0111 - Instructions

Plus d'instruction et leur représentation

Jean Privat

Université du Québec à Montréal

INF2171 - Organisation des ordinateurs et assembleur
v233

Objectifs

- Des instructions RISC-V supplémentaires
- Le codage des instructions RISC-V
- Des exercices

Plan

1. Codage des instructions machines
2. Code automodifiable
3. Arithmétique et logique (extra)
4. Registres d'état et de configuration
5. Conclusion

Codage des instructions machines

Langage machine

Instructions

- Opération = quoi faire?
- Opérandes = quels registres, valeurs immédiates, emplacements mémoires?

Catégories d'instructions

- Instruction arithmétiques et logiques (calculs)
- Instructions de transfert (déplacement d'information)
- Instructions de contrôle (déroulement de l'exécution)

Types d'architectures

CISC

- *Complex instruction-set computer*
- Nombreuse instructions générales et spécialisées
- Programmes courts et lisibles

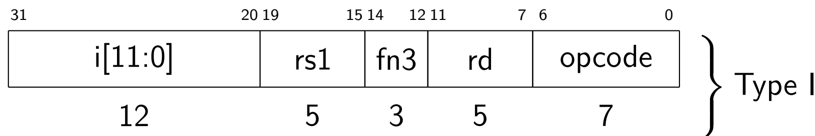
RISC

- *Reduced instruction-set computer*
- Seulement des instructions simples
- Plus de place pour des registres et des caches

Instructions RISC-V

- 32 bits (ou 16 bits avec l'extension C)
- Organisation régulière pour simplifier les micro-architectures

Type I: instructions avec une petite valeur immédiate



- opcode → code d'opération
- rd → registre destination (5 bits: $2^5 = 32$ registres)
- fn3 → sous-fonction (3 parce que 3 bits ici)
- rs1 → registre source
- i[11:0] → les bits 0 à 11 de la valeur immédiate (généralement avec extension de signe)

Exemple addi (instruction de type I)

- `addi s0, s1, 42` « range dans `s0` la somme de `s1` et de 42 »

31 20 19 15 14 12 11 7 6 0

<table>
<tr><td colspan="12">i[11:0]</td><td colspan="5">rs1</td><td colspan="3">fn3</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
<tr><td colspan="12">42</td><td colspan="5">s1=x9</td><td colspan="3">addi</td><td colspan="5">s0=x8</td><td colspan="7">OP-IMM</td></tr>
<tr><td colspan="12">0 0 0 0 0 0 1 0 1 0 1 0</td><td colspan="5">0 1 0 0 1</td><td colspan="3">0 0 0</td><td colspan="5">0 1 0 0 0</td><td colspan="7">0 0 1 0 0 1 1</td></tr>
<tr><td colspan="4">0</td><td colspan="4">2</td><td colspan="4">A</td><td colspan="4">4</td><td colspan="4">8</td><td colspan="4">4</td><td colspan="4">1</td><td colspan="4">3</td></tr>
</table>

- Soit `02A48413` en hexadécimal

Notes

- Les frontière de champs ne sont pas aux octets
- Les immédiats ont 12 bits seulement dans le type I

Les différents types d'instructions RISC-V

<table>
<tr><th>31 25</th><th>24 20</th><th>19 15</th><th>14 12</th><th>11 7</th><th>6 0</th><th></th></tr>
<tr><td>fn7</td><td>rs2</td><td>rs1</td><td>fn3</td><td>rd</td><td>opcode</td><td>Type R</td></tr>
<tr><td colspan="2">i[11:0]</td><td>rs1</td><td>fn3</td><td>rd</td><td>opcode</td><td>Type I</td></tr>
<tr><td>i[11:5]</td><td>rs2</td><td>rs1</td><td>fn3</td><td>i[4:0]</td><td>opcode</td><td>Type S</td></tr>
<tr><td>i[12|10:5]</td><td>rs2</td><td>rs1</td><td>fn3</td><td>i[4:1|11]</td><td>opcode</td><td>Type B</td></tr>
<tr><td colspan="4">i[31:12]</td><td>rd</td><td>opcode</td><td>Type U</td></tr>
<tr><td colspan="4">i[20|10:1|11|19:12]</td><td>rd</td><td>opcode</td><td>Type J</td></tr>
</table>

- Le décodage est pénible à la main
  Surtout les valeurs immédiates
- Mais facile pour une micro-architecture

Exercice de décodage

Instruction `00813903`?

- Utilisez l'annexe ou fouillez dans les spécifications.
- Transformer en binaire
- Chercher l'opcode (7 bits de poids faible)
- Découper les champs
- Décoder les champs un à un
- Fabriquer l'instruction
- Trouver la pseudoinstruction la plus simple équivalente (si besoin)

Instruction en mémoire

Petit-boutisme

- Les instructions RISC-V sont petit boutistes
- L'octet de poids faible est stocké en mémoire en premier
- Avantage: La micro-architecture connait l'opcode en premier
- Attention: L'ordre des octets quand on décode à la main
  - 32 bits d'un coup (dans un registre interne)
  - 32 bits en mémoire (4 octets un par un petit-boutiste)

Alignement

- Les instruction RISC-V doivent être alignés sur 32 bits (ou 16 bits avec l'extension C)
- L'adresse de l'instruction doit être un multiple de 4 (ou de 2 avec l'extension C)

Note: cela explique pourquoi i[0] n'apparait pas dans les type B et J

Exercice

Écrire un programme `immediate.s`

- Afficher la valeur de l'opérande immédiate d'une instruction de type I (ex. `addi`)
- Même exercice pour une instruction de type S (ex. `sw`)
- Attention au signe

```
ici: addi s0, s1, -1234
labas: sw s0, -1234(s1)
```

Autres architectures

RISC-V extension C (*compressed instruction*)

- Permet de compresser certaines instructions fréquentes sur 2 octets (au lieu de 4)
- Complexifie la micro-architecture, mais augmente la densité du code

x86

- Codages de taille variable
  1 à 10 octets par instruction
  Instructions spécialisées globalement plus compactes
- Beaucoup d'irrégularités
  Complique les micro-architectures
  Fatigue les humains

Code automodifiable

 CC-BY-NC-SA

Code automodifiable (*self-modifying code*)

Principe

- Les instructions sont des octets en mémoire
- Or, on peut écrire ou modifier les octets en mémoire
- Donc, écrire ou modifier les instructions lors de l'exécution
- Le programme s'écrit lui-même! **Méta-programmation!**

Utilité

- Style à proscrire en général
- Chargement dynamique de bibliothèques
- Compression d'exécutable (*packer*)
- Optimisation (dont compilateurs juste-à-temps)
- Offuscation et mutation de code (principalement logiciels malveillants)
- Débogage et instrumentation dynamique

Micro-architecture modernes

Caches, pipelines, superscalaire, etc.

- Processeurs cachent, voire traitent plusieurs instructions d'avance
  - Moins d'accès en mémoire
  - Plus de performance
- Détails en INF4170 Architecture des ordinateurs

Cohérence

Altérer les instruction futures cause des problème de cohérence

- x86: la cohérence des caches est gérée par la micro-architecture
  - Architecture complexe et moins performante
- RISC-V et ARM: demande l'aide du programme
  - Instruction barrière mémoire / synchronisation de caches

Barrière d'instruction RISC-V

`fence.i` (*instruction fence*)

- Garantie que les écritures passées sont visible par les *fetchs* d'instructions futures
- Le `.` fait parti du nom de l'opcode (ce n'est pas une syntaxe particulière)
- Note: pas d'opérande
- Comportement indéfini si la barrière manque

Ne pas confondre avec `fence`

- `fence` concerne la cohérence multi-cœurs (parallélisme)
- Les écritures passées d'un cœur sont visible par les lectures futures des autres cœurs

Exemple: code automodifiable.s

Que fait ce programme ?

```
    li a0, 22
    lw a1, ici
    sw a1, labas, t0
    fence.i
labas:  addi a0, a0, 50
ici:    addi a0, a0, 10
    call printInt
    li a0, 0
    call exit
```

Exercice `getreg.s`

- Écrire une routine `getreg` qui retourne le contenu d'un registre indiqué par son numéro.
  - Entrée: `a0` le numéro d'un registre
  - Sortie: `a0` la valeur de ce registre
  - Note: attention à ne pas écraser par inadvertance le registre à afficher

Contraintes d'environnement

Raisons de sécurité

- Les segments de code (*text*) sont non modifiables par défaut
- Les segments de donnée sont non exécutables par défaut
- Les détails en INF3173 et INF600C

Changer le défaut

- RARS: *Settings>self-modifying code*
- Unix: appel système dédié pour changer les protections (`mprotect`)

Arithmétique et logique (extra)

 CC-BY-NC-SA

Comparaison sans branchement

Ne branchent pas, mais assignent 1 ou 0 au registre destination

- **`slt rd, rs1, rs2`** (*set less than* type R)
  si `rs1<rs2` alors `rd=1`, sinon `rd=0`
- **`sltu rd, rs1, rs2`** (*set less than unsigned* type R)
  Version non signée du précédent
- **`slti rd, rs1, imm`** (*set less than immediate* type I)
  si `rs1<imm` alors `rd=1`, sinon `rd=0`
  `imm` sur 12 bits signé
- **`sltiu rd, rs1, rs2`** (*set less than immediate unsigned* type I)
  Version non signée du précédent
  `imm` sur 12 bits non signé

Comparaison sans branchement

Pseudoinstructions

- **`sltz`** et **`sgtz`** pseudoinstructions, comparent avec 0
  - `sltz rd, rs1` → `slt rd, rs1, x0`
  - `sgtz rd, rs2` → `slt rd, x0, rs2`
- **`seqz`** et **`snez`** pseudoinstructions, testent l'égalité avec 0
  - `seqz rd, rs1` → `sltiu rd, rs1, 1`
  - `snez rd, rs2` → `sltu rd, x0, rs2`
  - Notez les astuces mathématiques

Notes

- Pas de version *égal* ou *différent* seule
- Pour plus grand, permuter les opérandes
- Pour plus plus petit ou égal
  - Jouer sur les bornes: $a \leq b \Leftrightarrow a < b + 1$
  - ou faire l'opération opposée: $a \leq b \Leftrightarrow b \nless a$

Instruction 32 bits en RV64

- RV64 ajoute des instructions « 32 bits »

Travailler comme en RV32

- Considèrent que les 32 bits de poids faible des opérandes
- Tronquent les résultats à 32 bits (ignorent les débordement)
- Font une extension de signe dans le registre destination
  - Le bit 31 est copié sur les bits 32 à 63

RV64I

- `addw`, `addiw`, `subw`, `sllw`, `slliw`, `slrw`, `srliw`, `sraw`, `sraiw`

RV64M

- `mulw`, `divw`, `divuw`, `remw`, `remuw`

Registres d'état et de configuration

État et contrôle du système

Nombreux détails à gérer

- Connaître l'état du processeur, sa configuration, ses capacités
- Changer ou contrôler le comportement général de l'UTC
- Nombreux besoins très spécifiques

Variations architecturales (et vocabulaire)

- Registre d'état (*status register*)
- Registre de drapeaux (*flag register*)
- Registre de code de condition (*condition code register*, CCR)
- Registre de contrôle (*control register*)
- Registre d'état et de configuration (*Control and Status Register*, CSR)

Registres d'état et de configuration RISC-V

Registres CSR

- Nommés et numérotés (uniques)
- Taille standard: 32bits en RV32I, 64bits en RV64I
- Rôles et utilisation spécifique: RTFM
- Mais accès uniforme

Quelques ensembles de CSR

- RV32I: Compteur de performance matériel
- Extension "F": Modes et exceptions flottantes
- Extension "V": Mode et configuration
- Extension "N": Interruption utilisateur
- Mode privilégié (volume 2): nombreux CSR

Compteur de performance matériel

- *Hardware performance counters* (HPC)
- ou *performance monitoring counter* (PMC)
- Comptent des évènements matériels
- Sert au diagnostique et à la mesure de performance
  - Outil Linux `perf`
  - Exemple: `perf stat sha512sum /boot/initrd.*`

RISC-V

- Compteurs sur 64 bits
  - donc deux registres en RV32I, mais un seul en RV64I
- `cycle` (et `cycleh` en RV32I)
  - le nombre de cycle exécutés
- `time` (et `timeh` en RV32I)
  - le nombre d'unité de temps écoulés
- `instret` (et `instreth` en RV32I), *instructions retired*
  - le nombre d'instruction totalement exécutés

Accès (simple) au CSR

Pseudoinstructions dédiées

- `rdcycle`, `rdcycleh`, `rdtime`, etc.

```
rdcycle a0
```

Exercice: programme `cycle.s`

- Faire une boucle infinie qui affiche la valeur de `cycle`
- Même chose avec `gnu assembler`

Instructions d'accès aux CSR (type I)

- `csrrw rd, csr, rs1` (*CSR read write*)
- `csrrs rd, csr, rs1` (*CSR read set*)
- `csrrc rd, csr, rs1` (*CSR read clear*)

Où

- `csr`: nom ou numéro du CSR (12 bits non signé: 0 à 4095)
- `rd`: la valeur lue du CSR (*read*)
  - Avant modification éventuelle
  - `x0` pour ne pas lire, mais seulement écrire
- `rs1`: la valeur pour modifier
  - `csrrw`: `csr := rs1` (écriture)
  - `csrrs`: `csr := csr | rs1` (masque de bits)
  - `csrrc`: `csr := csr & ~rs1` (masque de bits)
  - `x0` pour ne pas écrire, mais seulement lire

Instructions d'accès aux CSR

Version immédiates (type I aussi)

- `csrrwi rd, csr, imm` (*CSR read write immediate*)
- `csrrsi rd, csr, imm` (*CSR read set immediate*)
- `csrrci rd, csr, imm` (*CSR read clear immediate*)

Où

- `csr`: nom ou numéro du CSR (12 bits non signé: 0 à 4095)
- `rd`: la valeur lue du CSR (*read*)
- `imm`: la valeur pour modifier
  - Attention: seulement 5 bits non signé (0 à 32)

Pseudoinstructions

- Lecture seule (`rs1` vaut `x0`): `csrr`
- Écriture seule (`rd` vaut `x0`): `csrw`, `csrwi`, `csrs`, `csrsi`, `csrc`, `csrci`

Autres architectures

Registre d'état FLAGS (x86)

- Registre `FLAGS` 16 bits (Intel 8086, 1978)
  - Extension à 32 bits nommée EFLAGS (Intel 80386, 1985)
  - Extension à 64 bits nommée RFLAGS (AMD64, 2000)
  - Mais 20 bits seulement utilisés
- Registre spécial (pas utilisable directement)
  - Instruction `pushf` et `popf` pour manipuler directement FLAGS
  - Chaque instruction peut indépendamment utiliser ou modifier FLAGS

Instruction `RDPMC` (x86)

- Lit le compteur de performance dont le numéro est ECX
- Écrit le résultat dans les registres EDX et EAX

Conclusion

On à fini RV64IM!

Nos dernières instructions

- Instructions: `slt`, `sltu`, `slti`, `sltiu`, `fence.i`, `addw`, `addiw`, `subw`, `sllw`, `slliw`, `slrw`, `srliw`, `sraw`, `sraiw`, `mulw`, `divw`, `divuw`, `remw`, `remuw`, `csrrw`, `csrrs`, `csrrc`, `csrrwi`, `csrrsi`, `csrrci`
- Pseudoinstructions: `sltz`, `sgtz`, `seqz`, `snez` et toutes les pseudoinstructions CSR

Programmes

- `immediate.s`, `automodifiable.s`, `getreg.s`, `cycle.s`

Qu'est-ce qu'il manque?

Au niveau architecture (ISA)

- Extensions "F" et "D" (flottants simple et double précision)
  - Plus tard
- Extension "A" (instructions atomiques) et autres extensions
  - Pas au programme
  - Vous pouvez lire la spécification
- Supervision et mode privilégié (et extension "N")
  - Plus tard

Au niveau programmation

- Plein de choses (voir plan de cours)

La prochaine fois

Examen

- Ça devrait bien aller

Routines

- Récursivité, programmes, bibliothèques, pointeurs de routine, paramètres dans la pile et variables locales

 CC-BY-NC-SA