Abstractions and Computers and the MAL programming Language
Computer Architecture

Interface between a computer's hardware and its software. Defines exactly what the computer's instructions do, and how they are specified.
Compiler: A computer program that translates code written in a high level language into an intermediate level abstract language.
Computer Science

Fundamentally the study of algorithms and data structures.

Abstraction: Use of level of abstraction in software design allows the programmer to focus on a critical set of problems without having to deal with irrelevant details.
Procedure or Function

**Average (a, b)**

Begin
   \[ \text{avg} = (a+b)/2; \]
   return (avg);
End

**Main ()**

\[ \ldots \]
\[ x = 4; \]
\[ y = 2; \]
\[ k = \text{average} \ (x,y); \]
\[ \text{print} \ ("%d", \ k); \]
\[ \ldots \]
CPU Interacts with the memory in 3 ways:
• fetches instructions
• loads the value of a variable
• stores the new value of a variable

Memory is capable of only 2 operations:
• reads – a load or a fetch
• writes – operation of a memorizing the value of a variable
Hierarchy

Designs are hierarchical (though not strictly)

1. Transistors
2. Logic gates, flip-flops (defined in terms of transistors)
3. Components – registers, ALUs, memory chips
4. Computer system
Instruction Fetch / Execute Cycle

In addition to input & output a program also does:

• Evaluates arithmetic & logical functions to determine values to assign to variable.
• Determines the order of execution of the statements in the program.

In assembly this distinction is captured in the notion of Arithmetic, logical, and control instructions.
Arithmetic and logical instructions evaluate variables and assign new values to variables.

Control instructions test or compare values of variable and make decisions about what instruction is to be executed next.

**Program Counter (PC)**

Basically the address at which the current executing instruction exists.

1. load rega, 10
2. load regb, 20
3. add regc, rega, regb
4. beq regc, regd, 8
5. store regd, rege
6. store regc, regd
7. load regb, 15
8. load rega, 30
The CPU begins the execution of an instruction by supplying the value of the PC to the memory & initiating a read operation (fetch).

The CPU “decodes” the instruction by identifying the opcode and the operands.

For example:

**PC → add A, B, C**

CPU gets instruction Decodes it and sees it is an add operation, needs to get B & C
CPU executes a load operation, gives address of variable B Does the same for variable C
Does the “add” operation and stores the result in A

PC increments automatically unless a control instruction is used.

**Branch** – like a goto instruction, next instruction to be fetched & executed is an instruction other than the next in memory.
Registers and MAL

CISC vs. RISC
CISC: complex instruction set computer
RISC: reduced instruction set computer
Breaking down an instruction

add a, b, c
Locality of reference

We need techniques to reduce the instruction size. From observation of programs we see that a small and predictable set of variables tend to be referenced much more often than other variables.

Basically, locality is an indication that memory is not referenced randomly.

This is where the use of registers comes into play.
Specifying addresses

For a load/store architecture, registers are used to supply source operands and receive results from all instructions except loads and stores.

Basically, load the registers with the operands first, then perform the operation.
How do we fit the “stuff” in 32-bit instructions?

So we have arithmetic instructions and branch type instructions that cannot contain all the needed info in a single 32-bit word.

Ways to handle this:

1. Instruction might occupy 2 words.

2. Instruction might specify a register that contains the address.
3. Instruction might specify a small constant and a second register.

4. The instruction might specify 2 additional registers.
Solution: Addressing modes

- Immediate
  the operand is contained directly in the instruction
- Register
  the operand is contained in a register
- Direct
  The address of the operand is contained in the instruction (two-word instruction)
- Register Direct
  The address of the operand is contained in a register
- Base Displacement
  The address is computed as the sum of the contents of a register (the base) and a constant contained in the instruction (the displacement)
• Indirect

The instruction specifies a register containing an address the content of which is the address of the operand.

1-word instruction

```
| opcode | reg |
```

```
| reg    | address |
```

```
| Memory address | address |
```

Effective address
MAL

2 distinct register files, 32 general registers, and 16 floating point registers.

The 32 general registers are numbered $0 - $31.

$0 is always the value “Zero”
$1 is used by the assembler
$26 & $27 are used by the operating system
$28, $29, & $31 have special conventions for the use of
The 16 floating point registers are intended exclusively for holding floating point operands. These registers are 64-bits in size for holding both single precision (32-bit) floats and double precision (64-bit) floats.

These registers are named $f0$, $f2$, $f4$, …., $f30$.

MAL uses a single, versatile addressing mode for its regular load store instructions – base displacement.

General since its special cases provide for both direct and register direct address.
MAL has 3 basic types: integer, floating point, and character

C:
    type    variablename;

MAL:
    variablename:    type    value

Type is
    .word (integer)
    .byte (character)
    .float (floating point)

Value is optional – the initial value
Examples:
flag: .word 0
counter: .word 0
variable3: .word
e: .float 2.71828
uservalue: .byte
letter: .byte ‘a’

• One declaration per line
• Default initial value is 0
  (but you may lose points if you make use of this!!!!)
Directives give information to the assembler. All directives start with ‘.’ (period)

Examples:
- .byte
- .word
- .float
- .data  # identifies the start of the declaration section
  # there can be more than 1 .data sections in a program
- .text  # identifies where instructions are, there can be more than 1 .text sections in a program
- .asciiz “a string.\n”  # places a string into memory and null terminates the string
- .ascii “new string.”  # places a string into memory with no null termination.
<table>
<thead>
<tr>
<th>MAL</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $s1, x</td>
<td></td>
</tr>
<tr>
<td>lw $s2, y</td>
<td></td>
</tr>
<tr>
<td>move $s3, $s2</td>
<td></td>
</tr>
<tr>
<td>add $s3, $s1, $s2</td>
<td></td>
</tr>
<tr>
<td>sub $s3, $s1, $s2</td>
<td></td>
</tr>
<tr>
<td>mul $s3, $s1, $s2</td>
<td></td>
</tr>
<tr>
<td>div $s3, $s1, $s2</td>
<td></td>
</tr>
<tr>
<td>rem $s3, $s1, $s2</td>
<td></td>
</tr>
<tr>
<td>sw $s3, z</td>
<td></td>
</tr>
<tr>
<td>z = y;</td>
<td></td>
</tr>
<tr>
<td>z = x + y;</td>
<td></td>
</tr>
<tr>
<td>z = x - y;</td>
<td></td>
</tr>
<tr>
<td>z = x * y;</td>
<td></td>
</tr>
<tr>
<td>z = x / y;</td>
<td></td>
</tr>
<tr>
<td>z = x % y;</td>
<td></td>
</tr>
</tbody>
</table>

An immediate is a value specified in an instruction, not in .data.

Examples:

```
li $s2, 0               # load immediate
add $s2, $s2, 3        # add immediate
```
Simple MAL program

.data
avg: .word 0
i1: .word 20
i2: .word 13
i3: .word 2
.text
main:
lw $s1, i1
lw $s2, i2
lw $s3, i3
add $s4, $s1, $s2

div $s4, $s4, $s3
sw $s4, avg
li $2, 10  # done cmd
syscall
• Assembler translates to executable – machine language
• Linker combines multiple MAL files – if any
• Loader puts executable into memory and makes the CPU jump to first instruction or “main:”
  • Executes
• When executing done returns control to OS
  • Or simulator or monitor
• Load again to run again with different data
  • In this case, assemble again, too, since data is in program.

Special Symbols:
  # comment follows and ends at the end of line
. data    # data follows
. text    # instructions follow
main      # label to start program
HLL – if/else statements...

if (condition)
    statement;
else
    statement;

C:
if (count < 0)
    count = count + 1;

MAL:
    lw $t1, count
    bltz $t1, ifstuff
    b endif
ifstuff: add $t1, $t1, 1
endif:    # next program instruction goes here

OR:
    lw $t1, count
    bgez $t1, endif
    add $t1, $t1, 1
endif:    # next instruction goes here
Loops can be built out of IF’s – WHILE:

C:

```c
while (count > 0) {
    a = a % count;
    count--;
}
```

MAL:

```
lw $s1, count
lw $s2, a
while:    blez $s1, endwhile
rem $s2, $s2, $s1
sub $s1, $s1, 1
b while
endwhile: sw $s2, a
sw $s1, count
```
Repeat loops:

C:

/* do statement while expression is TRUE */
/* when expression is FALSE, exit loop */
do {
    if (a < b)
        a++;
    if (a > b)
        a--;
} while (a != b)

MAL:
    lw $s3, a
    lw $s4, b
repeat:    bge $s3, $s4, secondif
            add $s3, $s3, 1
    secondif:  ble $s3, $s4, until
            sub $s3, $s3, 1
    until:     bne $s3, $s4, repeat
While Loops (Part II)

C:

    while ( (count < limit) && (c == d) )
    {
        /* loop’s code goes here */
    }

MAL:

    while:    ?
    ?
    ?
    # loop code goes here
    ?

    endwhile:
For loops

C:

for ( I = 3; I <= 8; I++)
    { a = a+I;}

MAL:

? for: ?
?
?
?
?
endfor:
Simple procedure calls require 2 instructions:

**JAL** Jump and Link
- Link means save the return address in $ra ($31)

**JR** Jump Register
- Be careful with registers!!
- Cannot nest unless $ra is saved elsewhere
- Cannot be recursive without a stack
Communication with the user or outside world (IO)

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Putc</td>
<td>11</td>
</tr>
<tr>
<td>Put</td>
<td>1</td>
</tr>
<tr>
<td>Getc</td>
<td>12</td>
</tr>
<tr>
<td>Get</td>
<td>5</td>
</tr>
<tr>
<td>Puts</td>
<td>4</td>
</tr>
<tr>
<td>Done</td>
<td>10</td>
</tr>
</tbody>
</table>

A carriage return is ‘\n’.