CC4301 Arquitectura de computadores

Assembler Risc-V

2da clase

Resumen clase anterior

Uso del Makefile

- Ver código generado en assembler optimizado:

  make *prog*.s Ejemplo: make fun.s

- Depurar con ddd:

  make *binario*.ddd Ejemplo: make test-fun.ddd

Assembler Risc-V

- 32 registros enteros: a0-a7, t0-t6, s0-s11, zero
- Formato instrucciones aritmético/lógicas:

  Operación registro-destino, registro-op1, registro-op2

  Ejemplo: add a0, t3, s9

  Operaciones: add, sub, mul, div, and, or, xor, sll, srl, sra
- Formato instrucciones con valor inmediato:

  *Operación***i** *registro-destino, registro-op1, valor*

  Ejemplo: add**i** s4, a2, 6 ~~muli divi~~

Continuación resumen clase anterior

- Formato instrucciones de salto condicional:

  B*condición reg1, reg2, etiqueta-destino*

  Ejemplo: bge t1, s3, continuación

  Condición: beq, bne, blt, bge, bltu, bgeu

- Salto incondicional: **J** *etiqueta-destino*

  Es equivalente a: bge zero, zero, *etiqueta-destino*

- Llamadas a funciones: JAL ra, *nombre-función*
- Llamada a dirección de función: JALR ra, *offset(reg)*
- Retorno de función: RET

  Es equivalente a: JALR zero, 0(ra)

- Pseudo instrucciones:

  mv s0, t1 equivalente a add s0, t1, zero

  li a3, -4 equivalente a addi a3, zero, -4

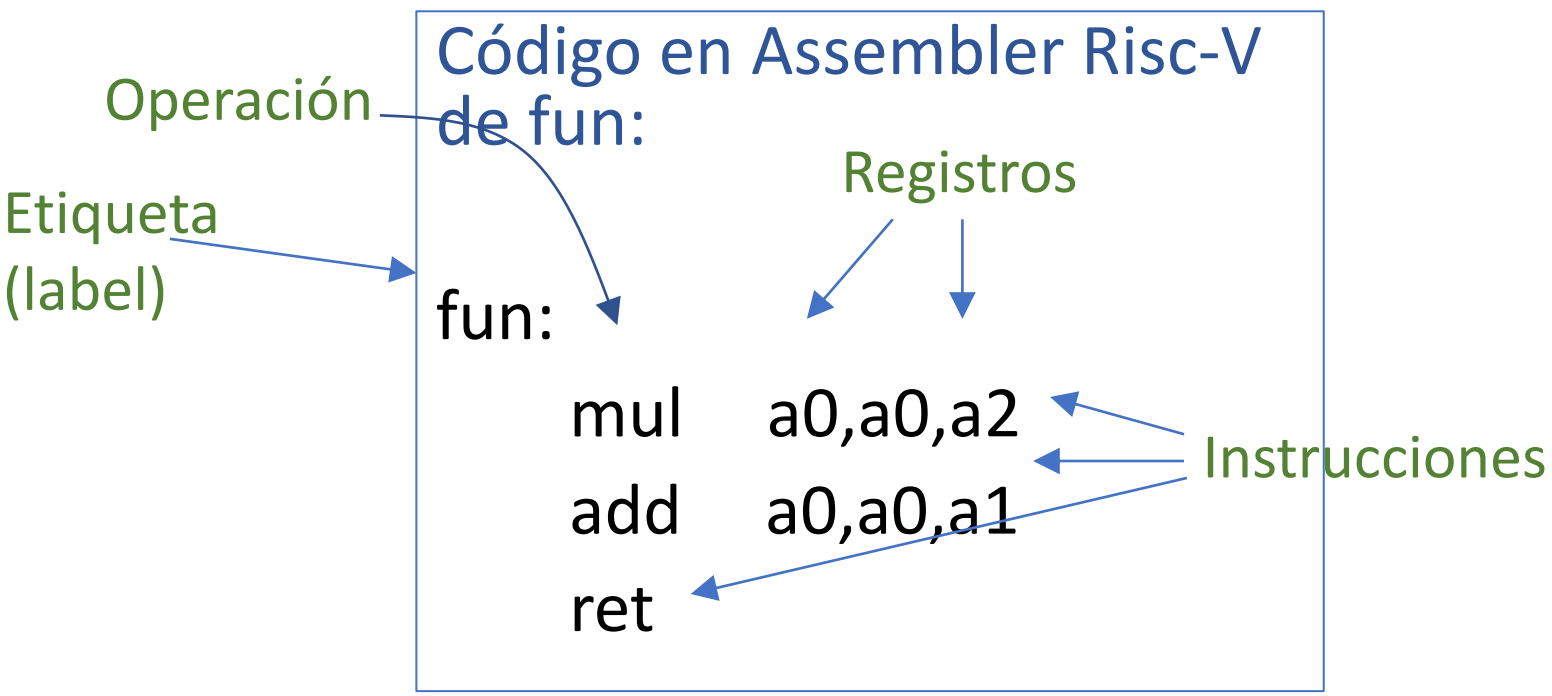
Compilación de expresiones aritméticas

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;
}
```



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] y p->next

```
int a[]= { … };
…
int x= a[i];

struct nodo {
    int x;
    struct nodo *next;
}
…
p->next
```

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

… colocar p en t3 …
lw  t5, 4(t3)       # p->next
```

Compilación de llamadas a funciones

x= fun(a, b, c);

…

… colocar a en a0 …

… colocar b en a1 …

… colocar c en a2 …

jal ra, fun

… x es a0 …

Compilación de funciones simples

*Función **no** llama a otras funciones*

```
int fun(…) {
  …
  return x;
}
```

```
fun:
    … parámetros en a0, a1, a2, …
    … colocar valor retorno en a0 …
    ret     # jr ra  o jalr zero, 0(ra)
```

Compilación de funciones que llaman a otras funciones

Función sí llama a otras funciones

```
int g(…) {
  …
  x= fun(…);
  …
  return x;
}
```

¡Se necesita resguardar registros en memoria!

```
g:
    … parámetros en a0, a1, a2, …
    addi  sp, sp, -16
    … resguardar registros …
    sw    ra, 12(sp)
    … colocar valor retorno en a0 …
    …
    jal   ra, fun
    …
    … restaurar registros …
    lw    ra, 12(sp)
    addi  sp, sp, 16
    ret
```

Crea registro de activación en la pila

Libera registro de activación

Registro sp siempre está alineado en 16 bytes

Compilación de tipo char

- *En Risc-V y ARM el tipo char es sin signo, en x86 es con signo*
- Al cargar una variable en memoria de tipo char siempre se convierte al tamaño del registro extendiendo con ceros:

```
char *p;
…                          … coloca p en a1 …
char c= *p;                lbu  a5, 0(a1);
```

- Al guardar una expresión de tipo char siempre se trunca a 8 bits:

```
*p = x;                    … coloca x en t4 …
                           sb    t4, 0(a1);
```

- Al retornar una expresión de tipo char siempre se colocan en 0 los bits más significativos

```
return x;                  … coloca x en t4 …
                           andi    a0, t4, 0xff
                           ret
```

Compilación de tipo signed char

- Al cargar una variable en memoria de tipo signed char siempre se convierte al tamaño del registro extendiendo con el signo:

```
signed char *p;
…                                   … coloca p en a1 …
signed char c= *p;                  lb   a5, 0(a1);
```

- Al guardar una expresión de tipo signed char siempre se trunca a 8 bits:

```
*p = x;                             … coloca x en t4 …
                                    sb    t4, 0(a1);
```

- Al retornar una expresión de tipo signed char siempre se extienden con signo los 8 bits menos significativos

```
return x;                           … coloca x en t4 …
                                    slli     a0, t4, 24
                                    srai   a0, a0, 24  # extiende signo
                                    ret
```