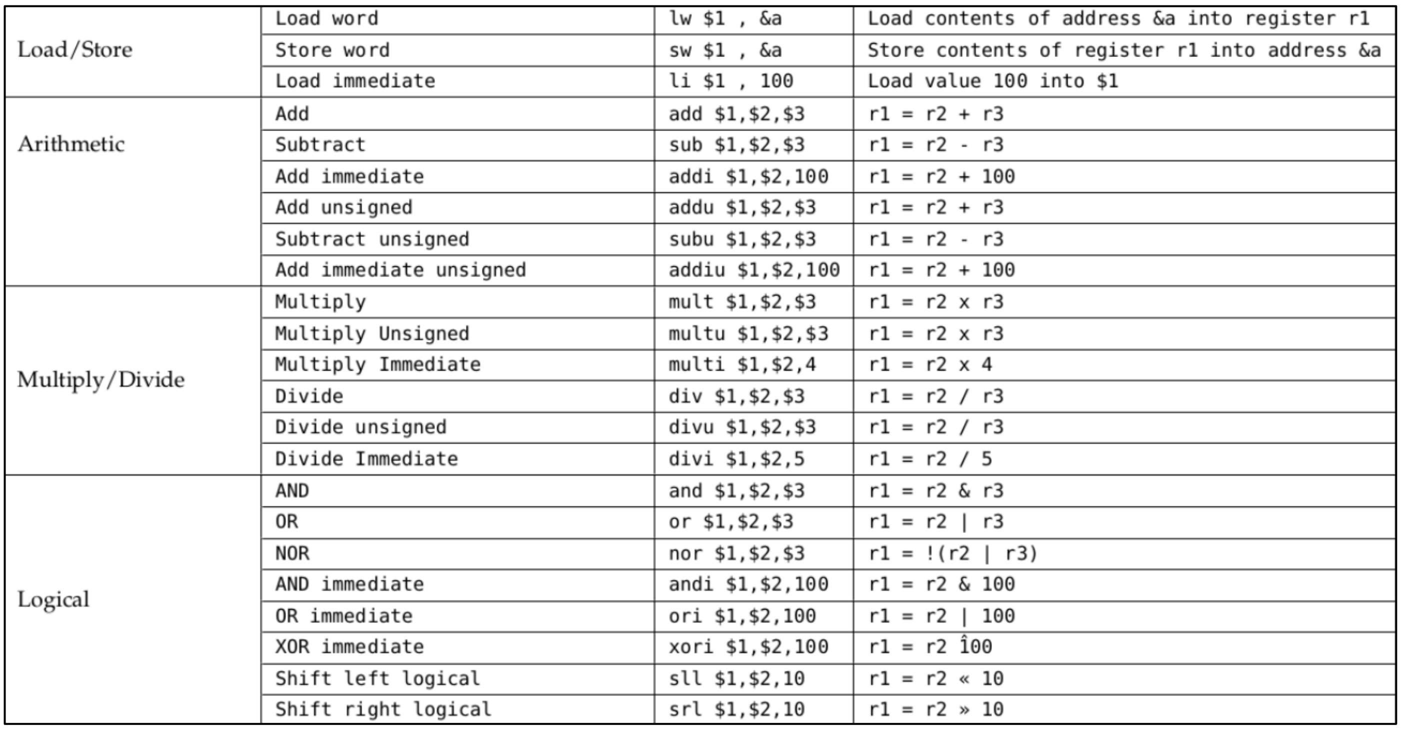
The MIPS Architecture

MIPS is a 32-bit architecture, meaning instructions, data and addresses are all 32-bits in size. MIPS is also:

- Byte Addressed – Addresses refer to a single byte of memory.
- Word Aligned – Data is padded out to take up a whole word-length (32 bits in this case).
- Big-endian – Addresses point to the most significant byte in the word.

MIPS has 32 general purpose registers (from $0 - $31). There is convention for the purposes certain ranges of the registers are put used for, but these are not enforced by the hardware at all.

Most instructions in the roughly 60-strong instruction set can only use registers, leaving a couple of special instructions for fetching data from memory and placing it into a register. Some of the most important instructions are below:



Load/Store	Load word	`lw $1 , &a`	Load contents of address &a into register r1
	Store word	`sw $1 , &a`	Store contents of register r1 into address &a
	Load immediate	`li $1 , 100`	Load value 100 into $1
Arithmetic	Add	`add $1,$2,$3`	r1 = r2 + r3
	Subtract	`sub $1,$2,$3`	r1 = r2 - r3
	Add immediate	`addi $1,$2,100`	r1 = r2 + 100
	Add unsigned	`addu $1,$2,$3`	r1 = r2 + r3
	Subtract unsigned	`subu $1,$2,$3`	r1 = r2 - r3
	Add immediate unsigned	`addiu $1,$2,100`	r1 = r2 + 100
Multiply/Divide	Multiply	`mult $1,$2,$3`	r1 = r2 x r3
	Multiply Unsigned	`multu $1,$2,$3`	r1 = r2 x r3
	Multiply Immediate	`multi $1,$2,4`	r1 = r2 x 4
	Divide	`div $1,$2,$3`	r1 = r2 / r3
	Divide unsigned	`divu $1,$2,$3`	r1 = r2 / r3
	Divide Immediate	`divi $1,$2,5`	r1 = r2 / 5
Logical	AND	`and $1,$2,$3`	r1 = r2 & r3
	OR	`or $1,$2,$3`	r1 = r2 | r3
	NOR	`nor $1,$2,$3`	r1 = !(r2 | r3)
	AND immediate	`andi $1,$2,100`	r1 = r2 & 100
	OR immediate	`ori $1,$2,100`	r1 = r2 | 100
	XOR immediate	`xori $1,$2,100`	r1 = r2 î100
	Shift left logical	`sll $1,$2,10`	r1 = r2 « 10
	Shift right logical	`srl $1,$2,10`	r1 = r2 » 10

Branches/Conditionals	Branch on equal	beq $1,$2,100	if (r1 == r2) PC = PC+4+100
	Branch on not equal	bne $1,$2,100	if (r1 != r2) PC = PC+4+100
	Branch on < 0	bltz $1, 100	if (r1 < 0) PC = PC+4+100
	Branch on <= 0	blez $1, 100	if (r1 <= 0) PC = PC+4+100
	Branch on > 0	bgtz $1, 100	if (r1 > 0) PC = PC+4+100
	Branch on >= 0	bgez $1, 100	if (r1 >= 0) PC = PC+4+100
	Branch on < 0 and link	bltzal $1, 100	if (r1 < 0) r31 = PC; PC = PC+4+100
	Branch on >= 0 and link	bgezal $1, 100	if (r1 >= 0) r31 = PC; PC = PC+4+100
	Set on less than	slt $1,$2,$3	if (r2 < r3) r1 = 1; else r1 = 0
	Set less than immediate	slti $1,$2,100	if (r2 < 100) r1 = 1; else r1 = 0
	Set less than unsigned	sltu $1,$2,$3	if ($2 < $3) $1 = 1; else $1 = 0
	Set less than immediate unsigned	sltiu $1,$2,100	if (r2 < 100) r1 = 1; else r1 = 0
Jumps	Jump	j 10000	PC = 10000
	Jump to register	jr $31	PC = $31
	Jump and link	jal 10000	r31 = PC + 4; PC = 10000

As you can see, most of the instructions make use of the registers, and all use explicit memory referencing instead of symbolic variable representation.

Using the MIPS Instruction Set

Example One

Say we wish to perform the following computation:

`a = b + c`

We need to think about how to programme this in quite a different way to what we're used to. As MIPS is load/store (can only operate on registers) we need to put variables **b** and **c** somewhere accessible before we can perform the addition. We'd also need to store the result suitable. A rough outline of the programme would be as follows:

1. Load **b** into registers - **`lw $8, &b`**
2. Load **c** into registers - **`lw $9, &c`**
3. Add **b** and **c**, put the result in a register - **`add $10, $8, $9`**
4. Store result from register to memory - **`sw $10, &a`**

Example Two

Here's another simple code fragment.

```
if (a==b) c=1;
else c=0;
```

This would be broken down into the following MIPS instructions:

```
        lw $8, &a             // Load a.
        lw $9, &b             // Load b.
        beq $8, $9, EQ        // If a==b go to EQ.
        li $10, 0             // Set c=0.
        j ST                  // Jump to ST.

EQ:     li $10, 1             // Set c=1.
ST:     sw $10, &c            // Store c.
```

«INSTRUCTION TYPES»

LECTURE THREE – BASIC ASSEMBLY LANGUAGE PROGRAMMING

Register Conventions

In the last lecture, we alluded to the fact that there are conventions for register usage in MIPS assembly language. These are not enforced by the hardware, but adherence is highly recommended to avoid bugs and compatibility issues when working with large programmes. The conventions are as follows:

Name	**Number**	**Use**
$zero	$0	Constant 0
$at	$1	Assembler temporary
$v0 - $v1	$2 - $3	Function return and expression evaluation
$a0 - $a3	$4 - $7	Function arguments
$t0 - $t7	$8 - $15	Temporaries
$s0 - $s7	$16 - $23	Saved temporaries
$t8 - $t9	$24 - $25	Temporaries
$k0 - $k1	$26 - $27	Reserved for OS kernel
$gp	$28	Global pointer
$so	$29	Stack pointer
$fp	$30	Frame pointer
$ra	$31	Return address

Be sure to stick to these conventions as closely as possible – it'll save you a lot of hassle in the long run.

A Working Example

We stepped through some very basic examples in the last lecture. Now we're going to make last lecture's "Example One" code snippet into a valid MIPS programme by adding some boilerplate code:

```
      .data                 // Designates the following as data declaration.
      .align 2              // Aligns data to cell 2^2 (4) bytes wide.

vara: .word 2               // Defines the variables.
varb: .word 5
varc: .word 7

      .text                 // Designates the following as code.
      .globl main           // main can be accessed from external files.

main:                       // Labels the entry point of main.
      lw $50,varb
      lw $t1,varc
      add $t2,$t0,$t1
      sw $t2,vara

      li $v0,10             // Sets "exit" parameter.
      syscall               // Calls syscall – exits programme.
```

Place this in a file with a .s file extension and it will run successfully in either SPIM (here) or Simulizer (here).

Input/Output

There are no explicit Input or Output instructions in the MIPS instruction set. Instead, I/O devices are typically mapped to the memory layout, and interrupts are used to divert attention to these devices when required. This is all handled by the Operating System.

We've already touched upon OS function calls, though you may not have realised. Take a look at the final two lines of our last working example of MIPS code. What these lines are doing is an OS function call – by loading an OS code into the `v0` register, and performing a `syscall`. In this instance the OS code 10 is used, which tells the OS to terminate the programme.

However, there are other OS codes available:

OS Code	Function	Set Up	Result
1	Print integer	\$a0: integer to print	
2	Print float	\$f12: float to print	
3	Print double	\$f12: double to print	
4	Print string	\$a0: addr. of null-terminated string to print	
5	Read integer		\$v0: integer read
6	Read float		\$f0: float read
7	Read double		\$f0: double read
8	Read string	\$a0: input buffer addr. \$a1: max no. of chars	
9	Allocate heap memory	\$a0: no. bytes to alloc.	\$v0: address of allocated memory
10	Terminate		
11	Print character	\$a0: character to print	
12	Read character		\$v0: char read
13	Open file	\$a0: addr. of null-terminated filename \$a1: flags \$a2: mode	\$v0: file descriptor
14	Read from file	\$a0: file descriptor \$a1: input buffer addr. \$a2: max no. of chars	\$v0: Number of chars read.
15	Write to file	\$a0: file descriptor \$a1: output buffer addr. \$a2: no. of chars	\$v0: Number of chars written.
16	Close file	\$a0: file descriptor	
17	Terminate with value	\$a0: termination result	

For example, if we wished to read an integer from the input, we would put the OS code 5 into register **v0**, and perform a **syscall**. This procedure would look like this:

```
li $v0, 5          // Loads a value into the register.
syscall
```

The inputted integer would then be found inside register **v0**.

If we wished to prompt the user to make than input, we'd need to output a string. This can be achieved by defining a string in memory (perhaps in the Data section of the program, with a label), putting the address of that string into register **a0**, and the OS code 4 into register **v0**. Finally, we'd perform a **syscall**. If we had an asciiz (null-terminated) string stored at the label **prompt**, this procedure would look like this:

```
la $a0, prompt     // Loads the value at an address into the register.
li $v0, 4
syscall
```

Branches and Jumps

In most computer programs, we need to make decisions about the flow of the program (*selection*), repeat a piece of code many times (*iteration*), or execute a piece of code that is defined somewhere else. Each of these tasks require us to break the normal sequential program flow. MIPS provides several ways of doing this, known collectively as **branches** and **jumps**.

Branches

Branches occur when the flow of the program changes following a comparison between two things. Take a look at the following snippet of C code:

```
if (i==j) {
      x = a + b;
}
else {
      x = c + d;
}
```

To the untrained eye, it may seem obvious that the way to interpret this snippet in MIPS would be with use of the beq (branch-on-equal) instruction. However, this would actually cause the ordering of branches to swap. Instead, you may prefer to use the bne (branch-on-not-equals) instruction, which does the opposite to beq, thus maintaining the original flow of the program and making it more readable. Ultimately this is just a stylistic decision, so consistency is more important than compliance.

The MIPS representation of the C snippet above, using bne, would look something like this:

```
main:
      lw $t1, vari
      lw $t2, varj
      bne $t1, $t2, L1
      lw $t3, vara
      lw $t4, varb
      add $t5, $t3, $t4
      j L2
L1:   lw $t3, varc
```

```
            lw $t4, vard
            add $t5, $t3, $t4
      L2:   sw $t5, varx
```

Jumps

While a branch is used for changing the flow of the program in response to a condition, a jump is unconditional. We use jumps to implement iteration in MIPS, amongst many other things. Consider this very simple "For" loop:

```
y=0;
for (int i=0; i<x; i++) {
      y = y+1;
}
```

To implement this in MIPS, we need to understand what happens where in the loop.

Firstly, the initialisation of i as zero happens outside of the loop. We then enter the body of the loop – firstly checking the condition (i<x). If the termination condition is met, we jump to the end of the loop, but if not, we work through the main loop body. Next, we increment the counter, and finally, we jump back to the start of the loop. This would look something like the following when implemented in MIPS:

```
      main:
            lw $t1, varx      // Load x.
            li $t2, 0         // Load y=0.
            li $t3, 1         // Load i=1.
      LP:   sub $t4, $t3, t1  // Find difference between x and i.
            bgtz $t4, END     // Compare difference to zero.
            add $t2, $t2, $t1 // Perform main Loop body.
            addi $t3, $t3, 1  // Increment i.
            j LP              // Loop.
      END:  sw $t2, vary      // Store y.
```

Lecture Four – Memory Management

Occasionally, code is reusable by many parts of the program. To be able to use this code, we must be able to divert sequential program flow. We've looked at this before in the form of branches and jumps, but the use of functions comes with its own set of issues:

1. The function will likely require arguments. How do we pass them?
2. The function may also return a value.
3. How do we know where to return to?
4. How do we make sure the machine state is restored afterwards?

The MIPS register conventions can point us in the right direction for solving these problems. For example, registers **v0** and **v1** are used for function return values, and registers **a0**-**a3** are used for function arguments. Bearing this in mind, we could construct a very simple example snippet:

```
main:
        li $a0, 3
        j SQ
RET:    li $v0, 10
        syscall

SQ:     mul $v0, $a0, $a0
        j RET
```

What this snippet does is perform a squaring operation by jumping to a function named **SQ**, and then jumping back. It uses all the correct register conventions for passing arguments and returning the value.

However, it has a pretty big flaw. The **SQ** function's return address is **hard-coded**, so it's not re-useable. Luckily there are two instructions we can use to fix this problem:

1. **jal** – jumps to a given address, storing the return address in **$ra**.
2. **jr** – jumps to the address stored in a given register.

If we use **jal** instead of **j** when calling a function, the return address will be stashed in register **$ra**. Then, at the end of the function, we can use **jr** to return to the address stored in register **$ra**. This enables us to reuse the same function in multiple places in the code. Our previous snippet could be fixed to demonstrate this:

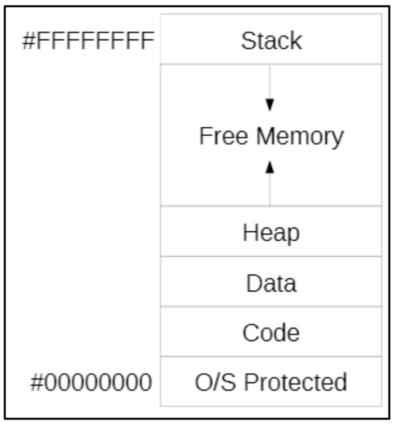
```
main:
        li $a0, 3
        jal SQ
        li $a0, 5
        jal SQ
        li $v0, 10
        syscall

SQ:     mul $v0, $a0, $a0
        jr $ra
```

Note how we no longer need the **`RET`** label. Our code now looks substantially more elegant.

The Stack

It may seem like we've fixed the problem of knowing where to return once we've finished executing the function – but what happens if our function calls *another function?* The return address stored in **`$ra`** will get overwritten by the new return address. We need to find a way around this. That's where the Stack comes in.



In MIPS, the stack sits at the top of memory, and grows downwards. We're allowed to write to the stack at runtime.

To keep track of where we are in the stack, we use one of the registers (**`$sp`**) as a stack pointer – which always stores the address of the most recently-added stack entry.

Before we start using the stack, we need to fully understand the variety of addressing modes available to us in MIPS. This is because so far we have only specified memory addresses explicitly, either using a label, or by specifying a register that contains an address (e.g. **`jr`**).

Addressing Modes

There are a few ways in which we can specify or compute memory addresses, depending on the type of instruction that we are concerned with. Since we only have 32 bits in which to specify all of the information needed to execute an instruction, it is clear that we cannot usually specify an absolute memory address - as we will have even fewer than 32 bits available. These modes can be summarised into the following four categories:

- **Register Addressing** – Where the address is stored in a register. Used when returning from sub-routines, as just discussed, such as with the **`jr`** instruction.
- **Base Addressing** – Where the address is specified relative to an address stored in a register. Far more common, due to usage for working with the stack. Used by the **`lw`** and **`sw`** instructions.
- **PC-Relative Addressing** – Where the address is calculated based on the current value of the Program Counter. Used mainly in branches where the memory we need access to is fairly close-by, such as by the **beq** and **bne** instructions.
- **Pseudo-Direct Addressing** - Used in the **`j`** instruction.

Using the Stack

The general process for a function that utilises the stack is as follows:

1. Make space on the stack by adjusting stack pointer (**`$sp`**).
2. Push data onto the stack.
3. Do function.

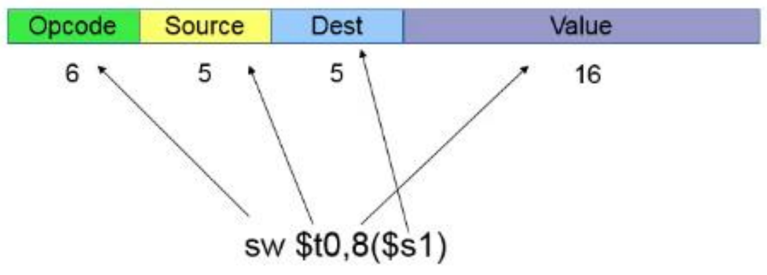
4. Pop data off the stack into their original registers.
5. Shift stack pointer back to original position.

In assembly language, this would look something like this:

```
f:
addi $sp, $sp, -12      // Adjust stack pointer (by 3 words).
sw $s0, 8($sp)          // Push s-registers onto stack.
sw $s1, 4($sp)
sw $s2, 0($sp)
...                     // Do function.
lw $s0, 8($sp)          // Pop values on stack back into s-registers.
lw $s1, 4($sp)
lw $s2, 0($sp)
addi $sp, $sp, 12       // Reset stack pointer.
jr $ra                  // Return.
```

Take a look at the **sw** and **lw** instructions. These are I-type instructions, and address memory using two parts: a register, and an offset.

In this case, the intended destination on the stack is calculated by adding the value 8 to the stack pointer. Note that there are only 16 bi available for storing an offset, which can be problematic if t grows particularly large.



Static vs. Dynamic Memory

It's possible to define data and arrays at compile-time. For example:

```
.data
.align 2
prompt:     .asciiz “\nEnter a number >”  // Define a string.
arr:        .word 0, 0, 0, 0             // Allocate a 4-word array.
```

This data is known as **static**, and is stored in the data portion of memory. As such, the size of the array cannot be adjusted during run-time. It is very unlikely that the required size of an array will be known at compile time, so instead we usually declare arrays in runtime using **dynamic** memory.

Dynamic memory is allocated on the heap. The exact location for the data is determined by the OS – meaning dynamic memory management requires system calls.

Heap memory is allocated using **syscall** code 9, with the number of bytes required stored in register **$a0**. Once the memory is allocated, it's address will be found in register **$v0**.

Recursion

The key issue with recursion is that data needs to be preserved at each call, such as arguments and the return address. Take a function that calculates the factorial of the given argument:

```
fact(n) P
        if n==1
                return 1;
        else
                return n*fact(n-1);
}
```

This is a recursive function. n and the return address must be preserved for each recursive call. As touched upon previously, this can be achieved using the stack.

In MIPS, this function would look something like the following:

```
FACT: subi $t0, $a0, 1
      bgez $t0, REC          // If N>1, go to recursive case.
      li   $v0, 1            // Else return 1
      j    RET               // Jump to the return

REC:  addi $sp, $sp, -8      // Decrement stack pointer.
      sw   $a0, 4($sp)       // Push argument onto stack.
      sw   $ra, 0($sp)       // Push return address onto stack.
      addi $a0, $a0, -1      // Set new argument as N-1.
      jal  FACT              // Recurse.
      lw   $a0, 4($sp)       // Pop argument off stack.
      lw   $ra, 0($sp)       // Pop return address off stack.
      addi $sp, $sp, 8       // Increment stack pointer.
      mul  $v0, $v0, $a0     // Calculate return value.

RET:  jr   $ra               // Return to appropriate place.
```

Recursion in MIPS is complex but far easier to understand if the underlying pattern is recognised:

1. Make room on stack.
2. Push argument/s to stack.
3. Push return address to stack.
4. Recurse.
5. Pop arguments.
6. Pop return address.
7. Calculate return value and return.

Lecture Five – The MIPS Microarchitecture

We are now going to start studying the MIPS architecture in detail, and look at how we can implement the MIPS instruction set. There are two parts to this:

- Control
- Datapath

We will break the implementation down into four sections:

- Instruction fetch
- R-Type instructions (such as **`add`** and **`sub`**)
- I-Type instructions (such as **`lw`** and **`sw`**)
- Branches and Jumps

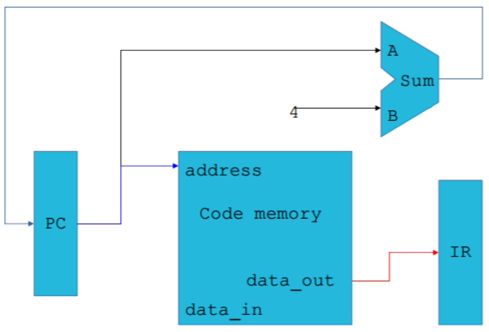
Building a Datapath

Instruction Fetch

The first part of building the Datapath is something common to every instruction – instruction retrieval. For this to happen, we need to:

1. Feed the address stored in the program counter (PC) into the code memory, in order to fetch the instruction.
2. Store the instruction in the Instruction Register (IR).
3. Increment the PC.

The hardware needed to achieve this is represented in the diagram on the right.

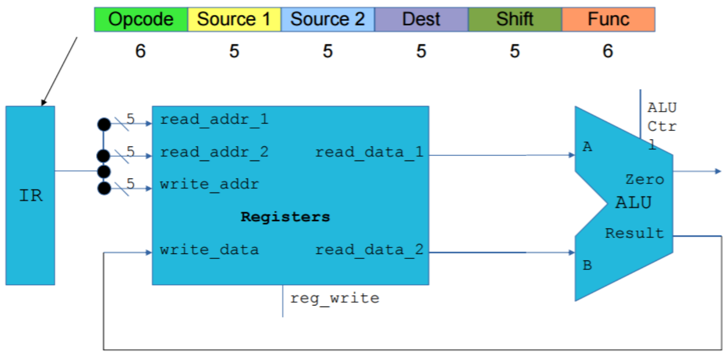


R-Type Instructions

Next, we should consider how we can process R-Type Instructions. The steps for this are:

1. Opcode is dealt with by Control Unit.
2. Feed addresses into relevent register ports.
3. Use ALU to process data in accordance with signal from Control Unit.
4. Write to destination register if necessary (controlled by reg_write signal from Control Unit).

The hardware necessary for supporting this is shown on the right.



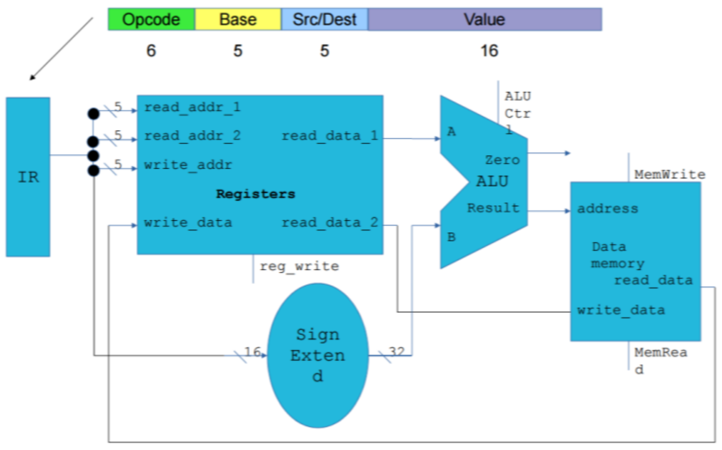
I-Type Instructions

Much of the hardware for supporting I-type instructions is already in place, following the implementation of R-type instructions. However we do need to make a few adjustments.

This time, the datapath will look something like this:

1. Opcode handled by Control Unit.
2. Determine whether the instruction is a load or a store, and feed the register address into the appropriate Register Unit port.
3. Extend 16-bit offset into 32-bit number (because base address will be 32-bit).
4. Add offset and base address together.
5. Feed absolute address into Memory Unit.
6. Determine whether instruction is a load or a store, and read/write data appropriately.

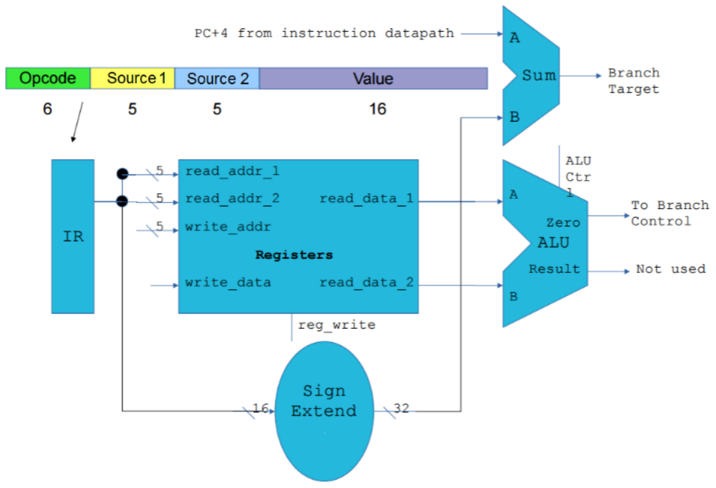
The hardware necessary for supporting this is shown on the right.



Branch Instructions

Branches add another layer of complexity to the datapath, as they influence the flow of the program.

1. Feed register addresses into the read ports of the Register Unit.
2. Output the data at those registers into the ALU for comparison.
3. Extend instruction address offset from 16 bits to 32.
4. Execute comparison in accordance with the type recquired, as determined by the Control Unit.
5. If comparison yields a result of zero, take the branch, by adding the offset to the incremented program counter (which will be on its way around the Instruction Fetch datapath at the time).
6. If comparison does not yield a zero, discard the result and leave the program counter unaffected.



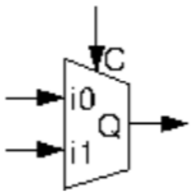
Combining Datapaths

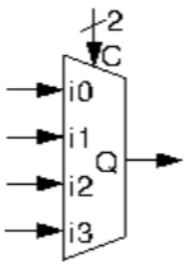
We've now established Datapaths for each type of MIPS instruction (apart from Jumps, but we'll come back to those later). The next task is combining them, which we should do without duplicating components.

However, we can't simply combine the wires between components and hope for the best. This is because it would result in the connection of multiple binary outputs to a single input – which if left uncontrolled would lead to indeterminate machine states.

Instead, we must introduce a form of active switching into the hardware.

Multiplexing is a method that allows components to share the communication channels, and vice versa. When discussing microarchitecture, a multiplexer is a component that takes multiple inputs and produces a single output, depending on a control input.





To avoid confusion, note that one binary output needing to be distributed to multiple inputs isn't problematic, and is called **fanning out.**

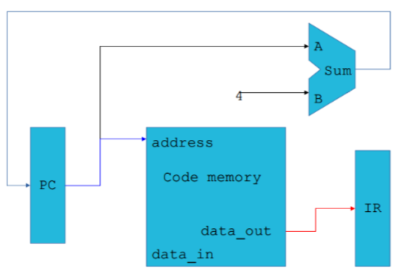
We will take a look at how to use multiplexers to combine our four datapaths into a single datapath in detail in the next lecture.

Lecture Six – Integrating the Datapath

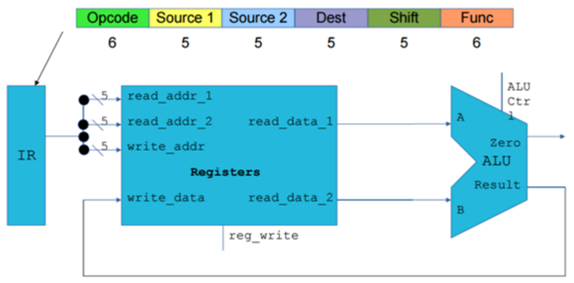
Recap

You'll recall that in the last lecture we formulated four "datapaths" that, together, would be capable of implementing nearly all of the MIPS instruction set. These datapaths are in the diagrams below:

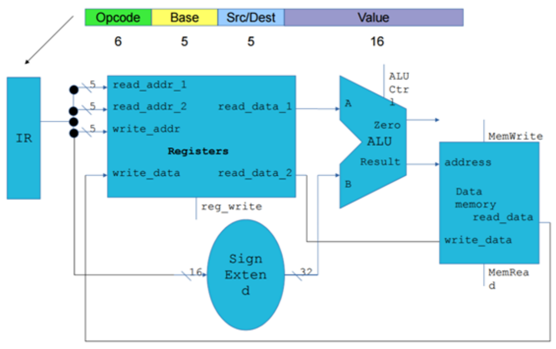
Instruction Fetch



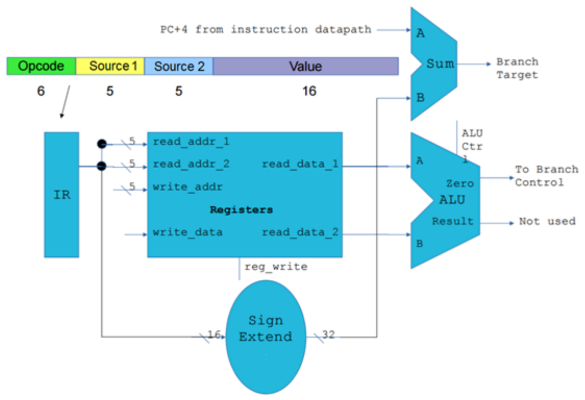
R-Type



I-Type



Branches

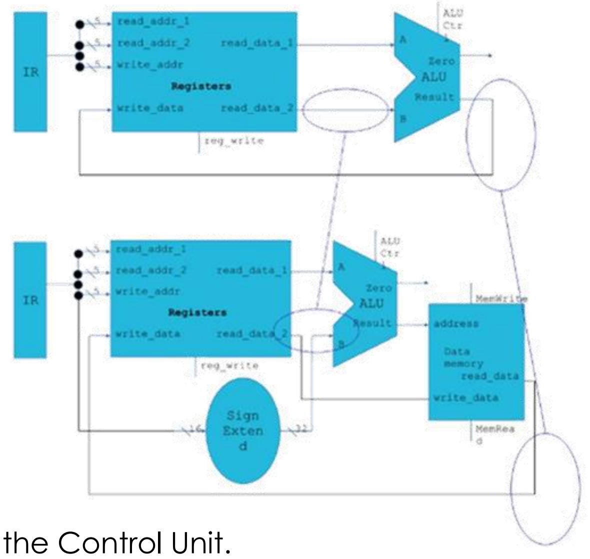


We now need to combine these into a single datapath with minimal component duplication. We'll use multiplexing to do this.

Integrating R-Type and I-Type Datapaths

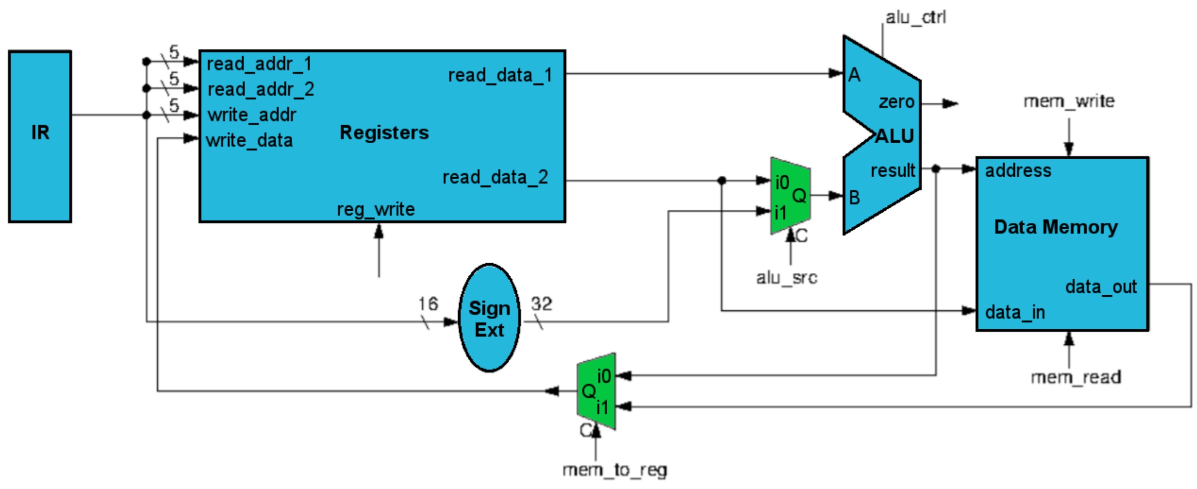
When we take a look at the R- and I- Type Datapaths in parallel, we can see that there are two main conflicts:

- ALU unit's **B** input:
  - **R-Type** – Taken directly from registers.
  - **I-Type** – Taken from Sign Extension Unit.
- Register's **`write_data`** input:
  - **R-Type** - Taken from result port of ALU.
  - **I -Type** – Taken from memory.



These conflicts can be resolved simply by adding a multiplexer at each conflicted input – both of which will be controlled by the Control Unit.

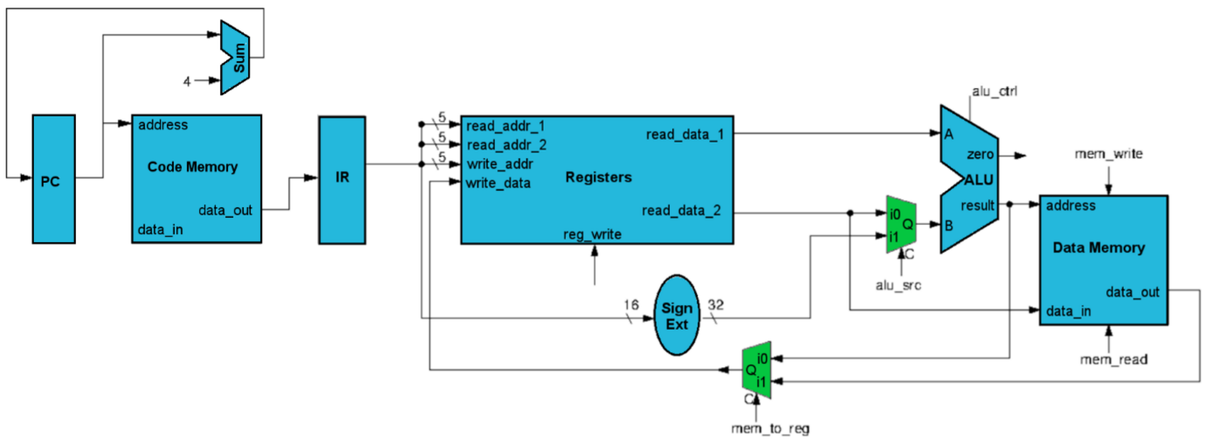
The combined datapath, with multiplexers, ends up looking like this:



The multiplexers are configured such that we can process R-type instructions by setting their control inputs (**`alu_src`** and **`mem_to_reg`**) to **`0`**, and process I-type instructions by setting them to **`1`**. The **`mem_to_reg`** is also used for preventing data from being written to registers during non-write operations.

Integrating Instruction Fetch Datapath

Plugging the components necessary for Instruction Fetch into our current design is completely trivial. It simply bolts onto the front, by connecting the Code Memory's output to the input of the Instruction Register. The diagram is below:



Integrating Branch Datapath

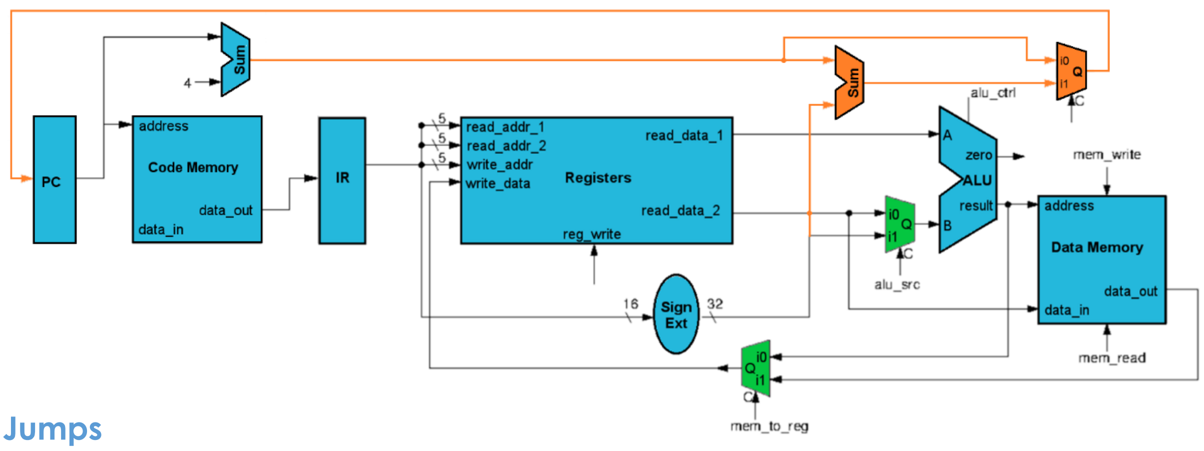
Much of the hardware necessary for supporting branching is already present in our design – there's simply a conflict with how the components are connected in both cases. This makes adding the hardware for supporting branches is more complex.

Here, we need to be able to influence the value being fed into the Program Counter:

- If the current instruction is a branch instruction, AND the ALU outputs a zero, we need to add the branch offset (transmitted by the Sign Extension unit) to the incoming instruction address, so that the branch target is put into the PC.

- Otherwise, we allow the incoming instruction address to bypass the addition and continue unaffected.

To achieve this, we need another multiplexer. The multiplexer will, again, take its control signal from the Control Unit, which determines whether we need the branch target address or the address of the next instruction. The offsetting will be done by a second Sum component. See the diagram, where changes are highlighted in orange:



Jumps

We haven't yet discussed the hardware configuration for implementing jumps. This is because it's far easier to just add the functionality to our existing model.

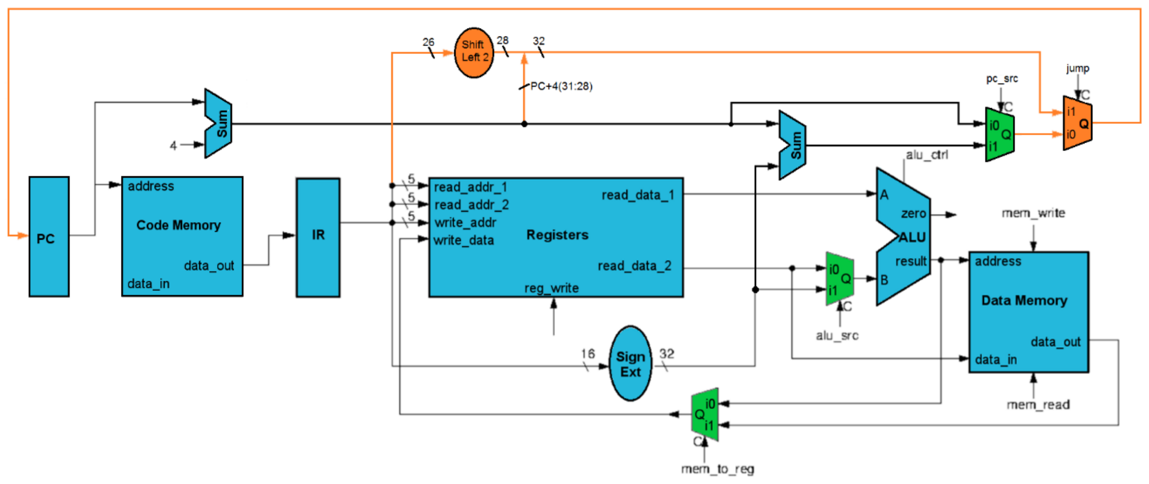
The 26-bit value in a Jump instruction is an absolute word address. This is because the instructions are fixed length and word-aligned, so the two least significant bits of the address aren't needed. Absolute word addresses also cannot be negative, so there is no need to perform sign extension.

Opcode	Value
6	26

Therefore, in order to convert a word address into a byte address, we need to left shift by 2 bits; and then prepend with 4 zeros.

This allows us to cover 2^{28} word addresses, or 256MB at the bottom of memory. To access anything further afield, the **jr** command must be used instead.

Our design now looks like the diagram below, where the changes are highlighted in orange:



With this design, we can now support the vast majority of the MIPS instruction set. However, we're not quite done yet.

R-Type Instructions Revisited

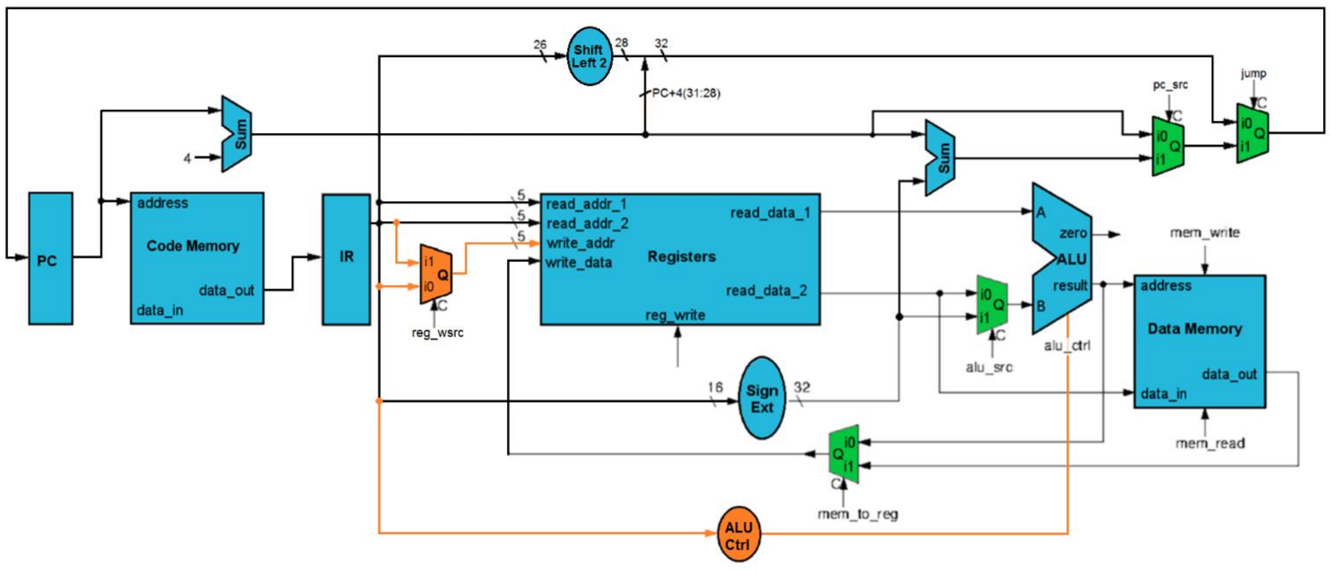
R-Type instructions are particularly complex due to the amount of information they require for operation. However, the flow of data is largely the same for each R-type instruction:

Opcode	Source 1	Source 2	Dest	Shift	Func
6	5	5	5	5	6

- Fetch something from the registers
- Do something to it
- Put it back in a register.

For this reason, the Opcode for all R-type instructions is #000000, and the **`Func`** component specifies the type R-type operation being used. It is the **`Func`** bits that determine what the ALU will do, in conjunction with the **`Shift`** bits when performing shift operations. We deal with these bits using a special ALU controller.

The resulting architecture looks like this:



PC
address
Code Memory
data_out
data_in
IR
Sum
4
26
Shift Left 2
28
32
PC+4(31:28)
5
5
5
i1
i0
Q
C
reg_wsrc
read_addr_1
read_addr_2
write_addr
write_data
Registers
read_data_1
read_data_2
reg_write
16
Sign Ext
32
Sum
pc_src
jump
A
zero
ALU
result
B
alu_ctrl
i0
i1
alu_src
mem_write
address
Data Memory
data_out
data_in
mem_read
mem_to_reg
ALU Ctrl