Conception et exploitation des processeurs

Frédéric Pétrot

Équipe pédagogique :
Julie Dumas, Claire Maiza, Olivier Muller, Frédéric Pétrot, Lionel Rieg (resp.),
Manu Selva et Sebastien Viardot

Année universitaire 2021-2022

C1 Présentation du projet CEP et rappels de VHDL
C2 Chaîne de compilation et assembleur RISC-V
C3 Conventions pour les appels de fonctions
C4 Gestion des interruptions par le logiciel

Sommaire

Introduction

Étude du langage d'assemblage du processeur RISC-V

RISC-V

- ISA RISC : Reduced Instruction Set Computer
  jeu d'instruction inspiré du MIPS, ISA RISC emblématique
  conception récente, forgé à Berkeley
  c.f. `riscv.org`
- existe en 32 et 64 bits, avec ou sans instructions sur les flottants, etc
- dans ce cours : sous-ensemble rv32im

Introduction

RISC-V rv32i

- jeu d'instruction de base minimaliste, 37 instructions dans le rv32i [a].
- destiné à être la cible d'un compilateur
- instruction de taille fixe : 32 bits
- 32 registres à usage général de 32 bits
- bus d'adresse de 32 bits
- un mode d'adressage unique, architecture *load/store*
- opérandes = registres, sauf pour *load/store*
- représentation des nombres en *little-endian*

a. Mais plus de 200 avec les flottants, les instructions compressées, les instructions système, etc

Sommaire

Rappel sur la chaîne de compilation

Cible RISC-V sur hôte x86-64 : développement *croisé*

Compilation

`riscv32-unknown-elf-gcc -g -S hello.c` produit `hello.s`
`riscv32-unknown-elf-gcc -g -S world.c` produit `world.s`

Assemblage

`riscv32-unknown-elf-as -o hello.o hello.s` produit `hello.o`
`riscv32-unknown-elf-as -o world.o world.s` produit `world.o`

Édition de liens

`riscv32-unknown-elf-ld -T cep.ld -o gogogo hello.o world.o`
produit l'exécutable `gogogo`

En pratique on appelle toujours `riscv32-unknown-elf-gcc` !

Rappel sur la chaîne de compilation

Désassemblage et inspection des fichier elf

`riscv32-unknown-elf-objdump -d hello.o` désassemble `hello.o`
`riscv32-unknown-elf-objdump -d gogogo` désassemble `gogogo`
`riscv32-unknown-elf-readelf -a gogogo` informations sur `gogogo`

Placement des objets en mémoire

`riscv32-unknown-elf-nm hello.o` informe sur les adresses dans `hello.o`

Pour plus d'informations

`RTFM`, référez vous aux excellents manuels d'Unix : `man objdump`, par ex.

Exécutable en mémoire

Exécutable constitué de :

Code

- directive `.text`
- en lecture seulement
- commence « en bas » de la mémoire

Données connues statiquement

- directive `.data`
  variables globales initialisées au lancement
  en lecture/écriture
- directive `.bss`
  variables globales non explicitement
  initialisées, initialisées à zéro au lancement
  en lecture/écriture

Optimisation :
données de « petite taille » dans sections spéciales
`.sdata` et `.sbss`

Exécutable en mémoire

Données constantes connues statiquement

- Directive `.rodata` (sur certains systèmes)
  Constantes initialisées au lancement, en lecture seulement

Données allouées dynamiquement : tas (ou *heap*)

- variables globales dont les adresses sont définies à l'exécution
- commence « à la fin » du programme, « va » dans le sens des adresses croissantes
- allocateur logiciel nécessaire pour gérer le tas [a]
  en C :
  `malloc` fournit des adresses libres
  `free` libère les adresses inutiles

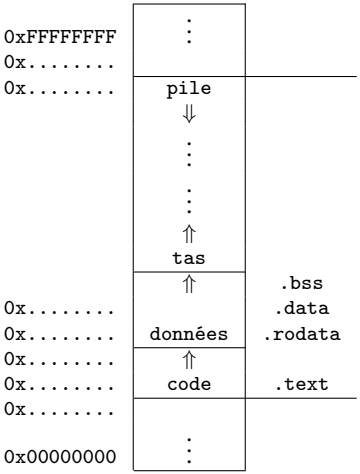
a. En réalité c'est un peu plus compliqué, *c.f.* `man sbrk` et l'excellent cours PCSEA en deuxième année

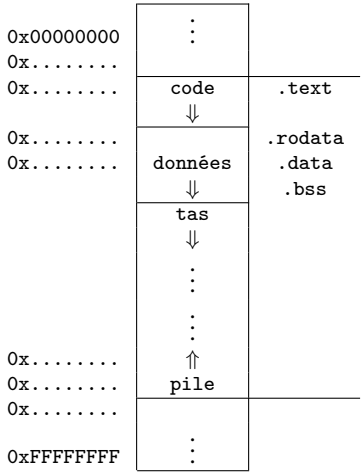
Variables automatiques : pile (ou *stack*)

- gérée par le logiciel lors des *appels de fonctions* (c.f. cours n°3)
- commence « à la fin » de la mémoire, « va » dans le sens des adresses décroissantes

Résumé

Même information, mais représentation du bas vers le haut ou inversement !





Rappel sur les types de données

Types entiers de tailles fixes

Types C99 (`stdint.h`)	Taille en octets
`int64_t`, `uint64_t`	8
`int32_t`, `uint32_t`	4
`int16_t`, `uint16_t`	2
`int8_t`, `uint8_t`	1

Cas des pointeurs (adresses)

Dépendant de l'ISA :

`RV32I` = 4 octets (`x86-64` = 8 octets)

indépendant type donnée pointée (ex. : `char *`, `int64_t *`, ...)!

Attention !

Taille des types standards du C K&R (`int`, `long`, ...) varie selon l'ISA !

Sommaire

ISA RISC-V

Registre 32 bits	Remarque
`x0`	registre poubelle (peut être écrit, zéro en lecture)
`x1-x31`	non spécialisés

Adresses

- sur 32 bits, de `0x00000000` à `0xffffffff`

Données

- sur 8, 16 ou 32 bits
- seules les instructions de chargement ou de stockage mémoire contiennent information de taille
- organisation mémoire *little endian*

ABI MIPS

Convention d'usage des registres

Nom matériel	Nom logiciel	Signification	Préservé lors des appels ?
x0	zero	Zéro	Oui (Toujours zéro)
x1	ra	Adresse de retour	Non
x2	sp	Pointeur de pile	Oui
x3	gp	Pointeur global	Ne pas utiliser
x4	tp	Pointeur de tâche	Ne pas utiliser
x5-x7	t0-t2	Registres temporaires	Non
x8-x9	s0-s1	Registres préservés	Oui
x10-x17	a0-a7	Registres arguments	Non
x18-x27	s2-s11	Registres préservés	Oui
x28-x31	t3-t6	Registres temporaires	Non

- utilisation détaillée précisée dans le cours n°3
- usage principalement des registres t_i dans les premiers programmes

Pseudo-instructions

Macros : pseudo-instructions facilitant l'écriture du code

- syntaxe simplifiée pour une petite séquence d'instructions
- généralement expansées en une ou deux instructions par l'assembleur

Branchements et sauts

```
beqz rs, symbole => branche si rs = zero   bgtz rs, symbole     => branche si rs > zero
bnez rs, symbole => branche si rs ≠ zero   bgt  rs, rt, symbole => branche si rs > rt
blez rs, symbole => branche si rs ≤ zero   ble  rs, rt, symbole => branche si rs ≤ rt
bgez rs, symbole => branche si rs ≥ zero   bgtu rs, rt, symbole => branche si rs u> rt
bltz rs, symbole => branche si rs < zero   bleu rs, rt, symbole => branche si rs u≤ rt

j    symbole          => branche inconditionelle, adresse connue statiquement
jal  symbole          => appel de fonction, adresse connue statiquement
call symbole          => appel de fonction, adresse connue statiquement
jr   rs               => branche inconditionelle, adresse dans registre
jalr rs               => appel de fonction, adresse dans registre
ret                   => retour de fonction
```

Pseudo-instructions

Chargements de registres et constantes 32 bits

```
mv rd, rs          => [move] copie rs dans rd
li rd, 0xdeadbeef => [load immediate] charge la constante dans rd
la rd, symbole     => [load address] charge l'adresse du symbole dans rd
```

Nombre d'instructions produites dépend de la taille des constantes/symboles

Accès à la mémoire

```
lw rd, var     => récupère dans rd le contenu de la variable var
sw rd, var, rt => stocke dans la variable var la valeur de rd
                  (rt est nécessaire et écrasé par la macro)
```

Autres

```
not rd, rs => Complement à 1 de rs
neg rd, rs => Opposé de rs
nop        => instruction nulle
```

Et encore quelques autres!

Directives

Définition variées pour déclarer des objets

Un bon exemple vaut mieux qu'un long discours, ...

```
      .globl func
      .equ uart_base, 0x10013000
      .text
func : add ...
      .data
bvar : .byte  1, 2, 3, 4, 5
svar : .short 1000, 2000, 3000
wvar : .word  uart_base, 0xdeadbeef
xvar : .word  func, svar
.comm ptr, 4
```

- `.globl symb` rend `symb` visible à l'éditeur de lien
- `label :` déclare le symbole `label` et lui donne l'adresse courante
  le label est une constante
- `.byte expr`, `.short expr`, `.word expr` crée à l'adresse courante des données du type considéré avec les valeurs fournies
- `.comm symb, taille` déclare le symbole `symb` avec la taille précisée
  le symbole est une variable

Relocations

(Pour la culture)
Placement des objets en mémoire pour création exécutable
Décidé lors de l'édition de liens

Relocations pour la résolution des adresses

Notation assembleur	Description	Instruction / Macro
`%hi(symbol)`	absolue	`lui`
`%lo(symbol)`	absolue	`load, store, add`
`%pcrel_hi(symbol)`	relative à PC	`auipc`
`%pcrel_lo(label)`	relative à PC	`load, store, add`

Autres possibles, pour les tâches parallèles ou les langages objets

Par défaut relatif à PC

Sommaire

if ...

Conditions

Condition assembleur complémentaire de celle exprimée au départ

if

```
if (x == 0) {
   x = 12;
}
...
```

```
if:    bnez t0, endif # t0 est x
       li   t0, 12
endif: ...
```

if ... else ...

```
if (x == 0) {
   x = 12;
} else {
   x = 24;
}
...
```

```
if:    bnez t0, else
       li   t0, 12
       j    endif
else:  li   t0, 24
endif: ...
```

if ... else ... if ... else ...

if ... else ...

```
if (x == 0) {
   x = 12;
} else if (x > y) {
   x = 24;
} else {
   x = 36;
}
...
```

```
if:     bnez t1, elseif   # t1 est x
        li   t1, 12
        j    endif
elseif: ble  t1, t2, else # t2 est y
        li   t1, 24
        j    endif
else:   li   t1, 36
endif: ...
```

Condition compliquée

évaluation paresseuse ⇒ utilisation de branchements successifs

```
if ((a > 0 && b < 0xbad)
    || c == d) {
   ...
}
```

```
if   :  blez  t0, L0        # t0 est a
        slti  t6, t1,0xbad  # t1 est b
        bnez  t6, L1
L0   :  bne   t2, t3, endif # t2 est c
                            # t3 est d
L1   : ...
endif:
```

while/for

En C, le `for` est un `while`

`while`

```
while (i > 0) {                    loop    : blez t0, endloop  # t0 est i
   ...                                       ...
   i--;                                      addi t0, t0, -1
}                                            j    loop
...                                endloop : ....
```

`for`, autre manière d'écrire un `while` en C

```
for (i = 0; i < n; i++) {                    li   t0, 0            # t0 est i
   ...                             loop    : bge  t0, t1, endloop  # t1 est n
}                                            ...
...                                          addi t0, t0, 1
                                             j    loop
                                   endloop : ....
```

switch

Switch

Après la section suivante !
Utilise une « table de saut » stockée en mémoire de données

Sommaire

Modes d'adressage

Ou comment accéder aux données

Constantes : placées *dans* l'instruction elle-même, *immediat*

C :

```
int32_t mcdo = 0x8badf00d;
```

RISC-V :

```
li t6, 0x8badf00d
```

Variables scalaires : cases mémoire

Identifiées par leur *adresse*

C :

```
int32_t dice = 420;
void roll(void)
{
   ...
   dice++;
   ...
}
```

RISC-V : indirect registre

```
   .data
dice: .word 420
   .text
roll:
   ...
   la   t0, dice
   lw   t1, (t0)
   addi t1, t1, 1
   sw   t1, (t0)
   ...
   ret
```

Modes d'adressage

Horreur ! Variable identifiée par son adresse !

Variables *de tous types* : case(s) mémoire

Si connue statiquement (constante dans le code) :
(en supposant qu'on laisse les variables automatiques `one`, `five` et `nine` dans des registres)

C :

```
int32_t p[]  = {
    1,  3,  0,  7,
    9, 11, 13, 17};
void gloups(void)
{
    int32_t  one  = p[0];
    int32_t  nine = p[4];
    int32_t *five = &p[2];
    *five = 5;
    ...
}
```

RISC-V : indirect registre + déplacement (sur 12 bits)

```
  .data
p:.word 1, 3, 0, 7, 9, 11, 13, 17
  .text
gloups:
  ...
  la   t0, p       # t0 est &p[0]
  lw   t1, 0*4(t0) # t1 est one
  lw   t2, 4*4(t0) # t2 est nine
  addi t3, t0, 2*4 # t3 est &p[2], soit five
  li   t4, 5       # t4 est 5 sur 32 bits
  sw   t4, (t3)    # maj de p[2]
  ...
  ret
```

Modes d'adressage

Variables : case mémoire

Accès indicé : (en supposant qu'on laisse les variables `i` et `s` dans des registres)

C :

```
int32_t p[] = {
   1, 3, 5, 7, 9,
   11, 13, 17
};
void argh(void)
{
  int32_t i, s = 0;

  for (i = 7; i >= 0; i--)
     s += p[i];
}
```

RISC-V : indirect registre + déplacement

```
 .data
p:.word 1, 3, 0, 7, 9, 11, 13, 17
 .text
argh:
  ...
  la   t4, p      # &p[0]
  li   t5, 0      # s
  li   t6, 7      # i
for:
  bltz t6, endfor
  slli t0, t6, 2  # i*4
  add  t0, t4, t0 # p+i*4
  lw   t1, (t0)   # accès
  add  t5, t5, t1 # ajout
  addi t6, t6, -1
  j    for
endfor:
  ...
  ret
```

Switch, le retour

Switch

```
int32_t humf(int32_t x, int32_t y)
{
  ...
  switch (y) {
    case 0:  x += 11;
             break;
    case 1:  x += 13;
             break;
    case 2:  x += 17;
             break;
    case 3:  x += 19;
             break;
    default: x += 23;
  }
  ...
}
```

```
 .rodata                  # en lecture seule
table :
 .word L0, L1, L2, L3
 .text
humf: ...                 # x est a0, y est a1
     sltu t0, a1, 4       # y < 4 ?
     beq  t0, zero, def   # non, on va à def
     la   t6, table       # adresse table de saut
     slli t1, a1, 2       # y*4 dans t1
     add  t6, t6, t1      # adresse dans table
     lw   t0, 0(t6)       # charge adresse de saut
     jr   t0              # saute
L0:  addi a0, a0, 11      # x += 11
     j    esw             # break
L1:  addi a0, a0, 13      # x += 13
     j    esw             # break
L2:  addi a0, a0, 17      # x += 17
     j    esw             # break
L3:  addi a0, a0, 19      # x += 19
     j    esw             # break
def: addi a0, a0, 23      # car par defaut
esw : ...                 # après le switch
```

Exécution de code RISC-V

Comment exécuter du code RISC-V sur un x86-64?

Grâce à un simulateur de système à base de RISC-V : QEMU

- Utilise la traduction binaire dynamique ($\Rightarrow$ très rapide)
- Modélise les différents composants dont nous avons besoin
- Système *ad-hoc* simulé et réel sur carte Zybo identiques
  $\Rightarrow$ Code tourne sur les 2
- Mais le debug est beaucoup plus facile sur le simulateur!

Utilisation de QEMU

```
qemu-system-riscv32 -machine cep -nographic -kernel hello
qemu-system-riscv32 -sdl -serial mon:stdio -machine cep -m 32M -kernel snake
```

Utilisation de QEMU et GDB

1er terminal : `qemu-system-riscv32 -machine cep -nographic -kernel hello -s -S`
2nd terminal : `riscv32-unknown-elf-gdb hello -ex 'target remote :1234'`