**INSTRUCTION SET ARCHITECTURE (ISA)**

Diagram

Description automatically generated



3 levels we will study (not I/O system)

Also we will understand better how a compiler works

Software is easily convertible to ISA, ISA is easy to be executed by designing its hardware

* By looking at ISA, you can easily design the hardware
* By looking at software, you can easily convert it to ISA

Instruction Set

The repertoire of instructions of a computer

Different computers have different instruction sets

* But with many aspects in common

Early computers had very simple instruction sets

* Simplified implementation

Many modern computers also have simple instruction sets

Any high-level language program must be convertible to this instruction set

Couple of instructions (10-20) is enough to design a CPU which can handle any high-level language program

You can optimize RISC processor better than CISC processors. CISC processors’ initial hardware is already complex enough.

CISC 🡪 INTEL , RISC 🡪 ARM

The MIPS Instruction Set

Used as the example throughout the book

Stanford MIPS commercialized by MIPS Technologies ([www.mips.com](http://www.mips.com))

Large share of embedded core market

* Applications in consumer electronics, network/storage equipment, cameras, printers, …

Typical of many modern ISAs

* See MIPS Reference Data tear-out card, and Appendixes B and E

Mnemonic 🡪 text representation of instructions (such as ADD)

Generic Examples of Instruction Format Widths

Diagram

Description automatically generated

Width: how many bytes will be used in 1 instruction

Variable: Different instructions may have different # of bytes

Fixed: All instructions must have same # of bytes

Hybrid: Different widths for instructions but not each instruction has a different width. For example there are 3 widths and each instruction must use one of these 3 widths.

Variable ve Hybrid widthte bir sonraki instructionın nerede olduğunu bilmek için processor ekstra yük harcayacak. Yani Fixed width bu konuda avantajlı, program counter (sıradaki instructionın yerini gösteren register) her defasında belirli sayıda (örneğin 4 byte) artacak.

Variable ve Hybrid width, memory konusundan avantajlı.

Assume you want to design an instruction that puts number inside register. Inside the instruction we must see the number. If the register is 32 bits and if instruction is 32 bit, then you cant give 32 bit number inside instruction bc there is no space for 32 bit number bc inside the instruction there must be different parts by which CPU can understand the type of the instruction. It is called opcode. Each instruction must have an opcode field so that this opcode field is analysed by the processor and processor understand which instruction it is. So decision you make about width, directly affects your resultant instruction set.

In MIPS, it is fixed width (32 bits).

Basic ISA Classes

Memory to memory machines

🡪 You get an operand from the memory and put the result back to memory

Memory is combination of lots of registers

* we need storage for temporaries
* memory is slow
* memory is big (lots of address bits)

Architectural registers

🡪 Located inside the CPU architecture, they are inside the core, inside the datapath

Cache is inside the same chip but it doesn’t mean that it is part of the CPU architecture

There are 32 registers in MIPS, each register is generally 32 bit

* registers can hold temporary variables
* registers are faster than memory
* memory traffic is reduced, so program is speed up (since registers are faster than memory)
* code density improves (since register named with fewer bits than memory location)

Microcontroller, microprocessorlardan daha basit işler için tasarlanmış processorlerdir. Their purposes are different.

Microprocessors are much faster. They are designed only for better execution of the instruction set. The only aim of microprocessor is speed and throughput in executing the instructions. So they are optimized for that purpose. They don’t include any peripherals (çevresel birimler). Büyük kısmı cacheden oluşur. Datapath çok gelişmiştir. Yönetilmeleri OS olmadan oldukça zordur. You need external memory (main memory) for microprocessor.

Microcontroller içinde peripherallar da bulunur. You don’t need external memory for microcontroller. Microcontroller chip içerisinde ARM core/cortex (only part related with the processor), instruction and data memory, UART and SPI (modules that provide communication with external devices), voltage regulator, analog to digital converter, … bulunur.

On microprocessor we only have much complex ARM core/cortex part with much higher clock rates.

Why don’t we prefer memory to memory instructions for our microprocessors while we use it in microcontrollers?

Because memory slow and microprocessors must execute at very high speeds. For that purpose we use registers.

Get data from the memory to register (currently needed ones). Operate them. At the end put them back to memory.

Scenarios for Communication with Memory

Stack (not a register file but an operand stack), memory to memory access

* 0 address add tos = tos + next

Accumulator (1 register), take operand from memory take 2nd operand from the accumulator perform the operation and put result back to accumulator instead of putting back to memory. As you have only 1 register, at each change of variable, you have to write your accumulator content to the memory so it requires lots of memory accesses, not as many as stack.

* 1 address add A acc = acc + mem[A]

General Purpose Register File (Register-Memory), one operand from memory, one from register. Put result back to register.

* 2 address add A B EA(A) = EA(A) + EA(B)
* 3 address add A B C EA(A) = EA(B) + EA(C)

General Purpose Register File (Load/Store), case for our MIPS and most of the other RISC processors. You perform all operations only for these registers. You never get data from memory and perform an operation. There are special instructions to access to memory. Their only task is to get data from memory and put it to register or vice versa.

* 3 address add Ra Rb Rc Ra = Rb + Rc

load Ra Rb Ra = mem[Rb]

store Ra Rb mem[Rb] = Ra

load and store are the only instructions that don’t work on registers. So all other instructions are much faster. They reach to the memory.

Comparison:

* Bytes per instruction?
* Number of instructions?
* Cycles per instruction?

Comparing Number of Instructions

Table

Description automatically generated with low confidence

Add in stack automatically adds last 2 numbers. Pop is to get data from stack.

R olmayanlar memory adresi.

Microcontroller gibi daha düşük hızdaki architecturelar için register kullanılmayabilir, bu daha avantajlı olabilir (donanım rahatlar).

Chart

Description automatically generated with low confidenceMIPS Architecture – Registers

The MIPS architecture is considered to be a typical RISC architecture

* Simplified instruction set 🡪 easier to study
* Most new machines are RISC

Programmable storage

* 31 x 32-bit GPRs (r0 = 0)
* special purpose – HI, LO, PC
  + HI: most significant 32 bit of the result
  + LO: least significant 32 bit of the result
  + PC: program counter, stores the next instruction address
* 32 x 32-bit FP regs
* 232 x bytes of memory

Each register has 32 bit content.

HI and LO are used for multiplication and division instructions.

f registers are general purpose floating point registers.  
r registers are general purpose integer registers.

Register Names in MIPS Assembly Language

With MIPS, there is a convention for mapping register names into general purpose register numbers.

Table

Description automatically generated

* $v0-$v1 🡪 for returning the return arguments of a function
* $a0-$a3 🡪 for arguments passing to a function
* saved and temporary registers are for any purpose. But if a function wants to change the content of the s registers, it must save them to stack (to memory) and then at the end of the function it must restore them so that the calling function can find the contents unchanged. This is an overhead for s registers.
* $gp 🡪 shows the middle point of the data memory part of your memory
* $sp 🡪 used to record the address of the stack for last field position
* $fp 🡪 start of the stack for a function
  + All function variables are stored at addresses between the fp and sp
* $ra 🡪 holds the program counter’s address when a function is first called, we store return addresses of the functions in short.

4 a register var, fonksiyon 4’ten fazla argüman alırsa ne olur? Diğer registerları kullanabilirsin ama kontrata uymak ve programlama açısından en doğru yöntem kendi stackinin (memorydeki stack, stack pointerın son yeri gösterdiği) içerisine variablelarını yerleştirmek olur. Fonksiyon stack pointerı yani yüklediğinde en son kalan yeri biliyor. Bu adresten yola çıkarak geriye doğru kaç tane variableın varsa okuyor. Yani çok fazla variable varsa stack kullanılır.

You can use t0 instead of a0 but you don’t obey to contract and it will be hard to track.

Always put $ to register name, it means content of s0 🡪 $s0

Always supply 3 register addresses (for MIPS architecture)

MIPS Arithmetic

Arithmetic instructions have 3 operands

* Two sources and one destination
  + add a, b, c #a gets b+c

Operand order is fixed (destination first)

* Example:
  + C code: A = B + C
  + MIPS code: add $s0, $s1, $s2 -------> this is an R type instruction

(associated with variables by compiler)

s0 is address, $s0 content at s0

You can supply 3 same register address.

Each address is 5 bit (we have 32 registers – 25 = 32)

* content of a register is 32 bit number
* address of a register is 5 bit number

There are 3 types of instructions in MIPS:

* R type
  + We supply 3 different register addresses to instruction and instruction performs an operation between the last 2 register contents and put the result back to the first register
  + R type instructionlar registerlar arası işlem yapar ve sonucu registera yazar
* I type
* J type

Design principle: Simplicity favours regularity

* Regularity makes implementation simpler
* Simplicity enables higher performance at lower cost

Of course this complicates some things:

A picture containing calendar

Description automatically generated

Instead of $t0, you can use $s0. You could also use t registers for all of them.

Operands must be registers, only 32 registers provided

Design Principle: Smaller is faster.

*🡪 Each instruction is 32 bit (fixed MIPS). 🡨*

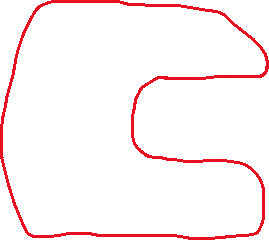
With 32 bit instruction, I can only give 3 register addresses (which requires 15 bits). Other parts are used for other purposes.

Big Picture

Diagram

Description automatically generated

MEMORY



assembly language

We put current instruction at instruction register. DATAPATH

MIPS R-Type Instructions

instr $rd, $rs, $rt



Assembler basically looks at an instruction and if it is an R-type, assembler converts it using this structure.

Instruction fields:

* op: operation code (opcode)
  + gives idea about the type of the instruction or which instruction has come
* rs: first source register number
* rt: second source register number
* rd: destination register number
* shamt: shift amount (00000 for now)
  + only used for shift instructions
  + 5 bits bc at most we want to shift number by 31 bits, we don’t want to shift it more than 31 bit bc the total size of the number is 32 bit
* funct: function code (extends opcode)
  + least significant 6 bit of the instruction
  + used for same purpose as op
  + Control unit, ALU’ya vereceği talimatları funct fielddan anlar.

For all R-type instructions opcode is 0. How can CPU differentiate different R-Type instructions? It understand that by looking at the funct field. This field is specific to the instruction for an R-Type instruction.

R-Type instructionların 2 register arasında gerçekleştirdiği operasyonun cinsi haricinde yaptıkları her şey (datapathte yapılacak işlemler, sonucun götürülüp registera yazılması vs.) ortak. Onun için control unit “bana instructionın R-Type olduğunu söylemen yeterli” diyor. R-Type “ben toplama mı çıkarma mı and mi or mu yapacağım söylemezsem sen bunu nasıl yapacaksın?” diyor. Control unit de “onu bana söyleme, benim R-Type olduğunu bilmem yeterli, hangi işlemin yapılacağı ALU’ya söylenecek olan şey” diyor. Instructionın R-Type olduğu opcode’dan anlaşılır, add mı subtract mı olduğu funct fielddan anlaşılır.

Opcodeları ezberlemeye gerek yok.

6 bit opcode, sahip olduğumuz instructionlardan fazlasını ifade edebilecek kapasitededir. 64 farklı instruction yok ancak processor geliştirilmeye açık tutulur.

R-Type Example

Table

Description automatically generated

add is an R-Type instruction bc it supplies 3 register addresses

Ek bir donanım yoksa (shamt 0 değilse ve instruction da shift değilse exception veririm tarzı) shamt 0 da olsa 1 de olsa CPU onu ignore edecekti çünkü instructionın shift olmadığı add’den anlaşılır. Add instruction shamta hiç bakmaz, sadece shift instruction bakar. Yine de instruction shift değilse 0 yazmak en sağlıklısı.

Assembler takes the instruction by parsing your assembly text file (assembly program), it takes instructions one by one and it converts them to binary number and stores this number in the instruction memory.

Hexadecimal

Base 16

* Compact representation of bit strings
* 4 bits per hex digit

Table

Description automatically generated

eca8 6420 = 1110 1100 1010 1000 0110 0100 0010 0000

Conversion between binary-hexadecimal is very easy. e is 1110, c is 1100, etc.

You can see in your computer when there is an error sth like this “there is an error at 0x0912…”.

* 0x means that this is an hexadecimal number
* 0912… is the address of the memory location

64 bit memory address (we use in today’s technology) is too long for binary so we use hexadecimal.

In order to represent 32 bit or 64 bit numbers, mostly hexadecimal is used.

We represent machine codes, data as hexadecimal. If we use AND, OR, etc. we need to use binary representation bc we AND or OR corresponding bits.

Registers vs. Memory

Registers are faster to access than memory

Arithmetic instructions operands must be registers,

* only 32 registers provided

Operating on memory data requires loads and stores

* more instructions to be executed

Compiler associates variables with registers

* Variables are both on memory and registers, they are operated on registers

What about programs with lots of variables?

* Registers may not be enough
* For example t0 belongs to variable x. Then if you want to use t0 for variable y, before using t0 register for y you first put the value inside t0 register to the corresponding memory location so that you don’t lose the value of x. For that purpose, you should use store word instruction which puts the register content to memory.

Diagram

Description automatically generated

Memory Organization

Viewed as a large, single-dimension array, with an address

A memory address is an index into the array

Byte addressing means that the index points to a byte of memory

Table

Description automatically generated

Each byte in memory has different address.

Store word 🡪 stores a word to a memory.

Word for MIPS is 4 bytes (32 bits). In modern processors, it is 8 bytes (64 bits).

When you say int in C, MIPS understands that you mean 1 word (4 bytes).

All instructions use word in MIPS.

If I only have store word and load word instructions, then I only need addresses for 1 word, not for individual bytes inside these words. But it is not the case, we have other instructions like load byte. It takes a single byte from memory and puts it into register. Store byte instruction takes a single byte from the register and it writes to memory. So we need to have individual address for each byte inside our memory.

Bytes are nice, but most data items use larger “words”

For MIPS, a word is 32 bits or 4 bytes.

Table

Description automatically generated

232 bytes with byte addresses from 0 to 232-1

230 words with byte addresses 0, 4, 8, … 232-4

Words are aligned

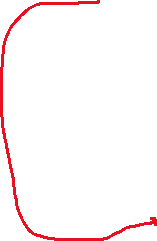
* i.e., what are the least 2 significant bits of a word address?

Chart

Description automatically generated



Alignment: require that objects fall on address that is multiple of their size



Aligned memory: You are sure that words are represented like it is showed in green

Not aligned memory: Words can be cut at different positions. You may have 3 bytes of word at some location and 1 remaining byte can be put to the next location because of the overflow. There must be some previous data here.

Memory’ye diyelim 4 bytelık worde 1 byte yazdık (store byte ile). 3 byte boş kaldı. Diyelim memory’ye integer koyacaksın:

* Aligned ise 3 byte boş kalır, sonraki wordün 4 byte’ı kullanılır.
* Not aligned ise 3 byte kullanılır, sonraki wordün de 1 byte’ı kullanılır.
* ---
* Not alignedın dezavantajı wordü okumanın zor olması olur. Wordün ne kadarı burada, nereden itibaren okuyacağız, kalan kısmı ne kadar vs. anlamak için CPU’da ekstra donanım çalışması lazım. O yüzden MIPS aligned memory kullanır.
* Alignedın dezavantajı ise yer kaybıdır.

not aligned memory

aligned memory

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

yeşil: koyacağımız int

kırmızı: önceden yazılan 1 byte (örneğin char)

Aligned kullanıp sonraki char geldiğinde yine önceki charın yanına yazarak yerden tasarruf sağlayabiliriz.

Günümüz compilerları bu tarz tasarrufları uyguluyorlar.

Addressing Objects: Endianess and Alignment

Individual bytes have different addresses.

It can be:

0 | 1 | 2 | 3

Table

Description automatically generated   
 Big Endian   
 Address of word is address of most significant byte

---OR---

3 | 2 | 1 | 0

Table

Description automatically generated  
Little Endian  
Address of word is address of least significant byte

There is no advantage or disadvantage.

We use Big Endian in MIPS.

Big Endian: address of most significant byte = word address (xx00 = Big End of word)

* IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

Little Endian: address of least significant byte = word address (xx00 = Little End of word)

* Intel 80x86, DEC Vax, DEC Alpha

Chart, calendar

Description automatically generated

load word veya store word adresleri 4’ün katı olmalı.

Memory Operand Example 1

A picture containing text

Description automatically generated

offset shows which element you want to access to

lw (takes word from memory and puts it into register):

* content of s3 (base address of array) is summed up with 32 and we get a memory address. Then content of t0 is loaded with the content into that address in memory.

For arrays, we use memory, not registers.

offset is always constant

Memory Operand Example 2

Text

Description automatically generated

MIPS I-TYPE INSTRUCTION

Table

Description automatically generated

lw and sw 🡪 I-Type instructions bc in R-type we give 3 register addresses but in lw and sw we give 2 register addresses and 1 offset. This offset must be inside our instruction. This is why we cant use R-type instruction bc in R-type instruction there is no location for offset. In I-type instruction, least significant 16 bit is used for this offset.

16-bit 🡪 for 2s complement

* 215-1 🡪 biggest immed
* -215  🡪 smallest immed

Each I-type has different opcodes.

Our instructions are 32-bit fixed.

MIPS

* Loading words but addressing bytes
* Arithmetic on registers only

A picture containing text

Description automatically generated

Constants, i.e. Immediates

Small constants are used quite frequently (50% of operands)

* e.g., A = A + 5;

B = B + 1;

C = C – 18;

MIPS instructions (I-Type):

A picture containing calendar

Description automatically generated

Immediate (I-Type) Arithmetic

Graphical user interface, application, table

Description automatically generated

We don’t have subi for subtracting immediately bc you can give negative values for immed field.

MIPS Arithmetic Instructions

A picture containing table

Description automatically generated

Exception is overflow

add unsigned is very same with add but it doesn’t detect overflow

divide also gives mod of the numbers in Hi register

You can write “add $zero, $t0, $t1” but hardware doesn’t allow.

*Her instruction için MIPS’te donanım vardır.*

Branch Instruction Design

Hardware for <, , … slower tha = ,

* Combining with branch involves more work per instruction, requiring a slower clock
* All instructions penalized

beq and bne are the common case, we also have slt

This is good design compromise

Multipliy / Divide

Graphical user interface

Description automatically generated with medium confidence

Logical Operations

Table

Description automatically generated

Shift Operations (R-Type)

Chart, box and whisker chart

Description automatically generated with medium confidence

shamt: how many positions to shift

Shift left logical

* shift left and fill with 0 bits
* sll by i bits multiplies by 2i
* EX ------> sll $t0, $t0, 5

Shift right logical

* Shift right and fill with 0 bits
* srl by i bits divides by 2i (unsigned only)

Similar to I-type instructions it takes 2 registers and 1 constant but this is R-type. Constant cannot be bigger than 31 bc if you shift more than 31, number is lost. It is same as writing all 0s.

AND Operations

Useful to mask bits in a word

* Select some bits, clear others to 0

Table

Description automatically generated with medium confidence

OR Operations

Useful to include bits in a word

* Set some bits to 1, leave others unchanged

Table

Description automatically generated

NOT Operations

There is no NOT in MIPS instructions

Useful to invert bits in a word

* Change 0 to 1, and 1 to 0

MIPS has NOR 3-operand instruction

* a NOR b == NOT (a OR b)

Table

Description automatically generated

MIPS Logical Instructions

Table

Description automatically generated with medium confidence

Load Upper Immediate

We cant directly load 32 bit number inside a 32 bit register bc complete instruction is 32 bit fixed. You have to give opcode, and the register address that you want to load to.

addi $t0, $zero, 0x1a2b ------> komutu ile most sign. 16 biti 0 olan, least sig. 16 biti de istediğim sayı (0x1a2b) olan bir sayı koyabilirim. Ancak yine istediğim 32 bitlik sayıyı koyamam.

Table

Description automatically generated with medium confidence

Transfers the immediate field into the register’s most significant 16 bits and fills the register’s lower 16 bits with zeros

Table

Description automatically generated

Least significant 16 bit de sıfırlandığı için değişime hazır hale geldi.

Bundan sonra “ori” komutuyla least significant 16 biti de istediğimiz gibi doldururuz.

Large Constants

We’d like to be able to load a 32 bit constant into a register

Must use two instructions, new “load upper immediate” instruction

Text

Description automatically generated with low confidence

Then must get the lower order bits right, i.e.,

Table

Description automatically generated

!!! Önce lui, sonra ori kullanılır !!!

addi, ori, andi,… bütün immediate instructionlar 🡪 registerde 32 bit var ama verdiğimiz sayı 16 bit nasıl işlem yapıyoruz?

* 16 bit sayı 32 bite çıkartılır. Kimisinde başına 0 koyarak, kimisinde başına 1 koyarak.
* ori 🡪 başına 0 koyarak 32 bite çıkartılır. orlama least sig 16 bitte gerçekleşir.
* andi 🡪 başına 0 koyarak 32 bite çıkartılır. Sayının most sig 16 biti kaybedilir, bu istenilen şey çünkü and maskelemek için kullanılıyor, sayının belli kısmını tutmak diğer kısımları sıfırlamak istiyoruz.
* addi 🡪 sign extend yapılır.

Unsigned Binary Integers

Given an n-bit number



Range: 0 to 2n-1

Example

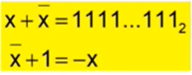
* 0000 0000 0000 0000 0000 0000 0000 10112
* 0 + … + 1x23 + 0x22 + 1x21 + 1x20
* 0 + … + 8 + 0 + 2 + 1 = 1110

Using 32 bits ----> 0 to +4,294,967,295

Signed Negation

Complement and add 1

* Complement means 1🡪0, 0🡪1



Example: negate +2

A picture containing icon

Description automatically generated

2s-Complement Signed Integers

Given an n-bit number

Text

Description automatically generated with low confidence

Range: -2n-1 to +2n-1-1

Example:

* 1111 1111 1111 1111 1111 1111 1111 11002
* = - 0000 0000 0000 0000 0000 0000 0000 01002
* -410

Using 32 bits

* -2,147,483,648 to +2,147,483,647

2s complement alma: Sağdan başla, ilk 1’den sonrasını ters çevir.

232 = 1000 000 … 0000 🡪 32 tane sıfır

B’s 2s complement = 2n - B



A – B = A + 232 – B = A – B + 232 ----> A – B is 32 bit

-231 = 1000 0000 … 0000   
231-1 = 0111 1111 … 1111

2s-Complement Signed Integers

Bit 31 is sign bit

* 1 for negative numbers
* 0 for non-negative numbers

-(-2n-1) cant be represented

Non-negative numbers have the same unsigned and 2s-complement representation

Some specific numbers

* 0: 0000 0000 … 0000
* -1: 1111 1111 … 1111
* Most-negative: 1000 0000 … 0000 = -231
* Most-positive: 0111 1111 … 1111 = -231-1

Sign Extension

Representing a number using more bits

* Preserve the numeric value

In MIPS instruction set

* addi: extend immediate value
* lb, lh: extend loaded byte/halfword
* beq, bne: extend the displacement

Replicate the sign bit to the left

* c.f. unsigned values: extend with 0s

Examples: 8-bit to 16-bit



For addi, we use sign extension.

MIPS Data Transfer Instructions

A picture containing text

Description automatically generated

MIPS supports byte addressing which means you can take individual bytes from memory and you can change the individual bytes inside memory.

sh 🡪 takes 16 bit number from register and stores it to memory  
sb 🡪 takes least significant byte from register and stores it to memory

There is unsigned notation of lh bc 16 bit number has to increased to 32 bits. lh performs sign extend, lhu performs zero extend.

For store byte, $1 + 12 may not be the multiple of 4 but for store word it must be.

Store Byte (sb) Instruction

Diagram

Description automatically generated

Load Byte (lb) Instruction

Diagram

Description automatically generated

lbu uses zero extend.

Conditional Operations

Branch to a labeled instruction if a condition is true

* Otherwise, continue sequentially

beq $rs, $rt, L1

* if (rs == rt) branch to instruction labeled L1;

bne $rs, $rt, L1

* if (rs != rt) branch to instruction labeled L1;

j L1

* unconditional jump to instruction labeled L1

Text

Description automatically generated

Normally, program counter is increased by 4 bytes (bc each instruction is 4 byte) each time so that it shows the next instruction.

In beq or bne, if the condition is satisfied, program counter shows the address of label L1. Assembler handles that. Assembler knows the labels and corresponding addresses for the labels and adjust the instruction machine code so that the CPU can understand where it should jump.

Instruction has to be 32 bit number. We have:

* 6 bits for opcode (bne) + 5 bits for $rs + 5 bits for $rt = 16 bits in machine code
* With the remaining 16 bits, we should explain to processor where it should jump.

Label can be any texture name. In Assembly, use first colon for the labels. You don’t put texture representation (or ASCII code) of the label characters to the machine code. Assembler puts a number to the machine code. This number helps CPU to understand address of the label.

Compiling If Statements

Diagram

Description automatically generated

You don’t have to write Else, it is just name of the label you can write anything.

j Exit --> bunu demezsen Else labelını da çalıştırır.

Compiling Loop Statements

Text

Description automatically generated

sll $t1, $s3, 2 ------> multiplies i with 4 bc &save[i] = $s6 + 4i -----> each integer is 4 bytes

add $t1, $t1, $s6 ------> $t1 = &save[i]

* 4i + &save[0] = &save[i]

lw $t0, 0($t1) -----> $t0 registerına save[i] değerini yükledim

addi kısmında s3’e 1 eklemek yerine 4 ekleyip sll instructionından kurtulabiliriz.

* add $t1, $t1, $s6
* lw $t0, 0($t1)
* bne $t0, $s5, Exit
* addi $s3, $s3, 4
* j Loop

En üste “add $t1, $zero, $s6” ekleyip (save[0]’ın adresini t1’e yükledik) addi kısmını “addi $t1, $t1, 4” şeklinde değiştirebiliriz:

* add(i?) $t1, $zero, $s6
* Loop: lw $t0, 0($t1)

bne $t0, $s5, Exit

addi $t1, $t1, 4

j Loop

Bu optimizasyon time complexity olarak bir değişiklik olmaz ama süre olarak epey bir verim sağlar.

Branch Instructions

Text, letter

Description automatically generated

Instead of “and $s0, $s0, $zero” we can also use “add(i?) $s0, $zero, $zero”

More Conditional Operations

Set result to 1 if a condition is true

* Otherwise, set to 0

slt $rd, $rs, $rt

* if ($rs < $rt) $rd = 1; else $rd = 0;

slti $rt, $rs, constant

* if ($rs < constant) $rt = 1; else $rt = 0;

Use in combination with beq, bne (branch if s1 <= s2)

A picture containing text

Description automatically generated

branch if s1 > s2:

* slt $t0, $s2, $s1
* bne $t0, $zero, L
  + Hemen öncesinde slt çalıştığı için t0 ya 0’dır ya da 1’dir

branch if s1 >= s2:

* slt $t0, $s1, $s2
* beq $t0, $zero, L

Signed vs. Unsigned

Signed comparison: slt, slti

Unsigned comparison: sltu, sltui

* Being small or big depends on the type of the number

For “<” sign in C:

* If numbers are unsigned int, then sltu or sltui are used
* If numbers are normal int, then slt or slti are used

Example:

Text

Description automatically generated

Signed vs. Unsigned Comparison Example

Text

Description automatically generated

R4 = 0  
R5 = 1  
R6 = 0  
R7 = 0

MIPS Compare and Branch

Compare and Branch (We have these two instructions in MIPS)

* beq rs, rt, offset if R[rs] == R[rt] then PC-relative branch
* bne rs, rt, offset <>

Compare to Zero and Branch (There are no instructions like these in MIPS, but you may use in MARS. These are pseudoinstructions)

* blez rs, offset if R[rs] <= 0 then PC-relative branch
* bgtz rs, offset >
* bltz rs, offset <
* bgez rs, offset >=

Remaining set of compare and branch take two instructions

MIPS Compare and Jump Instructions

A picture containing timeline

Description automatically generated

jal 🡪 puts the next instruction address into register with address 31 ($ra : return address) and then jumps, only difference from j is jal stores the next instruction address inside a register

jr 🡪 jumps to an address given by a register. If you use ra register, it means you can jump back to location which called the function. So jr is used for function return backs.

Procedure = Subroutine 🡪 function in assembly

How can we supply address to jump instructions bc our instructions are 32 bit wide, and memory address is also 32 bit wide? So we cant use 32 bit address inside our instruction bc total instruction is 32 bit. We have J-Type instructions for that purpose.

**J-TYPE OPERANDS**

Graphical user interface, application, table

Description automatically generated with medium confidence Ekran görüntüsündeki 10000 decimal sayısı. Binaryde sona 2 bit eklersen o sayıyı 4 ile çarparsın, 40000 oradan geliyor.

26 bits are for supplying a jump address. We should increase 26 bits to 32 bits.

Each instruction has an address which is multiple of 4 (4 bytes = 32 bits). Being a constant multiple of 4 means that least significant 2 bits must be 0. These 2 bits show the remainder when the number is divided by 4. I don’t need to give these 2 bits bc I know they are 0.

When jump instruction comes to the processor, it takes 26 bit number then it combines that number with 2 zeros. So number becomes 28 bit. In order to make this number 32 bit, we get most significant 4 bits from the program counter (PC). As a result we get 32 bit number which represents the jump address.

We cant jump any address for which the most significant 4 bits are different than the program counter’s current value. So the most sig 4 bits of the program counter will always be the same number. Other than that, we can jump any address by using jump instruction.

Neden program counterdaki most sig 4 bit alınır? Çünkü program counter ile adreslerde ilerlerken +4, +8,… diye en son değişecek kısım orası. Yani instruction memory’nin tamamı boyunca program counterın most sig 4 biti sabit kalır. Çünkü instruction memory’yi nereye yerleştirirsen yerleştir o yerleştirdiğin bölgede program counterın most sig 4 bitini değiştirecek kadar büyük bir bölge değil instruction memory. Onun için o 4 bit sabit. Hatta MIPS için program counterın most sig 4 biti instruction memory boyunca 0 olması lazım.

Assembler jump instructionı machine code’a çevirirken atlayacağı yerin adresinin sonundaki 2 biti (00) ve başındaki 4 biti kesiyor. Geriye 26 bit kalıyor.

Instructiona atlamak demek adresi program countera yüklemek demek.

j L1:

* Assembler L1’in adresinin ne olduğuna, instruction memory’de nereye denk geldiğine bakar. Sonundaki 2 biti (00) ve başındaki 4 biti (0000) atar.
* Processor bu işlemi tersten yapar. 32 bitlik adresi alıp program countera yükler. Böylece bir yerden bir yere atlanmış olur.

Program counterın most sig. 4 biti hiç değişmez 🡪 0000

Target Addressing Example

Loop code from earlier example

* Assume Loop at location 80000

Table

Description automatically generated with medium confidence

I-type bne instructionında Exit var. Bu label 16 bitlik sayıya çevriliyor. 16 bitlik sayıyı 32 bite çevirip (j’de olduğu gibi) o adrese gitmek yerine şöyle yaparız: 2 demek 2 instruction ileri git demek: 2x4 = 8. Bu 8’i sonraki instructiona eklersek (80016) 80024’e gideriz.

Processor 2’yi nasıl anlıyor:

* Processorün içerisinde adder var. Bu adderin görevi program counter ile gelen offseti toplamak:
  + PC + offset \* 4 = 80012+4 + 2\*4 🡪 Bu sonuç program countera yüklenir
    - PC’ın o anki değeri bne’nin bulunduğu adresin 4 fazlası çünkü PC kendisini çoktan update etmiş oluyor.

80012 yerine x diyelim o adresi bilmiyor olalım. Assembler x+12’ye atlayacağını biliyor, yaptığı işlem şu 🡪 Exit neresi: x+12 (3 instruction sonrası). PC nerede: x+4’te.

* x+12 – (x+4) = 8 🡪 8/4 = 2 instruction gitmeli o yüzden offsete 2 yaz
* Processor daha sonra buradan aldığı değeri 4’le çarpıp PC’nin o anki değeriyle (x+4) toplayıp PC’ye yükler ve zıplamasına sağlar

Kod derlendiğinde object file oluşur. Bu fileda adresler henüz ortaya çıkmış değil. Adresler link işleminden sonra netleşir. Link işleminden sonra her şey netleştiği zaman bütün adresler ortaya çıkar. Mesela pow fonksiyonu çağırıyorsun diyelim. Bu mat librarysi içinde. Sen math librarysinin object fileı ile (machine koduyla) kendi kodunu birleştirmeden o powun hangi adrese geleceği belli değil. O belli olduktan sonra powun kaç adres ileride olduğunu anlıyor ve ona göre yapar.

Kendi kodun içerisinde bir yere atlamak istersen assembler Exit’in neresi olduğunu bilebilir. Çünkü kodun memory’ye tek blok olarak yerleştirilecek.

Exit yerine Loop olsaydı 2 yerine -4 yazardık.

bne adresi x olsun, addi adresi de x+4 olurdu. Processor şöyle yapardı:

* x+4 + (-4 \* 4) = x-12 🡪 Loop’un adresi.

Bir instructionın, instruction memory’deki her yere atlayabilmesi lazım. 🡪 j

Bazen branch çok uzak bir yere branch etmek ister, o kadar uzağa branch etmek mümkün değil çünkü offset 16 bit. Burada jump kullanılır.

Branching Far Away

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

Example:

Text

Description automatically generated with medium confidence

* Diyelim L1 çok uzakta. Assembler 16 bitlik sayıyla o kadar uzağa atlayamaz. Bu instructionı (beq olan) diğer 2 instructiona çevirir.

Instruction memory, memorynin küçük bir kısmı. Yani PC most sig. 4 biti sabit olabilir.

MIPS Instruction Types

Table

Description automatically generated

Translating MIPS Assembly into Machine Language

Table

Description automatically generated

MIPS Addressing Modes

Addressing modes specify where the data used by an instruction is located

Often, the type of addressing mode depends on the type of operation being performed (e.g., branches all use PC relative 🡪 How many addresses you go below or above)

Five addressing modes are used in MIPS 3000:

1. Register addressing (R-type)

Graphical user interface, application

Description automatically generated

Direkt register adresi veriliyor, o adresin içerisinden içerik alınır.

1. Immediate addressing (I-type)
   1. A picture containing diagram

      Description automatically generated
   2. Direkt instructionın içerisindeki sayı kullanılır.
2. Base addressing (I-type)

Diagram

Description automatically generated

Immed fieldından alınan sayı ile register contenti toplanarak bir memory adresi bulunur ve gidip content memoryden alınır.

1. PC-relative addressing (I-type)

Diagram

Description automatically generated

16 bitlik offset alınır, program counter ile toplanarak gideceği yerin adresi hesaplanır.

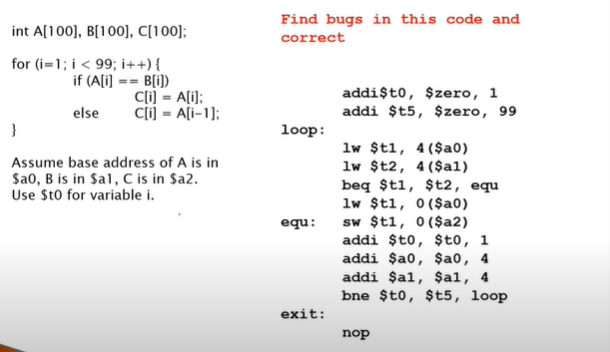
1. Pseudodirect addressing (J-type)

Diagram

Description automatically generated

Neredeyse adresi direkt veriyorsun.

An Assembly Example



Pseudo-instructions

The MIPS assembler supports several pseudo-instructions:

* not directly supported in hardware
* implemented using one or more supported instructions
* simplify assembly language programming and translation

Graphical user interface, text, application, chat or text message

Description automatically generated

at (assembler temporary) register 🡪 used for assembler to convert pseudo-instructions to real instructions

at’ye bir değer atayıp pseudo-instruction kullanırsan at’deki değeri kaybedersin. YANİ at REGISTERINI HİÇ KULLANMA.

Assembler önce kodunu bi preprocess edip pseudo-instructionların hepsini gerçek instructionlara çevirir.

Pseudo-instruction kullanırsan cycle sayısına dikkat et. Mesela her instruction 1 cycle’sa “blt $s0, $s1, Else” instructionı 2 cycle harcayacak, 1 değil.

Details of the MIPS Instruction Set

Register zero always has the value zero (even if you try to write it)

Branch/jump and link put the return addr. PC+4 into the link register (R31)

All instruction change all 32 bits of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, …)

Immediate arithmetic and logical instructions are extended as follows:

* logical immediates ops are zero extended to 32 bits
* arithmetic immediates ops are sign extended to 32 bits

The data loaded by the instructions lb and lh are extended as follows:

* lbu, lhu are zero extended
* lb, lh are sign extended

Overflow can occur in these arithmetic and logical instructions:

* add, sub, addi

It cannot occur in

* addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu

Summary: Features of MIPS ISA

32-bit fixed format inst (3 formats)

31 tane 32-bit GPR (R0 contains zero) and 32 FP registers (and PC, HI, and LO)

3-address, reg-reg arithmetic instr.

Single address mode for load/store:

* base + displacement

Simple branch conditions

* compare one register against zero or two registers for =,

PROCESSOR TASARLAMAK DEMEK, GÖRDÜĞÜMÜZ TÜM BU INSTRUCTIONLARI ÇALIŞTIRAN MAKİNE YAPMAK DEMEK.