**Moore’s Law**

Number of transistors that can be integrated on single chip would double about every two years

**Instruction Set Architecture (ISA)**

The hardware/software interface..

**Response time** How long it takes to do a task.

**Throughput** Total work done per unit time.

**Performance = 1 / Execution Time**

**Elapsed time**

Total response time, including all aspects (Processing, I/O, OS overhead, idle time). Determines system performance.

**Clock period (time)** duration of a clock cycle

**Clock frequency (rate)** cycles per second

**CPU time**

Time spent processing a given job (Discounts I/O time and other job’s shares). Comprises user CPU time and system CPU time. Different programs are affected differently by CPU and system performance.

**Power**

**The power wall**

We can’t reduce voltage further and remove more heat.

* **How improve performance?**Multicore microprocessors: More than one processor per chip. Requires explicitly parallel programing: Hardware executes multiple instructions at once. ***But*** hard to programming for performance, load balancing and optimizing communication and synchronization.

**PC (Program Counter)**

현재 수행중인 명령어의 주소를 기억하는 레지스터

**Amdahl’s Law**

Improving an aspect of a computer and expecting a proportional improvement in overall performance.

**Instruction Set**

The repertoire of instructions of a computer. Different computers have different instruction sets.

**Instruction Set Architecture (ISA)**

The abstract interface between the hardware and the lowest level software a machine language program, including instructions, registers, memory access, I/O, …

The combination of the basic instruction set (the ISA) and the operating system interface is called application binary interface (ABI)

* **ABI** The user portion of the instruction set plus the operating system interfaces used by application programmers.

**Operation design principle**

1. Simplicity favours regularity.

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

1. Smaller is faster.

c.f. main memory: millions of locations.

1. Make the common case fast.

Small constants are common. Immediate operand avoids a load instruction.

1. Good design demands good compromises.

Different formats complicate decoding, but allow 32-bit instructions uniformly. Keep formats as similar as possible

**Endian**

Arrange 0xABCDEF12 to memory.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Addr. | 0x00 | 0x01 | 0x02 | 0x03 |
| Big | 0xAB | 0xCD | 0xEF | 0x12 |
| Little | 0x12 | 0xEF | 0xCD | 0xAB |

**MIPS Register File**

MIPS has 32 locations by 32-bit register file.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Reg Number | Usage | Preserve |
| $zero | 0 | Const (read only) | n.a. |
| $at | 1 | reserved for assmblr | n.a. |
| $v0 - $v1 | 2 - 3 | return val | no |
| $a0 - $a3 | 4 - 7 | arguments | yes |
| $t0 - $t7 | 8 - 15 | temp | no |
| $s0 - $s7 | 16 - 23 | saved val | yes |
| $t8 - $t9 | 24 - 25 | temp | no |
| $gp | 28 | global pointer | yes |
| $sp | 29 | stack pointer | yes |
| $fp | 30 | frame pointer | yes |
| $ra | 31 | return addr | yes |

**MIPS R-format Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| op | rs | rt | rd | shamt | Funct |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

Example Compiled MIPS code:

1. add $t0, $s1, $s2

Compile to binary:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| spc | $s1 | $s2 | $t0 | 0 | add |
| 0 | 17 | 18 | 8 | 0 | 32 |
| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |

**MIPS I-format Instructions**

|  |  |  |  |
| --- | --- | --- | --- |
| op | rs | rt | const or addr |
| 6 bits | 5 bits | 5 bits | 16 bits |

**Arithmetic Operations**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| add | rd, | rs, | rt | rd = rs + rt; |
| sub | rd | rs | rt | rd = rs - rt; |
| addi | rd | rs | imm | rd = rs + imm; |

참고로, 앞으로 나오는 몇가지 연산자에 대하여, 접미사 i 를 붙여 $t 대신 imm을 사용 할 수 있다.

Example C code:

1. f = (g + h) – (i + j);
2. f += 1;
3. k = f;
4. k -= 1;

Compiled MIPS code: f…k in $s0…$s5

1. add $t0, $s1, $s2 # g + h
2. add $t1, $s3, $s4 # i + j
3. sub $s0, $t0, $t1
4. addi $s0, $s0, 1
5. add $s5, $s0, $zero # k = f + 0
6. addi $s5, $s0, -1

**Register Operands**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| lw | rt, | offs | (rs) | rt = rs[offs/4] |
| sw | rs, | offs | (rt) | rs[offs/4] = rt |

Example C code:

1. f[12] = g + h[8];

Compiled MIPS code: f…h in $s0…$s2

1. lw $t0, 32($s2)
2. add $t0, $s1, $t0
3. sw $t0, 48($s0)

**Logical Operations**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sll | rd, | rs, | rt | rd = rs << rt; |
| srl | rd, | rs, | rt | rd = rs >> rt; |
| and | rd, | rs, | rt | rd = rs & rt; |
| or | rd, | rs, | rt | rd = rs | rt; |
| nor | rd, | rs, | rt | rd = rs ~ rt; |

Example C code:

1. f = !f;

Compiled MIPS code: f in $s0

1. nor $s0, $s0, $zero

**Conditional Operations**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| beq | rs, | rt, | L | if(rs == rt) goto L; |
| bne | rs, | rt, | L | if(rs != rt) goto L; |
| j | L |  |  | goto L; |

Example C code:

1. if (i == j) f = g + h;
2. else f = g - h

Compiled MIPS code: f…j in $s0…$4

1. bne $s3, $s4, Else
2. add $s0, $s1, $2
3. j Exit
4. Else:
5. sub $s0, $s1, $2
6. Exit:

Example C code:

1. while(save[i] == k) i += 1

Compiled MIPS code: i in $s0, j in $s1, addr in $6

1. Loop:
2. sll $t1, $s0, 2 # $t1 = $s3 \* 2^2
3. add $t1, $t1, $s6 # $t1 = $t1 + addr
4. lw $t0, 0($t1)
5. bne $t0, $s1, Exit
6. addi $s0, $s0, 1
7. j Loop
8. Exit:

**Conditional Operations**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| slt | rd, | rs, | rt | if (rs < rt) rd = 1; else rd = 0; |

Example C code:

1. if (i < j) f = g + h;
2. else f = g - h

Compiled MIPS code: f…j in $s0…$4

1. slt $t0, $s3, $s4
2. bne $t0, $zero, Else
3. add $s0, $s1, $2
4. j Exit
5. Else:
6. sub $s0, $s1, $2
7. Exit:

**Six Steps in Execution of a Procedure**

1. Main routine(*caller*) places parameters in a place where the procedure(*callee*) can access them
2. *Caller* transfers control to the *callee*
3. *Callee* acquires the storage resources needed
4. *Callee* performs the desired task
5. *Callee* places the result value in a place where the *caller* can access it
6. *Callee* returns control to the *caller*

**Procedure Call Instructions**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| jal | L |  |  | $ra = L; goto L; |
| jr | $ra |  |  | goto $ra; |

Example C code:

1. int foo(int d, int e){
2. int j = d + e;
3. int k = d – e;
4. return j & k;
5. }
6. a = foo(b, c);

Compiled MIPS code: a…c in $s0…$s2, d…e in $a0…$a1

1. foo:
2. addi $sp, $sp, -8 # Moving stck pntr
3. sw $s0, 0($sp) # Store prev val
4. sw $s1, 4($sp)
5. add $s0, $a0, $a1 # Procedure
6. sub $s1, $a0, $a1
7. and $t0, $s0, $s1
8. add $v0, $t0, $zero # Result
9. lw $s0, 0($sp) # load prev val
10. lw $s1, 4($sp)
11. addi $sp, $sp, 8 # Moving stck pntr
12. jr $ra # return (cpy $ra to pc)
13. Main:
14. add $a0, $s1, $zero # Store argm
15. add $a0, $s2, $zero
16. jal Foo # Call procedure
17. add $s0, $v0, $zero # Store retrun val

Example C code:

1. int fact(int n){
2. if(n < 1) return f;
3. else return n \* fact(n-1);
4. }

Compiled MIPS code: argm in $a0, rst in $v0

1. fact:
2. addi $sp, $sp, -8 # Moving stck pntr
3. sw $ra, 0($sp) # Store retrn addr
4. sw $a0, 4($sp) # Store argmt
5. slti $t0, $a0, 1 # n < 1
6. beq $t0, $zero, Else # if n < 1
7. addi $v0, $zero, 1 # set retrn val 1
8. addi $sp, $sp, 8 # Moving stck pntr
9. jr $ra # return
10. Else:
11. addi $a0, $a0, -1
12. jal fact
13. lw $ra, 0($sp) # Load retrn addr
14. lw $a0, 4($sp) # Load argmt
15. addi $sp, $sp, 8 # Moving stck pntr
16. mul $v0, $a0, $v0
17. jr $ra # return

**Byte/Halfword Operations**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| lb | rt | offs | (rs) | $ra = L; goto L; |
| lh | rt | offs | (rs) |  |
| sb | rt | offs | (rs) |  |
| sh | rt | offs | (rs) | goto $ra; |

Example C code:

1. void strcpy(char x[], char y[]){
2. int i = 0;
3. while((x[i] = y[i] != '\0')
4. i += 1
5. };

Compiled MIPS code: x…y in $a0…$a1, i in $s0

1. strcpy:
2. addi $sp, $sp, -4
3. sw $s0, 0($sp)
4. add $s0, $zero, $zero # i = 0
5. Loop:
6. add $t0, $s0, $a0 # $t0 = x + i
7. add $t1, $s0, $a1 # $t1 = y + i
8. lbu $t2, 0($t1) # $t2 = y[i]
9. sb $t2, 0($t0) # x[i] = $t2
10. beq $t2, $zeroe, End
11. addi $s0, $s0, 1
12. j L1
13. End:
14. lw $0, 0($sp)
15. jr $ra

**MIPS 주소지정 방식**

1. Immediate addressing: addi
2. Register addressing: R-type
3. Base addressing: lw/sw
4. PC-relative addressing: branch
5. Pseudodirect addressing: jump

**프로그램 번역과 실행**

C program -> (Compiler) -> Assembly language program -> (Assembler) -> Object: Machine language module + Lib module -> (Linker) -> Executable: Machine language program -> (Loader) -> Memory

**API (Application Programing Language)**

응용 프로그램에서 사용할 수 있도록, 운영 체제나 프로그래밍 언어가 제공하는 기능을 제어할 수 있게 만든 인터페이스.

**Object Module**

* Header: described contents of object module.
* Text segment: translated instructions.
* Reloctions 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.

**Load from images 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
4. Set up arguments on stack
5. Initialize registers
6. Jump to startup routine

**MIPS Feature**

* MIPS is byte addressed
* Words are aligned in memory
* MIPS is Big Endian

**Activation record**

Saved argument -> Saved return address -> Saved saved register -> Local arrays and structures

The segment of the stack containing a procedure's saved registers and local variables is calldd activation record.