CENG 311

Computer Architecture Lecture Notes

Ege Özkan

December 4, 2020

Chapter 1

Introduction - October 16, 2020

1.1 Four Key Current Directions

1. Fundementally secure/reliable/safe architectuers
2. Fundementally energy-efficent and memory centric architectures
3. Fundementally low latency and predictable archiectures
4. Architectures for AI/ML, Genomics, Medicine, etc.

1.2 Transformation Hierarchy

The order travels through different hierarchical levels until it reaches the electrons. From problems to algorithms, thrugh program/language to system software to SW/HW Interface to lower hardware components.

Computer architecture was traditionally limited to SW/HW Interface and to Microarchitecture, but in the present day, computer architecture expands from algorithms to devices. This is because, to achieve the highest energy efficency and performance, one must take the expanded view therefore co-designing across the hierarchy.

This way, once can specialize most of the components for a specific domain.

1.3 Computer Architecture

Computer architecture is the science and art of designin computing platforms to achibe a set of design goals. Designing a supercomputer is different from designing a smartphone, but many fundemental principles are similiar.

The computer architecture allows better systems to be built by making computer faster, cheaper, smaller and more reliable, it enables new applications and enables better solutions to be found.

Studying computer architecutre allows one to understand why computers work the way they do.

1.3.1 Computer Architecture Today

The present day industry has entered a paradigm shift to novel architectures, as many difficult problems motivate and cause a demand for novel architectures.

Chapter 2

Computer Abstractions - October 23, 2020

Computer architecture is the science and art of selecting and interconnecting hardware components to create a computer that meets functional, performance and cost goals. Computer systems come in many forms, from general purpose personal computers to supercomputers.

A system must be *Functional* (correct), *Reliable* (continue to perform correctly.), High performance, low cost, low power/energy consumption and it must be secure. Keep in mind that the word *correct* may mean different things in different accuracy levels.

2.1 Abstractions

The computer system structure consists of the Application software at the top, the system software in the middle and the hardware in the bottom. **Hardware-software interface** handles the translation.

The compiler takes a high level compiled language (such as C), converts it into an Assembly language, and the assembler takes the Assembler code and converts it into the machine code.

A Microarchitecture is an impleentation of an ISA, there may be multiple implementations from the same ISA.

Levels of transmformation create abstractions. High-level language programmer does not really need to know what the ISA is and how a computer executes instructions. Abstractions also improve productivity, decisions made in the underlaying level does not need to be considered.

Knowledge of the lower level may improve higher level design choices of a person. [For instance, consider the fact that modulo operator is *slower* in the hardware level, when writing C code, modulo operations can be transformed into bit-shift operations, which are *faster*, improving speed.]

2.2 the Von Neumann Architecture

the Von Neumann Architecture consists of a main memory, a CPU and the interconnection between them. Within the CPU, there is a control unit, and an arithmatic/logic unit. the Von Neumann Architecture is the *traditional* structure of computers.

2.3 Main Memory

Collection of locations, each of which is capable of storing both instructions and data. Every location consists of an adress, which is used to access the location, and the contents of the location. It is similar in structure to a programming array. [Intresting to note, many emulators actually use arrays to emulate the memory of simpler machines.]

2.4 Central Processing Unit

CPU consists of multiple parts, **control unit**, **arithmatic logic unit** and **registers**. Memory is *fetched/read* to the CPU, as CPU sizes tend to be significantly smaller then Memory sizes, that is why despite CPU being much faster, memory is used to read/fetch from and *write/store* to.

2.4.1 Program Compilation as Execution

C is a compiled language, which means C code such as `float value = x[i];` would get translated to assembly instructions such as `ld r0 addr[1]`. [Many C compilers also optimize your code, so there are changes made to it. (Some even causing bugs.)]

The resulting assembly code is stored in the memory as binary machine code. These instructions are fetched from the memory to the CPU (ALU). A specialised register called **program counter** (PC), *points* to the instruction to be executed. When ALU executes an instruction, program counter is incremented.

2.4.2 Metrics

Performance is generally valued in the CPU architecture, this is not as straightforward as one assumes, however. Consider an airplane, airplane performance can be defined as passsanger capacity, cruising speed, crusining range or passangers per mph. As such, there are different *metrics* to optimize for.

For computers, **Latency** is the elapsed time to do a task, how long it takes to do a task while **Throughput** is total work done per unit time. Although they may *look* similar, they are not the same.

In this course, while discussing processor performance, the focus will primarilly be on the execution time for a single job (latency). It can be measured in different ways, **Execution time** includes all aspects, the total response time while **CPU time** is the direct time spent on processing a given job, this discounts IO time and other jobs' shares.

$$\text{Performance} = \frac{1}{\text{Execution Time}} \tag{2.1}$$

Relative performance is the performance comperasion between two computers. Saying X is n time faster than Y is the same as dividing their performances or the inverse of their execution times.

If Computer A runs a program in 10 seconds, and B in 15 seconds, A is told to be 1.5 times faster than B.

2.4.3 CPU Clock

CPU clock determines when events take place in the hardware. [a clock cycle is similar to a Minecraft Tick.] a clock period is the duration of a clock cycle. Clock period has the unit of time and Clock frequency, which is the cycles per second, has the unit of Hz.

This has the effect that the CPU Time is the:

$$\text{CPU Time} = \text{Cycle Count} \times \text{Cycle Time} \tag{2.2}$$

Where Cycle Count count of cycles needed for program to run, CPU Time is the CPU time of the program, Cycle Time is the time it takes for a single CPU cycle.

$$\text{CPU Time} = \frac{\text{Cycle Count}}{\text{Clock Rate}} \tag{2.3}$$

Where Clock Rate is the clock rate. Therefore, increasing the clock rate, or decreasing the cycle time will improve performance.

For isntance, if a program runs in 10 seconds on a computer A, which has a 2GHz clock rate, whihc means the clock cycles needed for this program in computer A is $2 \times 10^9\text{Hz} \times 10\text{s}$ Clock cycles.

If computer B runs this program in 6 seconds, but 1.2x more clock cycles, to calculate its clock rate, one just has to multiply the Clock rate above with $\frac{1.2}{6\text{s}}$, which results in the calculation Clock Rate = 4GHz

Different instructions take different amounts of time on different machines, division generally takes more time than addtion, floating point operations take longer than itneger ones, accessing memory takes more time than accessing registers. So one can assume that the number of cycles will equal the number of instructions for simplicity, but it would be incorrect.

Instruction count for a program is determined by a program, ISA and compiler, but the avarage cycles per instruction is determined by CPU hardware, if different instructions have different CPI, avarage CPI is affected by instruction mix.

Clock Cycles equal Instruction count × Cycles per instruction, whereas the CPU time depends on the product of Instruction count, CPI and Clock Cycle time, keep in mind Clock Cycle Time can be changed with $\frac{1}{\text{Clock Rate}}$.

Imagine an example where:

Computer	Cycle Time	CPI
A	250ps	2.0
B	500ps	1.2

Here, $t_a = \text{CC} \times \text{CCT} = I \times 2 \times 250$ and $t_b = I \times 1.2 \times 500$ where I is the instruction count, comparing them, $\frac{t_a}{t_b} = \frac{500I}{600I}$, shows that computer a is $\frac{6}{5}$ times faster for a program.

Another example, this time with compilers:

	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

Here the compiler writer can chose between different code generation sequences for three classes A, B and C. But which code sequence is faster, and what is the CPI for each sequence?

For the first sequence, there is a total of $1 \times 2 + 2 \times 1 + 3 \times 2 = 10$ Clock Cycles spent, for the second sequence, there is a total of $1 \times 4 + 2 \times 1 + 3 \times 1 = 9$. Since the compiler will execute them in the same computer, the clock rate is equal, and hence, the time being spent is directly related to the number of clock cycles, therefore, the second sequence is much more beneficial.

One could also calulcate the avarage CPI, this is calculated using:

$$\mathrm{CPI}_A = \frac{\text{Clock Cycles}}{\text{Instruction Count}} \tag{2.4}$$

$$= \sum_{i=1}^{n} \left(\mathrm{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}} \right) \tag{2.5}$$

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\mathrm{CPI}_i \times \text{Instruction Count}_i) \tag{2.6}$$

Where CPI_A is the weighted avarage CPI. Overall:

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}} \tag{2.7}$$

Determining the Values

CPU Execution Time can be determined by running the program.

Clock cycle time is usually published with documentation.

Instruction Count via the software tools that profile execution or by simulators.

CPI however, depends on a wide variety of design details including the memory system and the processor structure, therefore it is much harder to determine.

Computer Benchmarks are programs or set of programs used to evaluate computer performance, benchmarks allow us to make performance comparisions based on eexecution times, they can vary greatly in terms of their complexity and their usefulness, benchmarks should:

- Be representitive of the type of applications that run on the computer.
- Not be overly dependent on one or two features of a computer.

Amdahl's Law

Improving an aspect of a computr and expecting a proportional improvement in overall performance. Where n is the improvement factor:

$$T_{\text{improved}} = \frac{T_{affected}}{n} + T_{\text{unaffected}} \tag{2.8}$$

For instance if multiply accounts for 80 seconds of 100, how much improvement in multiply perfromance to get 2 times overall improvement. [ie to make 100 seconds to fall to 50 seconds]

$$50 = \frac{80}{n} + 20 \tag{2.9}$$

Must imrpove by 2.6 times.

However if multiply accounts for 80 seconds of 100 seconds and we want five times overall improvement. This is impossible, since the unaffected part also takes 20 seconds.

Summary of Performance Evaluation

Good benchmarks, such as SPEC can provide an accurate method for evaluating and comparing computer performance. Ahmdal's law provides an efficent method for determining speedup due to an enhancment, and one must make the common case fast.

Chapter 3

Instructions: Language of the Computer - October 30, 2020

It was allready established that the Compilers and Assemblers bring high-level languages to binary machine language program. The Instruction Set Architecture (ISA) consists of Instructions and Memory.

ISA's instructions are controlled by Opcodes, Addressing Modes, Data Types, Instruction Types and Fromats, Registers and Condition codes. An ISA interracts with the memory's address space, it has to deal with its addressability, alignment and virtual memory managment.

A microarchitecture is a specific implementation of an ISA under specific design constraints and goals. It is the things done in hardware without exposure to the software. These range from pipelining to voltage/frequency scaling.

3.1 MIPS Instruction Set

MIPS is a simple ISA.

3.1.1 Assembler Instructions

Add and Substract

Takes two sources and has one destination.

```
add a, b, c
```

Where `a` gets `b + c`, arithmatic operations occur **only on registers** in MIPS. `sub`, `mult` and `div` take the same form.

This is an important lesson on design, as a design principle, *simplicity favours regularity.*

Consider the code

```
f = (g + h) - (i + j)
```

Compiles to the following MIPS Assembly

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

Register Operands

Observe that the arithmetic instructions use register operands. MIPS has a 32x32-bit register file, that is, 32 register with 32-bit register size. Used for frequently accessed data, numbered 0 to 31. In Assembly, they are named `$t0` to `$t9` for temporary values and `$s0` to `$s7` for saved variables.

Smaller is faster.

In the same C code, consider that all variables are stored in `$s0` to `$s4`.

(keep in mind that above MISP codes were not assemble-able).

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Memory Operands

Main memory is used for composite data (arrays, structures, dynamic data) to apply arithmatic operations (loading and storing values), it is byte-addressed. Words are alligned in memory (its address must be a multiple of 4.) [This design decision is shared with x86, this is why sometimes stuff is padded in C in weird ways.]

A **word** is a 4-byte construct, (32-bits) that correspond to an integer. Do consider that big-endianness or little-endianness of an architecture will impact in which order a word, or an integer, will be stored in the memory in which order.

MIPS is a big-endian architecture.[Meanwhile x86 is a little-endian architecture]

```
lw d, off(b)
sw s, off(b)
```

`d` is the destination register, `b` is the base register `off` is offset value, `b + off` forms the memory address to be accessed. `lw` is the **load word** instruction, while `sw` is the **store word** instruction.

Consider the C code `g = h + A[8];`, `g` is in `$s1`, `h` in `$s2` and the base address of the `A` array is in `$s3`.

```
lw $t0, 32($s3)
add $s1, $s2, $t0
```

Observe that the $8 \times 4 = 32$ since $4i$ is the offset value for a specific index value in C. This is due to the fact words occupy 4 bytes, and they align by four too.

Consider the similar C code `A[12] = h + A[8];` which also stores the result, this assembles into:

```
lw $t0, 32($s3)
add $s1, $s2, $t0
sw $t0, 48($s3)
```

!!*!**

Since registers are significantly faster to access than memory, compiler should try to use registers for variables as much as possible. This would be correct even if register access time and memory access time was the same, [Which they aren't], since memory access requires more instructions.

Immediate Operands

Immediate instructions negate the need for using load instructions by using scalars. Consider, `addi` which is the immediate addition instruction, compared to `add`, it does not need the usage of a load. For instance, this snippet:

```
li $t0, 32
add $s1, $s1, $t0
```

Can easilly be rewritten as

```
add $s1, $s1, 32
```

Without using `li`. Which itself is an immediate operand that immediatelly loads a value to a register. This corresponds to the third design principle of MIPS, making the common case fast. Here, small constants are common, immediate operand avoids a load instruction.

Related to this, `$zero` read-only register (it cannot be overwritten) holds the value zero, adding with the zero register can be used to move values between registers for instance.

Logical Operations

Logical operations are bitwise operations used to perform boolean logic on values. These are `and`, `or`, `nor`, and their immediate equivalents `andi` and `ori`.

And is useful for masking bits. For instance, `and`ing bits in a word with a word whose certain parts are zero would mask out these parts. Likewise, `or` is useful for swapping certain bits to 1 in a word.

MIPS does not have a not instruction, instead, one can `nor` (not or) with the special `$zero` register. Since $\neg p = \neg(p \vee 0)$

Shift Operations

Another bitwise manipulation operations. `sll` and `srl` is used for left logical and right logical shifts respectivelly. Both of them fill the shifted parts with zero.

Shifting a number left i times is same as multiplying it by 2^i, and shifting a number i times right would mean dividing it by 2^i.

3.1.2 Instruction Representation

Instructions are kept as a series of high and low electronic signals. MIPS instructions are 32 bits long.

MIPS Instructions are classified into different types of instructions.

R-Type Instructions

Most well-known of these R-Types are aritmathic instructions. Shift amount refeers to the amount of shift that will be applied to the number (For `sll` and `srl`.) Function code is used to further distinguish between instructions as an extension of the OP Code.

Part	Length	Explanation
`op`	6 Bits	OP-Code
`rs`	5 Bits	Source register
`rt`	5 Bits	Source register
`rd`	5 Bits	Destination register
`shamt`	5 Bits	Shift Amount
`funct`	6 Bits	Function Code

Table 3.1: The structure of R-Type Instructions

Register numbers in the machine codes differ from their assembler ccounterparts a bit. Registers between `$t0` and `$t7` are translated to numbers 8 to 15, Registers `$t8` and `$t9` gets translated to 24 and 25 and registers between `$s0` and `$s7` are translated to 16 to 23.

[I should probably point out the instructor at this point said we should now the general instruction structure but not the register translations.]

Function field may look redundant, it exists due to the fact that all arithmatic operations are encoded with OP-Code 0, hence they are differentiated via their Function codes. [This whole deal occurs to keep the OP-Code length the same between R, I and J type operations. There are many more R type operations, instead of giving OP-Code more space OP-Code field is used to group them to certain categories, and Function field specifies the exact instruction. This also greatly simplifies the processor design, all instructions with OP-Code 0 gets sent to ALU, for instance.]

[You may point out, but Amber, doesn't all of the R type instructions go to ALU anyway? What is the point? Well, padawan, MIPS is designed to be extended, Sony PSP for instance has an VFPU in it as well, and R Type instructions on VFPU use a different OP-Code]

Assembler instructions like `add`, `sub`, `div`, `srl` and `sll` are translated to R-Type instructions.

I-Type Instructions

Part	Length	Explanation
`op`	6 Bits	OP-Code
`rs`	5 Bits	Register 1
`rt`	5 Bits	Register 2
constant/address	16 Bits	16 Bit Number.

Table 3.2: The structure of I-Type Instructions

Instructions in the I-Type are imediate arithmatic instructions, load/store instructions and branch instructions.

Meaning of registers change from instruction to instruction. The address is used to offset one of the registers.

Assembler Instructions like `addi`, `lw`, `sw` are translated to I-Type instructions.

J-Type Instructions

Part	Length	Explanation
`op`	6 Bits	OP-Code
`pseudo-address`	26 Bits	Shortened address

Table 3.3: The structure of J-Type Instructions

[These were not mentioned, but I am still putting this here for the sake of completeness, although I believe we will return to these in the future.]

J-Type instructions are reserved for Jump instructions, hence they have a large amount of space left for an address. This value, stored at the `pseudo-address` field is the shortened address of the destibation, its two least significant and four most significant bits are removed and are assumed to be the same as the current instruction's address. (Wikibooks - MIPS Assembly)

Chapter 4

Instructions: Language of the Computer (Cont'd) - November 13, 2020

This chapter directly continues the previous chapter.

4.1 Instructions for Making Decisions

```
beq rs, rt, L1
bne rs, rt, L1
j L1
```

These instructions *jump* to the label, labelled as `L1`. `beq` does this when two register values are equal, `bne` jumps when they are **not** equal, and `j` **unconditionally jumps** to the label `L1`.

`bne` is preffered over `beq`, this is because *not branching* is preffered over branching for performance reasons.

`j` insrtuction is represented via J-Type instructions, while other branches are represented via I-Type.

4.1.1 Basic Branching

Branching instructions can be used to represent conditionals and loops easilly.

Representing If-Else Conditionals Using Branching

Consider the C code below:

```
if (i==j) f = g+h;
else f = g - h;
```

Assuming `f`, `g`, `h`, `i`, `j` is stored in `$s0`, `$s1`, `$s2`, `$s3` and `$s4` respectivelly, this can be translated to:

```
bne $s3, $s4, Else
add $s0, $s1, $s2
j Exit
Else: sub $s0, $s1, $s2
Exit: ...
```

Where under the label `Exit` will exit the program properly. This same program can also be written as:

```
beq $s3, $s4, If
sub $s0, $s1, $s2
j Exit
If: add $s0, $s1, $s2
Exit: ...
```

Representing Loops Using Branching

A while loop of the form

```
while (save[i] == k)
        i += 1;
```

If `i`, `k` and the base address of save is stored in `$s3`, `$s5` and `$s6` respectivelly, this is compiled to the following series of MIPS instructions:

```
Loop: sll $t1, $s3, 2
        add $t1, $t1, $s6
        lw $t0, 0($t1)
        bne $t0, $s5, Exit
        addi $s3, $s3, 1
        j Loop
Exit: ...
```

Where shift left by two is used to multiply the index by four, the idea behind this decision is of course the elements are words, so we need to multiply the index by four to find the offset.

Conditional Set

```
slt rd, rs, rt
slti rt, rs, constant
sltu rd, rs, rt
sltui rt, rs, constant
```

The `slt` instructions sets `rd` to 1 if the value in `rs` is less than `rt`, while `slti` does the same if the value in `rs` is less than the constant.

The `sltu` and `sltui` work the same, however they assume the numbers being hold at the registers are *unsigned*. This could [or rather, *will*] be problematic, given that signed integers work with two's complement.

4.1.2 Procedure Calls

Procedure calling works thusly:

1. Place parameters in registers.
2. Transfer control to procedure.
3. Acquire storage for procedure.
4. Perform procedure's operations.
5. Place result in register for caller.
6. Return to place of call. [Return control]

Register Structure

Registers	Explanation
`$a0 - $a3`	To pass arguments
`$v0`, `$v1`	To store return values
`$ra`	Return address
`$sp`	Call stack pointer

Table 4.1: Registers used in procedure calls

Procedure Call Instructions

```
jal ProcedureLabel
jr $ra
```

To call a procedure, we use the jump and link instruction, `jal`, the jump and link instruction stores the address of the next instruction (the adress the program will return to this address) [the return address] to the `$ra`and jumps to the target address.

To return from a procedure, we use jump register instruction, `jr`, which copies the `$ra`to the program counter, which means the next instruction to be executed will be the return address, that is, we jump back to where came. (Address of the last `jal` call + 1)

Procedure Call

The caller puts the parameter values in `$a0 - $a3`, then uses `jal X` to cump to the procedure `X`, `jal` stores `PC+4` (Instructions are four bytes apart) in `$ra`.

The calee performs the calculations (executes any instructions it has) places the results in `$v0` and `$v1`, it then returns the control to the caller using `jr $ra`.

The PC (Program counter) is the specialised register containing the address of the instruction in the program being executed.

Call Stack

Any registers needed by the caller must be restored to the values that they contained before the procedure was invoked, for this reason, a stack is used, the special stack pointer `$sp`denotes the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found.

Consider the C code

```
int leaf_example (int g, int h,
 int i, int j) {
        int f;
        f = (g + h) - (i + j);
        return f;
}
```

Assume that arguments are stored in `$a0 - $a3` and consider `f` in `$s0` (hence, `$s0` must be saved on stack). The C code `f = (g + h) - (i + j);` is translated to MIPS assembly:

```
add $t0, $a0, $a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
```

Since the callee will modify `$t0`, `$t1` and `$s0`. Their previous values (if they had any) must be stored in the call stack using `$sp`This is done by starting to add values to `$sp`by subtracting four times three from it, ie, allocating space for three registers.

```
leaf_example:
        addi $sp, $sp, -12 # Allocate space.
        sw $t1, 8($sp) # Save stack.
        sw $t0, 4($sp)
        sw $s0, 0($sp)
        add $t0, $a0, $a1
        add $t1, $a2, $a3
        sub $s0, $t0, $t1
        add $v0, $s0, $zero # Store result.
        lw $t1, 8($sp) # Restore values
        lw $t0, 4($sp)
        lw $s0, 0($sp)
        addi $sp, $sp, 12
        jr $ra # Return
```

The above examples shows the complete MIPS translation of the `leaf_example` function in C as a MIPS procedure.

[Stack pointer does not run out of space, because it is located in the top of the memory. It *grows* and shrinks usable memory more calls you made to a function like this.]

Convention

This is not actually how most of the time MIPS programmers do things, instead, we assume that, across a procedure call, the following registers are preserved:

1. Saved registers, `$s0-$s7`
2. The stack pointer register `$sp`
3. The return address `$ra`
4. Stack above the stack pointer, [ie, previous call].

And the following isn't preserved:

1. Temporary, `$t0-$t7`
2. The argument registers `$a0-$a3`
3. The return registers `$v0`, `$v2`

4. Stack below the stack pointer, [ie, next calls, procedure calls made inside the procedure call].

Using the convention, above code simplifies to:

```
leaf_example:
        addi $sp, $sp, -4 # Allocate space.
        sw $s0, 0($sp) # Store saved.
        add $t0, $a0, $a1
        add $t1, $a2, $a3
        sub $s0, $t0, $t1
        add $v0, $s0, $zero # Store result.
        lw $s0, 0($sp)
        addi $sp, $sp, 2
        jr $ra # Return
```

If we haven't used the saved register `$s0`, then we didn't needed to save and restore it at all.

(From November 20, 2020)

Nested Procedures

Procedures that call other procedures, for nested call, caller needs to save its return address, and any arguements and temporaries needed after the call to the stack, and restore them from the stack after the call.

Consider the C code:

```
int fact (int n) {
        if (n < 1) return 1;
        else return n * fact(n - 1);
}
```

This is a very simple (and well-know, recursive) factorial algorithm, with the argument `n` being stored in `$a0` and result in `$v0`, this can be compiled to the following MIPS assembly code:

```
fact:
        slti $t0, $a0, 1
        beq $t0, $zero, L1
        addi $v0, $zero, 1
        jr $ra # and return
L1:
        addi $sp, $sp, -8
        sw $ra, 4($sp)
        sw $a0, 0($sp)
        addi $a0, $a0, -1
        jal fact
        lw $a0, 0($sp)
        lw $ra, 4($sp)
        addi $sp, $sp, 8
        mul $v0, $a0, $v0
        jr $ra
```

Here, in the recursive part, we need to store the return address and the `$a0` in the stack to recover them later. We recover these values after the recursive call `jal fact`.

This is because, of course, if the values of `$a0` gets lost, then we will have significant problems, the code will not work correctly, and we do need to overwrite `$a0`, since the function we are calling, the function we are already in, expects its input value in `$a0`.

Original `$a0` may not be always have to stored however, since in certain recursive function we may not need to use the previous inputs, it is fine to overwrite them then.

Keep in mind there is a certain point that may arise if one uses the stack too much, if the stack size exceeds that of the memory, a stack overflow will occur.

[A similar problem occured in eraly Sinclair ZX80 computers, the screen memory was shared with the RAM, and if your program consumed too much memory, the display would literally shrink. It was reportedly not amusing to use Sinclair ZX 80 due to this reason, well, one of the reasons.]

Chapter 5

Instructions: Language of the Computer (Cont‘d) - November 20, 2020

5.1 Representing Instructions

Kept as a series of high and low electronic signals (binary) and represented as numbers. MIPS instructions are 32 bits long.

We had already gone over the I-Type and R-Type instructions extensivly. It should be noted that I-Type instructions also include the branching instructions, which does make sense, since they require two registers to compare and an address to jump to.

J-Type Instructions

Part	Length	Explanation
`op`	6 Bits	OP-Code
`pseudo-address`	26 Bits	Shortened address

Table 5.1: The structure of J-Type Instructions

J-Type instructions are reserved for `j` and `jal` unconditional jump instructions. Observe that the Address can be any unsigned number up to 2^{28}, compare to I-Type's 2^{16}

5.2 Addressing Modes

There are different sorts of addressing modes used in MIPS.

5.2.1 Immediate Addressing

The operand is a constant withing the instruction itself. This is used in I-Type instructions with a constant, such as in `addi`. This value is a 16-bit value.

32-Bit Immediate Operands

But, let us say we want to load a very big 32-bit number such as 4000000, this is done via the `lui rt, constant` instruction, which sets the first 16 bits (upper part) of the register `rt` to the `constant` and sets the remaining part to the zero.

```
lui $s0, 61
ori $s0, $s0, 2304
```

We **load** the upper 16 bits of the value to the first 16 bits of the `$s0` register using the Load Upper Immediate instruction, and then, we use the Logical Or Immediate operation to simply mask the remaining lower 16 bits (but signed, so can be negative) of the value to the register.

5.2.2 Register Addressing

Register addressing simply refeers to putting the number of the register to the register fields of the I-Type and R-Type instructions.

5.2.3 Base Addressing

The operand is at the memory location whose address is the sum of a register and a constant in the instruction, such as in, do realise that the offset might be negative as well:

```
lw $t0, 1200($t1)
```

Which loads an integer from the 1200 bytes after the address at `$t1`. (`1200 + [$t1]`).

5.2.4 PC-Relative Addressing

The branch address is the sum of the PC and a constant in the instruction. `PC + offset x 4`

```
beq rs, rt, L1
```

This is a relative addressing mode, this way, the amount of memory usable by the program increases substentially.

The multiplication by four part refeers to the fact that an instruction occupies four bytes (32-bits) in the memory.

5.2.5 Pseudo-direct Addressing

The jump address is the 26 bits of the instruction concatenated (after being multiplied by four) with the upper bits of the PC.

```
j L1
```

Here the 26 bits represent the 28 lower bits of the address (it will be multiplied by the four, or shifted by two) and are concanated with the upper four bits of the program counter.

For instance, the following MIPS assembler instructions,

```
Loop: sll $t1, $s3, 2
      add $t1, $t1, $s6
      lw $t0, 0($t1)
      bne $t0, $s5, Exit
      addi $s3, $s3, 1
      j Loop
Exit: ...
```

Compiles to these machine instructions:

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				

5.2.6 Branching Far Away

If the branching target is too far to encode with 16-bit offset, assembler rewrites the code:

```
beq $s0, $s1, L1
```

Becomes:

```
bne $s0, $s1, L2
j L1
L2:
        ...
```

Chapter 6

Instructions: Language of the Computer (Cont‘d) - November 27, 2020

6.1 Byte & Halfword Operations

```
lb rt, offset(rs)
lh rt, offset(rs)
```

For loading and storing bytes and halfwords, `lb`, `lh`, `lbu`, `lhu`; `sb` and `sh` can be used to get a single byte/halfword from the memory.

`lb` and `lh` sign extends the value to 32 bits in `rt`, their unsigned counterparts `lbu`, `lhu` on the other hand, zero-extends the value to 32 bits.

For instance, if we have `F0` in the memory, `lb` would load `FFFFFFF0`, but `lbu` would load `000000F0`.

```
void strcpy (char x[], char y[]) {
        int i;
        i = 0;
        while ((x[i]=y[i])!='\0')
                i += 1;
}
```

Assuming the addresses in `x, y` is in `$a0` and `$a1`. `i` in `$s0`.

```
void strcpy (char x[], char y[]) {
        int i;
        i = 0;
        while ((x[i]=y[i])!='\0')
                i += 1;
}
```

```
strcpy:
        addi $sp, $sp, -4
        sw $s0, 0($sp)
        add $s0, $zero, $zero
L1:
        add $t1, $s0, $a1
        lbu $t2, 0($t1)
        add $t3, $s0, $a0
        sb $t2, 0($t3)
        beq $t2, $zero, L2
        addi $s0, $s0, 1
        j L1
L2:
        lw $s0, 0($sp)
        addi $sp, $sp, 4
        jr $ra
```

6.2 Translation and Starting a Program

6.2.1 Compiler

Compiler transforms higher level programming languages into assembly language program, a symbolic form of what the machine understands. [Some Compilers translate directly to machine code too.]

6.2.2 Assembler

Assembler converts Assembly instructions to machine code. Most of the time, machine instructions have a one-to-one relationship with machine instructions. **Pseudoinstructions** are instructions that do not have implementations, but simplify translation and programming (in Assembly) For instance `blt` pseudoinstruction assembles into a `slt` and `bne` instruction.

The `$at` register is used as the assembler temporary to write this pseudoinstructions.

Assembler turns the program into an Object file that includes machine language instructions, data, and information needed to place instructions properly in memory.

6.2.3 Object File

The object file includes:

Object File Header The size and position of the other pieces of the object file.

Text Segment The machine language code.

Static Data Segment Data allocated for the life of the program.

Relocation information Instructions and data words that depend on absolute addresses.

Symbol Table Global definitions and external references (labels)

Debug Information Associates machine instructions with C source files. [In `gcc`, you can enable this via the `-g3` switch.]

6.2.4 Linker

Places code and data modules symbolically in memory. Determines the addresses of data and instruction labels (by using relocation info) patches the internal and external references and produces executable file.

6.2.5 Memory Layout

In MIPS architecture, on top of the reserved segment sits the text segment, pointed to by the program counter, on top of which sits the **static data**, pointed by the `$gp`, on top of which sits the **dynamic data**, the heap. and on top sits the stack, pointed by the `$sp`.

Text segment and data segment sits in the object file, stack and dynamic data provided by the system itself. [In MS-DOS MASM x86, you actually had to tell the system the size of the stack you wished.] Keep in mind that the stack grows *downwards*, towards the dynamic data segment.

6.2.6 Loader

This is the part of the computer that actually copies the instructions and date from the executable file into memory. Copies the parameters to the main program, initializes the machine registers and sets the stack pointer to the first free locations, jumps to the start up routine that copies the parameters into the argument registers and calls the main routine.

6.2.7 Dynamic Linking

With Static linking, the library routines become part of the executable, loads all routines in the library even if not used, dynamically linked libraries (DLLs) are linked during execution.

6.3 Compiler Optimization

Many compilers do not just generate assembler code, in the background, certain optimizations are done to speed up code.

gcc Optimization Levels

`gcc`, the de facto standard GNU/Linux compiler provides six different optimization levels.

Switch	Optimization Level
`-O0`	Optimize for compile time (default)
`-O1`, `-O`	first level of optimization size and execution time.
`-O2`	Second level of optimization size and execution time.
`-O3`	Third level of optimization for code size and execution time.
`-Os`	Optimize for code size
`-Ofast`	Optimizes even more than `-O3` but math is not IEEE compliant.

Table 6.1: Different optimization levels of GNU C Compiler.

As can be seen in Table 6.1, there are different levels of optimization in `gcc`, these may increase the compilation time and memory usage, but in general will decrease the execution time. `-Ofast` allows for fast mathematical operations, but it is not completely IEEE compliant.

6.3.1 Compiler Optimizations

Here are certain optimizations performed by the compiler.

Register Allocation

Given a block of code, the compiler wants to choose assignments of variables to register to minimize the total number of required registers.

```
w = a + b;
x = c + w;
y = c + d;
```

Here the naive register allocation would result in assigning seven registers for the seven variables, but this is not actually necessary, there is a better way.

Dead Code Elimination

Code that will never be executed can be safely removed from the program.

Loop Transformations

For the processors, loops come with overhead, therefore, the compilers try to limit them.

First loop transformation is **loop unrolling.**

```
for (i = 0; i < N; i++) {
        a[i] = b[i] * c[i];
}
```

If the `N = 4`, then it can be unrolled four times.

```
a[0] = b[0] * c[0];
a[1] = b[1] * c[1];
a[2] = b[2] * c[2];
a[3] = b[3] * c[3];
```

Or it may be unrolled "twice", reducing the times we check the condition.:

```
for (i = 0; i < 2; i++) {
        a[i*2] = b[i*2] * c[i*2];
        a[i*2 + 1] = b[i*2 + 1] * c[i*2 + 1];

}
```

Another transformation is **loop fusion**, which combines two or more loops into a single loop.

```
for (i = 0; i < 300; i++) {
        a[i] = a[i] + 3;
}

for (i = 0; i < 300; i++) {
        b[i] = b[i] + 3;
}
```

is transferred to:

```
for (i = 0; i < 300; i++) {
        a[i] = a[i] + 3;
        b[i] = b[i] + 3;
}
```

Another transformation is **loop tiling**, which breaks up a loop into a set of nested procedures that operate on a subset of data.

Procedure Inlining

The body of the procedure is substituted in place for the procedure call to eliminate the call overhead.

This however, increases the code size. [CPython does not do this, it is one of the reasons it has abysmal performance compared to PyPy.]

6.3.2 Performance Optimization

While optimizing, it is pivotal to optimize at multiple levels, taking into account the data representation, algorithm, procedures and loops.

One must also understand the system to optimize performance, how programs are compiled and executed, how modern processors and memory systems operate, how to measure program performance and identify bottlenecks, how to improve performance without destroying code modularity and generality.

Chapter 7

Arithmetic Operations - December 4, 2020

7.1 Representing Numbers

Using 32 bits, it is possible to represents unsigned numbers between zero and $2^{32} - 1$, for signed numbers, this number is $2^{31} - 1$

7.1.1 Unsigned Integers

While representing unsigned numbers of $n - 1$ numbers, the value of a number x can easily be calculated via:

$$x = \sum_{k=0}^{n-1} x_k 2^k \tag{7.1}$$

Where x_k represents the digits of the number, and is either one or zero.

And n bit unsigned integer can represent values between zero and $2^n - 1$

7.1.2 Signed Integers

The two's complement system is used in the signed integers, where the least significant bit is the sign bit, one for negative and zero for positive numbers. The value of a number x of n bits length can be calculated using:

$$x = -x_{m-1} 2^{n-1} + \sum_{k=0}^{n-2} x_k 2^k \tag{7.2}$$

In practice, one can convert a twos complement to a normal number by inverting its bits and adding one and multiplying via -1.

A number signed number x of n bits long can represent any number between 2^{n-1} to $2^{n-1} - 1$.

7.2 Arithmetic Operations

Processors can perform many arithmetic operations.

7.2.1 Integer Addition

Integer additions use a carry, and when adding two numbers of the same sign, an overflow may occur. If two negative numbers are being added, if the first bit ends up zero, (or one if both are positive) the overflow is said to have occurred.

7.2.2 Integer Subtraction

This is the same as adding the negation of the second number to the first number, subtracting a positive number from a negative operand is

Operation	Left	Right	Result
Addition	+	+	−
Addition	−	−	+
Subtraction	−	+	+
Subtraction	+	−	−

Table 7.1: Overflow conditions for subtraction and addition

said to have resulted in an overflow if the result sign is zero. (Or reverse in the reverse case).

7.2.3 Dealign with Overflows

The `addu`, `addiu` and `subu` ignores overflows and do not cause exceptions, on the other hand `add`, `addi` and `sub` cause an exception, when an overflow occurs, the PC is saved in the `EPC` register, predefined handler address is jumped and `mfc0` instruction can retrieve `EPC` value to return after corrective action.

7.2.4 Multiplication

One can just place a copy of the multiplicand in the proper place if the multiplier digit is 1, or zero if the multiplier digit is zero.

The *proper place* can be calculated using the shift operations. The multiplication operation uses shift and sum operations to achieve its goal in the hardware level.

As one can see, there is redundancy in the multiplication, half of the bits are spent for zeros, three steps per bit is used. And also the least significant bit is updated only once.

An improved version of the multiplication hardware performs steps in parallel, executing two steps per bit.

Signed Multiplication

Signed multiplication acts as if the numbers are positive, and remembers the original signs, converting to the appropriate sign in the end.

Multiplication in MIPS

MIPS provides two 32-bit registers for product, `HI` and `LO`.

mult rs , rt
multu rs , rt
mfhi rd
mflo rd

`mult` and `multu` multiply `rs` and `rd` and stored the 64-bit product in `LO` and `HI`, the special move instructions `mfhi` (move from `LO`) and `mflo`, (move from `HI`) can be used to get the values in these special registers.

The simpler `mul rd, rs, rt` places the least significant 32 bits of product to `rd` register.

7.2.5 Divison

n bit operands yield n bit quotient and remainder. It checks for divisor to be smaller than the divident bits, one bit in quotient, subtract otherwise 0 bit in quotient bring down next dividend bit. This is called the **long divison** approach.

Signed division divide using absolute values and then adjust signs as needed.

Similar to multiplication, MIPS divison is done with a `div` instruction that uses `HI` and `LO`.

Keep in mind that shift operations are significantly better.

Field	64-Bit	32-Bit
Sign	1 Bit	1 Bit
Exponent	11 Bits	8 Bits
Fraction	52 Bits	23 Bits

Table 7.2: Fields of single and double-precision floating point numerals.

7.3 Floating Point

Floating point is [used to represent] numbers with fractions, reals. It can be written in scientific notation, or normalized notation. Normalized notation is number in scientific notation that has no leading 0s.

The floating point takes the following form: Where the whole number x is:

$$x = (-1)^s \cdot (1 + F) \cdot 2^{E-B} \qquad (7.3)$$

Where B is the bias. Significant is normalized, that is, it always has a leading pre-binary-point 1 bit, so no need to represent it explicitly, exponent (E) is the actual representation plus a bias (B).

The bias of a single precision floating point number is 127.

[Other than double precision (64 bits) and single precision (32 bits), there is also extended precision (40 bits) and also extended double precision (80 bits)]

[Early computers and chipsets actually had dedicated Floating Point coprocessors. Intel x86 even had something called an x87, a subset of x86 working with the Intel 8087 Floating Point Coprocessors. Nowadays, x87 is a proper part of x86.]