A picture containing diagram

Description automatically generatedMemory Layout

Reserved: for interrupts, exceptions, OS, etc.

Text: program code, instruction memory

Static data: global variables, known before runtime

* e.g., static variables in C, constant arrays and strings
* $gp initialized to address allowing offsets into this segment
* Bir şeyin karşılığını bulmak için buraya lookup tabloları yerleştirebilirsin.

Dynamic data: heap

* e.g., malloc in C, new in Java

Stack: automatic storage, local variables

Instruction memory starts from 0040 0000. Last address for the text can be written as 🡪 0FFF FFFC (C bc instruction addresses must be constant multiple of 4).

* As you see from red 0s, most sig. 4 bits of PC are always 0.

So next value is 🡪 1000 0000 belongs to static data segment

Stack pointer shows the last position inside our stack. When you put a value to stack, you should always decrement stack pointer by 4.

global pointer shows the middle of the static data segment.

* In lw and sw, we have base addressing 🡪 lw … offset($register)
* global pointer is a register in which we have 1000 8000 value which shows the middle of the static data.
* By putting negative and positive 16-bits offsets, you can access any address inside the static data segment.

Static data segmentine koyduğun ilk variable adresi: 1000 0000

Assembler Syntax

Comments in assembler files begin with a sharp sign (#)

Instruction opcodes (e.g. lw, add, etc.) are reserved words that cannot be used as identifier

Labels are declared by putting them at the beginning of a line followed by a colon (:), for example:

Graphical user interface, text, application

Description automatically generated

Preprocessor directives (like define in C) start with a dot (.). These directives are not converted to machine code bc they are not instructions. They tell sth to assembler. For example you can store static data into memory using these directives. Assembler understands that you want to store some binary representation inside your static data location and puts them inside there.

In assembly you have 3 columns: label – directive/instruction – comment

.data directive tells the assembler that the next part will be put inside static data segment in memory. .word will be put as a word (32-bit number) inside memory. 25 can be 8 bit, 10 bit, etc. So it will be put as integer, as 32-bit in memory.

.text directive tells the assembler that put the machine codes of the below instructions inside the text segment.

You can write .data after .text.

.globl main means that if you use multiple files inside which your different assembly procedures are written, then you say that this main is not main only for this file but all files I will use will jump to this main. Even if you write main in separate file, it can be read by the other file.

You don’t have to have a main label.

Numbers are base 10 by default, for example:

* addi $t0, 20 -----------> addi $t0, 0x14 (0x means number is hexadecimal)

Strings are enclosed in double quotes (“)

Graphical user interface, text, application

Description automatically generated

temp1 label directly represent the address of word which is 3.

.asciiz directive puts a string instead of a number inside static data memory.

Two Similar Examples

Table

Description automatically generated

li pseudo-instruction can be converted to addi bc we want to put a number which is smaller than 16 bits. If we want to put 32 bit number, we could use li but that time assembler would convert that li to lui and ori

Normally lw is like “lw $t0, offset($register)” but lw in example is for making your task easier.

If you use temp1 and temp2 lots of times 2nd one is better. If you use them just once 1st is better.

System Services

Table

Description automatically generated

When you type syscall, your program stops executing and control is given to OS.

read\_string:

* start address of the string is put to a0 register
* length of the string is put to a1 register

Text

Description automatically generatedSystem Call Example

li – la yeri değişebilir, fark etmez.

la 🡪 load address. Tells assembler that we don’t want to content of str address. First 4 characters (32 bits) are the content of str address.

MIPS Register Conventions

This is an agreed upon “contract” or “protocol” that everybody follows

Specifies correct (and expected) usage, and some naming conventions

Established part of architecture

Used by all compilers, programs, and libraries

Assures compatibility

Table

Description automatically generated with medium confidence

You can’t change $zero.

You shouldn’t change $at.

You should use $v0 and $v1 for returning values. If you need to return more than 1 values, you should use stack.

For arguments passing to functions, you should use $a registers. If you want to pass more than 4 arguments, you should use stack.

You can use $t and $s register for any general purposes.

$k registers are reserved for kernel, don’t use them.

$gp always show the middle point of the static data segment so that you can access any location in static data segment by using offset.

$sp shows the last position in your stack. You shouldn’t modify it independently from the stack. If you push any word to stack, you should decrement it and if you pop values from stack, you should increment it.

$ra is used for especially jal (jump and link) instruction. jal puts the return address inside this register. Don’t change the value in $ra other than jal.

Difference between t and s registers:

* Inside a procedure, you want to change the content of a t registers, you don’t have to think if this content will be used by others or if I change the content what would happen to function which called me. You can change them without storing them inside stack or without holding their previous values. If the calling function (caller) needs the values inside t registers, it is the responsibility of that calling function to store these values inside stack and then restore them back. Restoration of t register contents is the responsibility of the caller.
* For s registers, if the function that is called wants to change the contents of the s registers, it should store the previous values into stack and before returning back to caller, it should restore these values so that caller will find the same values before the call. Restoration of s register contents is the responsibility of callee.

Any callee can change t registers without saving them. So you are responsible for them.

s register contents should be preserved to stack by callee.

Program Counter

We need a register to hold the address of the current instruction being executed

* “Program Counter” PC in MIPS

jal saves PC+4 in register $ra automatically

At the end of the procedure we jump back to the $ra (an unconditional jump)

* jr $ra 🡪 ra içindekini alıp program countera yükler

The caller puts the parameter values in $a0-$a3

The caller uses “jal X” to jump to procedure X

The callee performs the calculations, places the results in $v0-$v1

Returns control to the caller by “jr $ra”

If you don’t put “jr $ra” program continues with next instruction.

İç içe fonksiyon çağırırsan $ra’yı stackte store etmeli, sonra çağırdığın fonksiyon returnleyince geri yüklemelisin. Sonra “jr $ra” demelisin.

Other than calling procedure with jal and returning back with jr, procedures are same as labels.

Stack

Suppose the procedure needs more than 4 arguments

We store the values in Stack (a last-in-first-out queue)

A stack needs a pointer to the most recently allocated address in the stack: stack pointer

Placing data onto the stack is called a PUSH (you use sw and addi for this 🡪 addi for updating sp)  
Removing data from the stack is called a POP (you use lw and addi for this 🡪 addi for updating sp)

The stack pointer in MIPS is $sp. By convention stack “grow” from higher addresses to lower addresses!! (You push values onto the stack by subtracting from the stack pointer)

5 integer koyduysan sp’yi 20 (5x4) indirmelisin.

Assembly’de function yerine procedure deriz.

Procedure call

When making a procedure call, it is necessary to:

* Place inputs where the procedure can access them (into $a registers + stack)
* Transfer control to procedure (jal)
* Acquire the storage resources needed for the procedure
* Perform the desired task
* Place the result value(s) in a place where the calling program can access it (into $v registers + stack)
* Return control to the point of origin (jr $ra)

MIPS

* Provides instructions to assist in procedure calls (jal) and returns (jr)
* Uses software conventions to
  + place procedure input and output values
  + control which registers are saved/restored by caller and callee
* Uses a software stack to save/restore values

Procedure Call Instructions

Procedure call: jump and link

* jal ProcedureLabel
  + Address of following instruction put in $ra
  + Jumps to target address

Procedure return: jump register

* jr $ra
  + Copies $ra to program counter
  + Can also be used for computed jumps
    - e.g., for case/switch statements

Simple Procedure Call

Text

Description automatically generated with medium confidence

mult 🡪 çarpımın least s. 32 bitini LO’ya, most s. 32 bitini HI’ya koyar.

mflo $v0 🡪 least significant 32 bit result is copied to v0 (mflo : move from LO, mfhi : move from HI)

temp1: .word 3 🡪 bunu diyerek 1 word koyduk, yani adresi 4 artırdık. 4 ilerideki adres str’ın başlangıç adresi.  
str: .asciiz “Result = “ 🡪 str kaç bytesa bi sonraki koyduğunun adresi o kadar byte ilerisi

Leaf Procedure Example

Leaf procedure: last procedure in the hierarchy. They don’t call other procedures. There is no nested calls inside leaf procedures.

C code:

A picture containing text

Description automatically generated

Arguments g, h, i, j int $a0, $a1, $a2, $a3

f in $s0 (hence, need to save $s0 on stack)

Result in $v0

A picture containing table

Description automatically generated

If you open stack for more than -4 (for example -8), but you close it by 4, then it results in memory leakage.

A procedure call with a stack:

Text, table

Description automatically generated

According to contract, we don’t have to store t1 and t0. We can use them freely. It is responsibility of caller.

Diagram

Description automatically generated

Local Data on the Stack

Diagram

Description automatically generated with low confidence

$fp (frame pointer) is used by compiler but you as a programmer can use it for any purpose. Especially it is used for showing the part of the function stack. Whenever you are inside a function, first position of the stack for that function is shown by fp. Last position of the stack is shown by sp. All local variables are located between fp and sp. fp is not a necessary component but sp is necessary.

C Sort Example

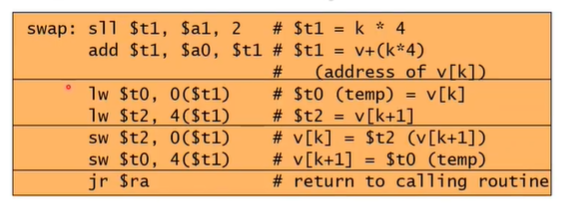
Illustrates use of assembly instructions for a C sort function

Swap procedure (leaf)

Text

Description automatically generated

* v in $a0, k in $a1, temp in $t0



It doesn’t use any s registers so no need to stack.

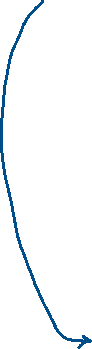
Sort procedure in C:

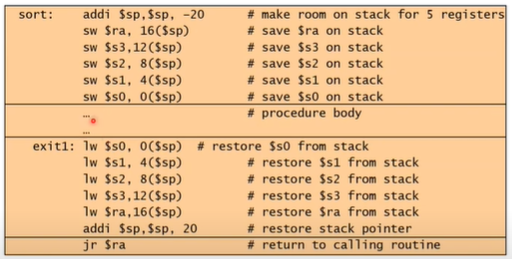
Text

Description automatically generated with medium confidence

Text, table

Description automatically generated





swap fonksiyonu çağrıldığı ve ra kaybolduğu için ra’yi ilk başta stack’e koyduk.

Non-Leaf Procedures

Procedures that call other procedures

For nested call, caller needs to save on the stack:

* Its return address
* Any arguments and temporaries needed after the call

Restore from the stack after the call

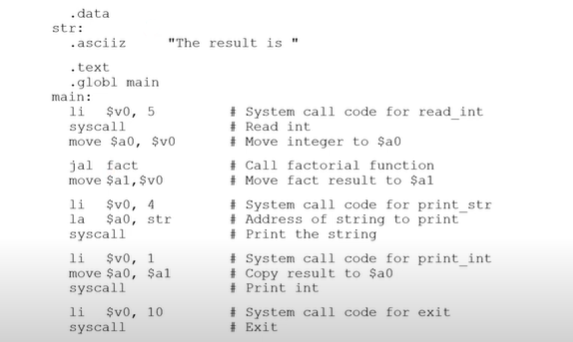
Nested Procedure Example

C code:

Text

Description automatically generated with medium confidence

Argument n in $a0  
 Result in $v0



3!

Text

Description automatically generated

Blue color 🡪 after 0 is inserted to stack

A picture containing text, whiteboard

Description automatically generated

Arrays versus Pointers

Graphical user interface, text, application

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

clear1 de 6 tane instruction var 🡪 6n clear2 de 4 tane instruction var 🡪 4n

Yani clear2 aynı işi yaptığı halde 1.5 kat daha hızlı çalışır.

clear1deki indexi 4’le çarpıp base adresle toplama işlemi zaman kaybettiriyor. Onun yerine başlangıç adresini 4, 4 artırarak da arrayin bütün elemanlarını gezebilirsin.

A Translation Hierarchy

A screenshot of a computer

Description automatically generated with low confidence

Object code is not an executable. It may require different library modules.

Executable file is put in disk.

Your object file has the machine code for all the instructions. For branch instructions, for jump and link instructions, etc. you don’t know the exact addresses yet. So you cant have the machine codes for these instructions. This is why you cant execute this object file.

After loader puts executable in memory, it tells to CPU the initial address of your instructions. CPU loads PC with initial address and control is taken by your program.

Static linking yaptığımızda library güncellenirse, biz eski versiyonsa link ettiğimiz için execute ettiğimizde eski versiyona göre çalışır.

Dynamic linkage, çalışma sırasında programın library’yi bulup object fileın ile birleştirmesi. İlk çağrılmada yavaş olur, sonraki çağrılmalarda yer belli olduğu için aynı hızda çalışır.

Producing an Object Module

There are different parts of the object module

Assembler (or compiler) translates program into machine instructions

Provides information for building a complete program from the pieces

* Header: described contents of object module
* Text segment: translated instructions
* Static data segment data allocated for the life of the program
* Relocation info: for contents that depend on absolute location of loaded program
* Symbol table: global definitions and external refs
* Debug info: for associating with source code

Linking Object Modules

Produces an executable image

1. Merges segments
2. Resolve labels (determine their addresses)
3. Patch location-dependent and external refs

Loading a Program

Load from image file on disk into memory

1. Read header to determine segment sizes
2. Create virtual address space
3. Copy text and initialized data into memory
   1. Or set page table entries so they can be faulted in
4. Set up arguments on stack
5. Initialize registers (including $sp, $fp, $gp)
6. Jump to startup routine
   1. Copies arguments to $a0, … and calls main
   2. When main returns, do exit syscall

Two Object Files

Table

Description automatically generated

Both object files calls functions that are located in other file.

“jal 0” corresponds to another procedure which is procedure B.

This tries to read data shown by the global pointer and 0 offset bc we don’t know the exact position of the static data yet. The exact position of the static data and the exact position of the functions are not known before linking the object files. We put 0s but these 0s may change after linking operation.

Relocation information shows us all the relevant information for resolving the labels inside our program. You can see in address 0 we have problematic instruction which is lw and it has a dependency on X. It has a static data X but for now, we don’t know the exact position of X. But we know lw wants to load that X value. We know bc it is a static data.

Also we know there is a problematic instruction in address 4 which is jal. We know that it wants to jump to procedure B but I don’t know the exact address of procedure B (bc procedure B is not located inside this object file). Exact address will be resolved after linking all object files required by my program. So I can only put that relocation information inside my object file. I cant convert them to machine code.

At symbol table, we have dependent values and their corresponding address which are unknown for now.

In the second object file of the second assembly program we have procedure B. Text size means this program requires this amount of bytes. Data size is static data required by this program.

In procedure B we have sw instruction, we don’t know the exact address but we want to store our register content to Y. I don’t know the exact position of Y so in relocation information I have problematic instruction sw and its dependency on Y. Also I have jal in which I want to jump and link to procedure A. Procedure A is in another object file.

Linker takes these 2 object files and combines them. At the end all addresses are resolved.

Executable

Table

Description automatically generated

We can put procedure A first or B first. It depends on linker.

I know first procedure has 20 bytes of data so I put procedure B’s data 20 after the procedure A’s data.

When I want to load the value of X, I know this is the first address of the static data segment. So in order to access the first address, I have to decrease the biggest offset from the global pointer. gp shows the middle location of the static data segment and if I decrease global pointer with the highest magnitude, then I get the first address of the static data segment. In order to get the first address of the static data segment, we have 8000 (lw instruction). This is a negative number with the largest magnitude that you can represent with 16 bits (negative bc most sig. bit is 1). We want to access to first location so we use this number.

As the location of X is resolved now, we can put required number (8000) in lw instruction.

Then I want to access 1000 0020 address in procedure B bc static data of procedure B is Y and it wants to access Y. In order to access Y, offset must be 8020 in sw instruction.

Now as the linker combined my object files and all addresses are resolved in my file, I can convert this whole program into machine code. So this is the executable (machine code version of the executable table).

Linker, farklı programların farklı librarylerin birbiriyle dependencylerine bakarak onları birleştirip tek bir koda dönüştürüyor ve bütün adresleri belirliyor. Ona göre de branchlerin, jump-and-link lerin, storeword lerin offsetleri ne olacaksa onları belirliyor. Sonra bunların hepsini machine koda çeviriyor ve bunları çalıştırılacağı zaman binary sayılar olarak memory ye alıyor. CPU da bu instructionların birincisinden başlayarak bunları çalıştırıyor.

Table

Description automatically generated

Advantages & Disadvantages

* Assembly programming is useful when the speed or size of a program is important.
  + n2, n olmaz ama başındaki constant düşer
* But assembly languages are machine specific and they must be rewritten to run on another machine.
* Another disadvantage is that assembly language programs are longer than the equivalent programs written in a high-level language.
* It is also true that programs written in assembly are more difficult to read and understand and they may contain more bugs.

Assembly Language & Programming

Assembly language is the symbolic representation of a computer’s binary encoding, which is called machine language.

Assembly language is more readable than machine language because it uses symbols instead of bits.

Assembly language permits programmers to use labels to identify and name particular memory words that hold instructions or data.

A tool called assembler translates assembly language into binary instructions.

An assembler reads a single assembly language source file and produces object file containing machine instructions and bookkeeping information that helps combine several object files into a program.

Object file 🡪 Access etmek istediğimiz yerler, dependencieler var. Kod size belli, static data ne kadar kullanacağımız belli. Ancak henüz resolve edilmemiş labellar var. Labelların adresleri belirlenmemiş. Bunlar linker handle ettikten sonra belirlenir ve executable oluşturulur. O yüzden libraryler object file olarak saklanır.

Assembler Pseudoinstructions

Most assembler instructions represent machine instructions one-to-one

Pseudoinstructions: figments of the assembler’s imagination

move $t0, $t1 → add $t0, $zero, $t1

blt $t0, $t1, L → slt $at, $t0, $t1   
 bne $at, $zero, L

$at (register 1): assembler temporary