4 Instruction Set

4.1 Introduction to MIPS

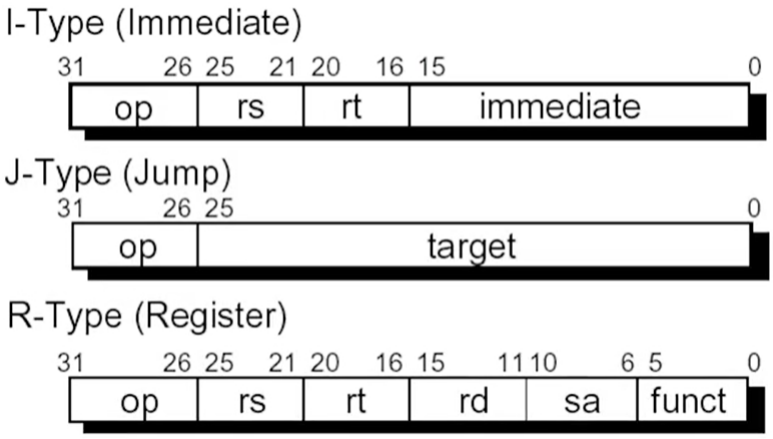
MIPS: Microprocessor without Interlocked Piped Stages.

- Principle: the simpler, the faster.
- Method: reduce the types and complexity of instruction.
- Feature:
  - Simple:
    - Fixed instruction length: 32 bit;
    - Less instructions, simple functions;
    - Simple Format: I-Type, J-Type, R-Type.
  - Few operand addressing mode:
    - base index + offset memory access;
    - immediate number access
    - register access
  - Use many general registers;
  - Only `load` or `store` can access the memory.

General Registers: 32 registers in total.

Reg. Number	Name	Function
0	`$zero`	Constant value 0
1	`$at`	Temporary variable for assembler
2-3	`$v0`, `$v1`	Return value, expression value
4-7	`$a0` ~ `$a3`	Arguments for function calls
8-15	`$t0` ~ `$t7`	Temporaries
16-23	`$s0` ~ `$s7`	Temporaries (*need to save in call*)
24-25	`$t8` ~ `$t9`	Temporaries
26-27	`$k0` ~ `$k1`	Retain for kernel use
28	`$gp`	Global pointer (*need to save in call*)
29	`$sp`	Stack pointer (*need to save in call*)
30	`$fp`	Frame pointer (*need to save in call*)
31	`$ra`	Return address (*need to save in call*)

Instruction Format: The first 6 bits are used to represent the specific instruction type. So we have $2^6 = 64$ specific types of instruction that can be distinguished by the first 6 bits. But there're more than 64 specific types of instructions in MIPS. Therefore, we use the last 6 bits of R-Type instructions as the extended code (扩展码) to represent the meanings of the instructions.



- *I-Type* (Immediate)
  - `rs` and `rt` are two operands, and `imm` is an immediate number.

> [*Example*]
>
> **MIPS Data Transfer Instruction (Type I)**
>
> ```
> lw $t0, 24 ($s3)
> sw $t0, 8 ($s3)
> ```
>
> `$s3` is a 32-bit address and `imm` (immediate number) is a 16-bit offset to the address.
>
> **MIPS Immediate Instruction (Type I)**
>
> ```
> andi $t0, $t1, 0xFF00  # $t0 = $t1 & ff00
> addi $sp, $sp, 4       # $sp = $sp + 4
> slti $t1, $s2, 15      # $t1 = 1 if $s2 < 15
> ```
>
> immediate number range: $[-2^{15}, 2^{15} - 1]$.

- *J-Type* (Jump)

Instruction	Operation	Notes
`j 10000`	`goto 10000`	Jump to `imm`
`jal 10000`	`$31 = pc + 4; goto 10000`	Jump and link

  - `jal` is used if we encounter a function call. It saves the next instruction after function into `$31` and jump to execute the function. After we come back from the function, we can know the instruction to be executed next by referring to the address in `$31`. When encounter `return` in the function, we actually implement `jr $31` to directly jump to the next instruction in the main function (`jr $31` is an R-Type instructions, it will be introduced later).
  - `j imm` is used if we need unconditional jumps. The destination address is calculated by left-shifting `imm` by 2 and combine with the highest 4 bits of `PC+4`. So instruction `j` will jump to the absolute offset（绝对偏移量） in the code segment.

    - Code segment: $2^{28} = 256M$ bytes;
    - The offset range of instruction `j`: $\sim 2^{28}$ bytes.
- *R-Type* (Register):
    - `op = 000000`, use `funct` to represent specific meanings (You can refer to the definition table of R-Type instructions).
    - `rs` and `rt` represent the source registers (two operands).
    - `rd` represents the destination register.
    - `sa` represents shift-length（位移量） in shift instructions（移位指令）, shift-length `sa = 31` at most. In other instructions, `sa = 00000`.

> [*Example*]
>
> ```
> add $t0, $t1, $t2 # $t0 = $t1 + $t2
> sub $t0, $t1, $t2 # $t0 = $t1 - $t2
> and $t0, $t1, $t2 # $t0 = $t1 & $t2
> sll $t2, $s0, 8   # $t2 = $s0 << 8 bits
>                # No rs, only rt, rd and sa.
> srl $t2, $s0, 8   # $t2 = $s0 >> 8 bits
>                # No rs, only rt, rd and sa.
> ```
>
> Left-shift and Right-shift are logical shift operations

    - `jr $31` is an *R-Type* instruction, it means `goto $31` (Jump to address in register `$31`).

4.2 MIPS Control Instructions

Conditional Transfer Instructions（条件转移指令）

Instruction	Operation	Notes
`beq $1, $2, 100`	`if($1 == $2) goto PC+4+100*4`	Jump if equal
`bne $1, $2, 100`	`if($1 != $2) goto PC+4+100*4`	Jump if not equal
`slt $1, $2, $3`	`if($2 < $3) $1 = 1; else $1 = 0;`	Set if less than
`slti $1, $2, 100`	`if($2 < 100) $1 = 1; else $1 = 0;`	Set if less than imm
`sltu $1, $2, $3`	`if($2 < $3) $1 = 1; else $1 = 0;`	Set if less than (unsigned)
`sltui $1, $2, 100`	`if($2 < 100) $1 = 1; else $1 = 0;`	Set if less than imm (unsigned)

- `bne` and `beq` are *I-Type Instructions*. `imm` multiplies with 4 because an instruction takes 4 bytes. We actually left-shift `imm` by 2 bits, and signed-extend it to 32 bits, and calculate the destination address. ($\sim \pm 2^{15}$-word or $\sim \pm 2^{17}$-byte offset)
- We can combine `slt` and `bne` to implement `blt` (jump if less than) using the following method.

```
slt $at, $s1, $s2     # $at set to 1 if $s1 < $s2
bne $at $zero, offset # if($at == 0) then jump
```

In MIPS, `blt`, `ble`, `bgt`, `bge` do not exist, because it can be implemented with the instructions above using the same method.

- If we want to jump further under conditions, we can combine `bne` and `beq` with `j` using the following method.

```
bne $s0, $s1, L2     # if($s0 != $s1) goto L2
j L1                 # goto L1
L2: ...
```

These instructions implement the instruction `beq $s0, $s1, L1`.

4.3 Function Call in MIPS

Function Call Instruction `jal imm`;

Function Return Instruction `jr $ra` (*R-Type*, `$ra` in `rs` district).

6 Steps of Function Execution

- The caller put the arguments in the place where callee can access. (`$a0` ~ `$a3`);
- Control transfer between caller and callee (`jal`);
- Callee apply for its own memory;
- Callee executes its instructions;
- Callee put the result in the place where caller can access (`$v0`, `$v1`).
- Control transfer between callee and caller (`jr $ra`).

Stack: If the arguments is more than the number of registers, then we need to use *stack*. We can use `$sp` to find the address of the stack.

- `push`: `$sp = $sp - 4`, and put data into `$sp`.
- `pop`: put data in `$sp` into the given address, and `$sp = $sp + 4`.
- `$sp` and `$fp` may be used as stack pointers: `$fp` may point to the base address （基地址） of a function call in some MIPS compilers.
- A function has its stack frame（栈帧）.

> [*Example*]
>
> ```
> int example(int g, int h, int i, int j) {
>     int f;
>     f = (g + h) - (i + j);
>     return f;
> }
> ```

```
example:
addi $sp, $sp, -4
sw $s0, 0 ($sp)       ; local variable f in $s0 and
                      ; push $s0 into stack to protect
                      ; the original data in it
add $t0, $a0, $a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
add $v0, $s0, $zero   ; result in $v0
lw  $s0, 0($sp)       ; recover $s0
addi $sp, $sp, 4      ; pop
jr $ra                ; return
```

> We need a stack to protect the data in registers when executes an function, (In this example, we protect the original data in `$s0`), especially in recursive functions and nested functions.

Registers

- Need to store in function calls: `$ra`, `$sp`, `$gp`, `$fp`, `$s0` ~ `$s7`, `$a0` ~ `a3`.
- No need to store in function calls: `$v0`, `$v1`, `$t0` ~ `$t9`.

> **Variables in C Programming Language**
>
> - static variables（静态变量）: store in the entry/exit of process
> - auto variables（自动变量）: local variables, destroy when exit the function.
>
> *We can use stack to store the auto variables in function calls.*

4.4 Introduction to Intel x86 Instructions

> Intel x86 Instruction Set: **CISC**.
>
> MIPS: **RISC**

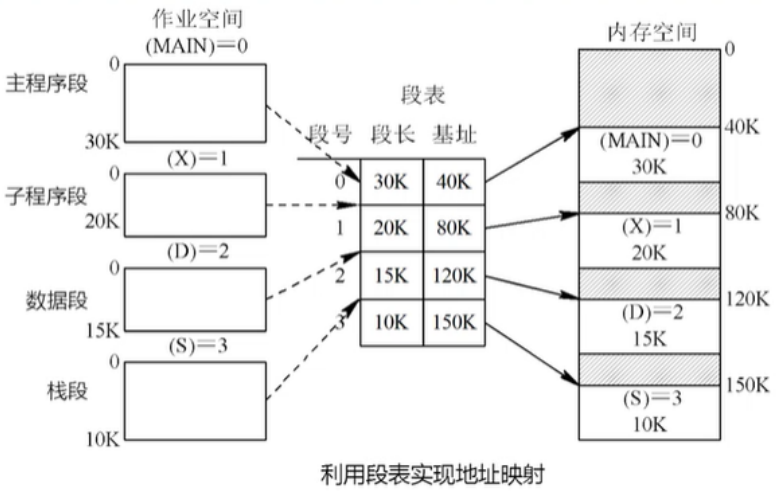
Registers in 8086: refer to notes in EI209 Chap 3.

Extension of Registers: `AX` extends to `EAX` + `AX`, where `AX` is the lower 16 bits of register and `EAX` is the higher 16 bits; and extends to `RAX` + `EAX` + `AX`, where `RAX` is the higher 32 bits of register, and `EAX` + `AX` is the lower 32 bits. Therefore, the new version is compatible with the old version.

Addressing Mode in 8086: refer to notes in EI209 Chap 3.

Modes in x86

- Real mode（实模式）: (8086) there is no OS, so the program can modify registers freely, which is unsafe.
- Protection mode（保护模式）: (80386+) OS implement virtual memory and make the system safer.



利用段表实现地址映射

- When access the memory, we need to know the address of a segment (begin address, 基址), the length of a segment and the permission of the segment（访问权限）. If an access is invalid, then the OS forbids it to access the memory, which makes the system safer.
- We can use the *segment table*（段表）in memory to store these information.

Two more registers: `GDTR` and `LDTR`: `GDTR` stores the base address of segment table in memory.

This time, `CS` refers to the offset address in segment table, then `GDTR[CS>>3].BaseAddr + EIP` is the physical address (if the data is not stored in pages). (here `CS >> 3` is because each line of the segment table contains 8 bytes.)

- In x64, we use pages to manage the memory, but an nearly-empty segment table retains to make the system compatible. Therefore, `GDTR[CS].BaseAddr + RIP` is the logical address. `BaseAddr` in segment table are all 0, so `CS:RIP` (or `RIP` itself) is the logical address. To find out the physical address, we need to access *page table* to get it.

4.5 Move Instructions in Intel x86

Mainly focus on the transfer instructions in x64.

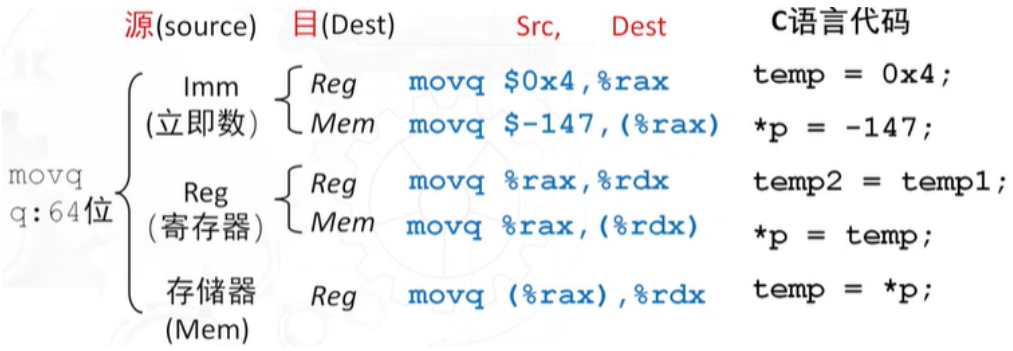
Suffices of Instructions

- `-q` means the operands are 64-bit (8 bytes).
- `-l` means the operands are 32-bit (4 bytes).
- `-w` means the operands are 16-bit (2 bytes).
- `-b` means the operands are 8-bit (1 byte).
- `-(s/z)(1)(2)` means casting from `(1)` to `(2)`, where `s` means sign extension and `z` means unsigned extension. `(1)` and `(2)` can be chosen from `q`, `l`, `w`, `b`.

> [*Example*]

```
movq %rax, %rdx
movl %eax, %edx
movsbl %al, (%rdx)
movslq %edx, %rax
```

`MOV` **Instructions**



Tips: cannot move a data from memory to memory!

Addressing Mode

- Register: `(R)`, access the address stored in register `R`.
- Base + Offset: `D(R)`, access the address stored in register `R` with an offset `D` (`D + Reg[R]`).
- General: `D(Rb, Ri, S)`, access `Mem[Reg[Rb] + S*Reg[Ri] + D]`.
  - `D`: offset;
  - `Rb`: Base address register (any register);
  - `Ri`: Index address register (any register except `%rsp`);
  - `S`: Scale factor (1, 2, 4, 8) (because of alignment).

`lea` **Instructions**: Transfer the address of `src` to the register `dst`, only calculate the address, but not access the pointing memory.

> [*Example*]
>
> ```
> leaq (%rbx, %rcx, 4), %rdx
> movl (%rbx, %rcx, 4), %eax
> ```

Special Trick: Use `lea` to implement arithmetic operations

> [*Example*]
>
> ```
> long long m12 (long long x) {
>  return x * 12;
> }
> ```
>
> ```
> leaq (%rdi, %rdi, 2), %rax  # t <- x + x * 2
> salq $2, %rax               $ return t << 2
> ```
>
> Sometimes we can view the 'address' as a number.

4.6 Arithmetic Instructions in Intel x86

Double-operands Instructions

- `add`, `sub` instructions: perform addition and subtraction operations, and do not distinguish unsigned and signed.

Format	Meanings
`add src, dest`	`dest = dest + src`
`sub src, dest`	`dest = dest - src`

  Tips: cannot perform operations between two memory data; can have suffices such as `addq`.

- Other instructions: such as `mul`, `sal`, `sar`, `shr`, `xor`, `and`, `or` (all need to add suffices such as `mulq` and `salq`)
- `mul` can have a prefix `i-`, that is, `imul`, means the multiplication of signed numbers.

Single-operands Instructions: `inc`, `dec`, `neg`, `not`, etc. All of them need to add suffix such as `incq`.

> [*Example*]
>
> ```
> long long arith(long long x, long long y, long long z) {
>     long long t1 = x + y;
>     long long t2 = z + t1;
>     long long t3 = x + 4;
>     long long t4 = y * 48;
>     long long t5 = t3 + t4;
>     long long rval = t2 * t5;
>     return rval;
> }
> ```
>
> ```
> # %rdi: x, %rsi: y, %rdx: z, %rax: t1 t2 rval, %rdx: t4, %rcx: t5
> arith:
> leaq (%rdi, %rsi), %rax          # t1
> addq %rdx, %rax                  # t2
> leaq (%rsi, %rsi, 2), %rdx       # y * 3
> salq $4, %rdx                    # y * 48 = (y * 3) << 4
> leaq 4(%rdi, %rdx), %rcx         # t5 = t3 + t4 = x + 4 + t4
> imulq %rcx, %rax                 # rval
> ret
> ```

4.7 Jump Instructions in Intel x86

Unconditional Jump: `JMP D`, jump to instruction in address `D` to continue executing unconditionally.

- `JMP SHORT LABEL`, (2-bytes instruction) transfer to the instruction labeled `LABEL`. The offset is in range $[-128, 127]$ bytes.
- `JMP NEAR PTR LABEL`, (3-bytes instructions) transfer to the instruction labeled `LABEL`. The offset is in range $[-32K, 32K-1]$ bytes.
- `JMP EAX` (indirect jump), the destination address is in register `EAX`;
- `JMP [ESI]` (indirect jump), the destination address is in memory (address `ESI` in memory).

Conditional Jump: `j(x)` instructions, where `(x)` is the condition.

Inst.	*Condition*	*Explanation*
`jmp`	`1`	*Unconditional*
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

The introductions of registers `CF`, `ZF`, `SF`, and `OF` are in EI209 Chap 3.

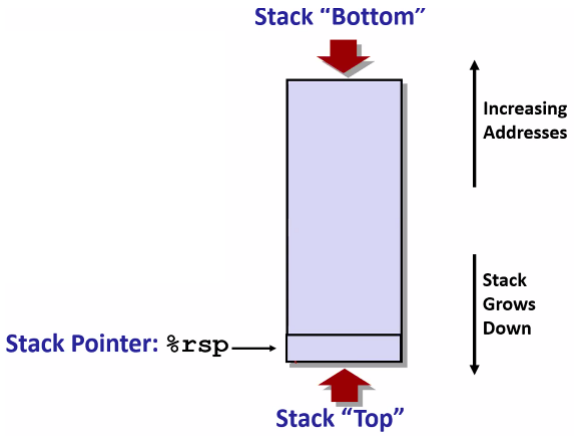
Tips: `CF` for unsigned, `SF` and `OF` for signed.

Use `cmp b,a` instruction (with suffix) to set the sign registers: perform `a-b` and set the sign registers, but not store the result into registers.

> [*Example*]
>
> ```
> long long absdiff(long long x, long long y) {
>     long long result;
>     if (x > y)
>         result = x - y;
>     else
>         result = y - x;
>     return result;
> }
> ```
>
> ```
> absdiff:
> cmpq %rsi, %rdi     # x - y (x in %rdi) and y in %rsi.
> jle .L4
> movq %rdi, %rax     # %rax = %rdi = x
> subq %rsi, %rax     # %rax = %rax - %rsi = x - y
> ret                 # return x - y
>
> .L4:                # x <= y
> movq %rsi, %rax     # %rax = %rsi = y
> subq %rdi, %rax     # %rax = %rax - %rdi = y - x
> ret                 # return y - x
> ```

4.8 Function Call in Intel x86-64

x86-64 Stack: Region of memory managed with stack discipline, and grows towards lower address.



- Register `%rsp` contains lowest stack address, also address of "top" element.
- Push: `%rsp = %rsp - 8` (`pushq src`);
- Pop: `%rsp = %rsp + 8` (`popq dest`).

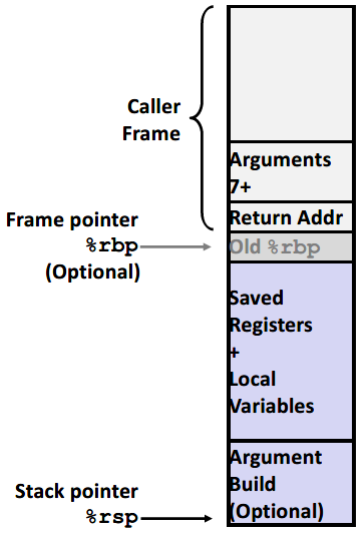
x86-64 Linux Register Usage

- `%rax` for return value, also caller-saved, it can be modified by procedure.
- `%rdi, ..., %r9`: arguments, also caller-saved, it can be modified by procedure.
- `%r10, %r11`: caller-saved, can be modified by procedure.

- `%rbx,%r12,%r13,%r14`: callee-saved, callee must save & restore.
- `%rbp`: callee-saved, callee must save & restore. It may be used as frame pointer and it can mix & match.

- `%rsp`: special form of callee save. It is restored to original value upon exit from procedure.

Stack Frame: Contains.



- Arguments (*Argument-build*);
- Local variables will be kept in stack frame (If it can't be kept in registers);
- Saved register context;
- Old frame pointer (optional);
- Caller stack frame contains return address which is pushed by `call` instruction, and arguments for this call.

4.9 Buffer Overflow

Buffer Overflow: exceed the memory size allocated for an array. It's the first technical cause of security vulnerabilities. Its most common from is *uncheck lengths on string inputs*, particularly for bounded character arrays on the stack.

> [*Example*]
>
> C function `gets()` do not specify limit on number of characters to read.
>
> Some other functions `strcpy`, `strcat` also have this problem, which is dangerous.
>
> ```
> void echo() {
>     char buf[4];    /* Way to small */
>     gets(buf);
>     puts(buf);
> }
> void call_echo() {
>     echo();
> }
> ```
>
> If we input a string `0123456789012345678901 23`, maybe the result is still correct; but if we input a string `01234567890123456789012 34`, maybe the result will be segment fault!
>
> Why? The stack segment has a length of 24. And the data is stored with little endian, so if the buffer is overflowed, the address will come to the parent function's stack frame!

Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines.

Solving Methods

- Avoid overflow vulnerabilities in code. For instance, `fgets` instead of `gets`, `strncpy` for `strcpy`.
- **System-level protections** can help. **Randomized stack offsets**（栈随机化） and **Non-executable code segments**（限制可执行代码区域）(*stack marked as non-executable*).
- **Stack Canaries** can help. Place specify value (*canary*) on stack just beyond buffer, and check for corruption before exiting function. (GCC implementation: `-fstack-protector`).

Return-Oriented Programming

- Challenge (for hackers)
  - Stack randomization makes it hard to predict buffer location;
  - Marking stack nonexecutable makes it hard to insert binary code.
- Alternative Strategy
  - Use existing code;
  - String together fragments to achieve overall desired outcome;
  - *Does not overcome stack canaries*.
- Construct program from *gadgets*
  - Sequence of instructions ending in `ret`, encoded by single byte `0xc3`.
  - Code position fixed from run to run.
  - Conde is executable.