CC3301
Programación de software de sistemas

Assembler Risc-V

Assembler Risc-V

Risc-V es una familia de arquitecturas libres de pago de licencias:

- Cualquier empresa puede implementar Risc-V y no pagar licencia a nadie (~~Intel~~, ~~ARM~~)
- La ventaja es que dispone de gcc y linux
- Puede agregar nuevas instrucciones, sin tener que publicar su implementación y aún lucrar sin pagar licencia.
- Diversas implementaciones:
  - Propietarias (hay que pagar licencia)
  - Open-source (hay que publicar los cambios, a la Linux)
- Múltiples variantes: 32 o 64 bits, con o sin multiplicación/división, con o sin punto flotante
- ¿Por qué Risc-V y no x86? Porque es mucho más simple.

¿Cómo compilar/ejecutar en x86?

- Gnu toolchain incluye:
  - compilador de C que genera risc-v
  - depurador gdb para risc-v
  - Emulador para x86
- qemu-riscv32 o qemu-riscv64 son emuladores que ejecutan binarios para risc-v traduciendo las instrucciones a x86 *on the fly*
- Se puede usar ddd con qemu-riscv32 para depurar los programas en assembler risc-v
- Descargar el toolchain y qemu-riscv32 desde:

  https://users.dcc.uchile.cl/~lmateu/CC4301

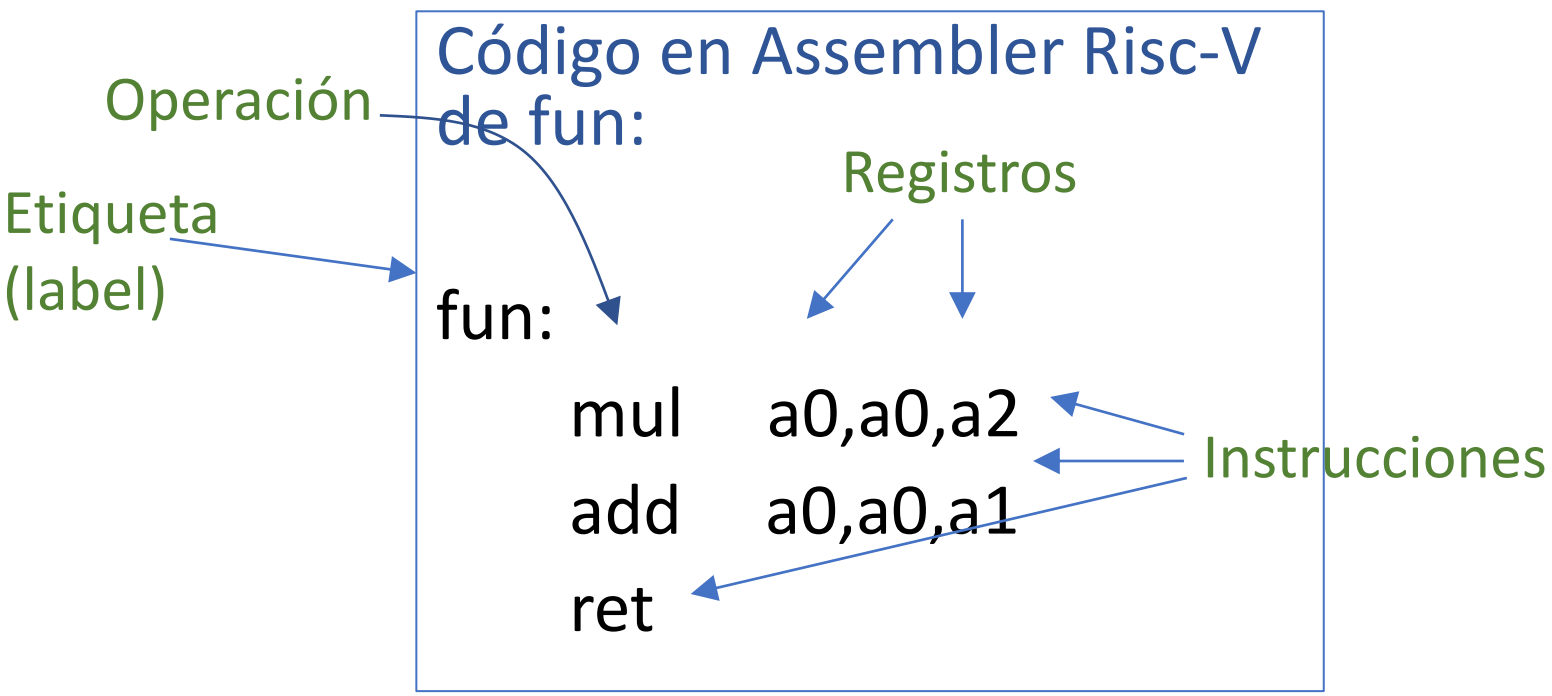
Una primera función

Código en C en *fun.c*:

```
int  fun(int a, int b, int x) {
   return a*x+b;
}
```

Código en C en test-*fun.c*:

```
int main() {
  int f= fun(2, 5, 10);
  printf("%d\n", f);
  return 0;
}
```



¿Como probar?

- Compilar para debugging: make test-fun
- Ejecutar: **qemu-riscv32** test-fun
- Generar Código en assembler: make fun.s
- Compilar optimizado: make -B test-fun.opt
- Depurar con ddd: make test-fun.ddd

Arquitectura de Risc-V

- 32 registros enteros: x0 a x31 que también tienes alias
  - x0: zero es siempre 0
  - x10-x17: a0-a7 son los argumentos de una función
  - x5-x7, x28-x31: t0-t6 son valores temporales
  - x8-x9, x18-x27: s0-s11 son los registros resguardados
  - x1: ra es la dirección de retorno de una función
  - x2: sp es el puntero a la pila
  - x3: gp es el puntero a las variables globales
  - x4: tp es el puntero a las variables locales de un thread
- Las instrucciones se codifican en 32 bits
- Excepcionalmente en 16 bits
- Operaciones aritmético/lógicas: add, sub, mul, div, and, or, xor, sll, srl, sra, etc.
  - Ejemplos: add a0, a2, a3 addi t1, a1, 5 mul t4, a3, s2
- Accesos a memoria: lw, lb, lh, lbu, lhu, sw, sb, sh
  - Ejemplo: lw t4, 0(t1)
- Saltos condicionales: beq, bne, blt, bge, bltu, bgeu
  - Ejemplo: bge t1, t0, else
- Llamadas a funciones y retorno: jal, jalr, ret

Compilación de if

```
if (x<=y)
  inst1
else
  inst2
```

```
    … colocar x en t0 …
    … colocar y en t1 …
    bge t1, t0, else
    … inst1 …
    j cont      # bge zero,zero
else:
    … inst2 …
cont:
```

Compilación de for

```
for (int i=0; i<n; i++) {
   … inst …
}
```

Equivalente a:

```
int i= 0;
while (i<n) {
  … inst …
  i++;
}
```

```
      li t0, 0
      … colocar n en t1 …
      j cond
ini:
      … inst …
      addi t0, t0, 1
cond:
      blt t0, t1, ini
      …
```

Compilación a[i]

```
int a[]= { … };
int x= a[i];
```

```
… colocar a en t0 …
… colocar i en t1 …
slli t2, t1, 2        # i*4
add t2, t2, t1    #  a+i*4
lw t4, 0(t1)       # a[i]
```