Projet
Simulateur de Jeu d'Instructions

ISS – *Instruction Set Simulator*

Jean-Christophe Le Lann

Hannah Boenning

Rappels

- Notre projet vise à réaliser un simulateur
- L'intérêt d'une telle démarche est *multiple* :
  - Le processeur est simplement à l'étude
    - Hardware architects
  - Le processeur réel n'existe pas encore
    - Embedded software developpers
    - Ahead of time
  - Le processeur existe mais je n'en dispose pas
    - Embedded software developper
      - Extended debug. Visibility. Ease of use, etc
      - Parfois : meilleure performance !
  - Le processeur n'existera jamais
    - Cas d'une VM : « virtual machine »
      - Ex : JVM, Parrot, YARV

VM in C

- Donné à titre *d'exemple*
- Répond à plusieurs questions, à plusieurs niveaux :
  - Niveau code assembleur :
    - A quoi *ressemble* de l'assembleur (en général) ?
  - Niveau asm → binaire :
    - Comment **encoder** des instructions (en général) ?
  - Niveau binaire → interpétation
    - Comment **organiser** le code du fetch-decode-execute ?

VM in C

- Niveau binaire → interpétation
  - Comment organiser le code du fetch-decode-execute ?
  - Comment modéliser une mémoire ?
    - Tableau : `int mem[TAILLE] (={…})`
  - Comment modéliser les registres ?
    - Tableau : `int reg[NB_REGS]`
  - Comment extraire les champs (bitfields) ?
    - Code opératoire (« codeop » plutôt que « instrNum »)
    - Numéro des registres opérandes
    - Flag imm : « immediate » (1 => immediate, 0=> register)
      - (Uniquement pour notre ISS)

ISS : notre ISA

Notations : r nom de registre (`r 0`, `r 1`, ..., `r 31`)
o nom de registre ou constante entière (`12`, `-34`, ...)
a constante entière

Syntaxe	Instruction	Effet
`add`(r_1, o, r_2)	Addition entière	r_2 reçoit $r_1 + o$
`sub`(r_1, o, r_2)	Soustraction entière	r_2 reçoit $r_1 - o$
`mult`(r_1, o, r_2)	Multiplication entière	r_2 reçoit $r_1 * o$
`div`(r_1, o, r_2)	Quotient entier	r_2 reçoit r_1/o
`and`(r_1, o, r_2)	« Et » bit à bit	r_2 reçoit r_1 « et » o
`or`(r_1, o, r_2)	« Ou » bit à bit	r_2 reçoit r_1 « ou » o
`xor`(r_1, o, r_2)	« Ou exclusif » bit à bit	r_2 reçoit r_1 « ou exclusif » o
`shl`(r_1, o, r_2)	Décalage arithmétique logique à gauche	r_2 reçoit r_1 décalé à gauche de o bits
`shr`(r_1, o, r_2)	Décalage arithmétique logique à droite	r_2 reçoit r_1 décalé à droite de o bits
`slt`(r_1, o, r_2)	Test « inférieur »	r_2 reçoit 1 si $r_1 < o$, 0 sinon
`sle`(r_1, o, r_2)	Test « inférieur ou égal »	r_2 reçoit 1 si $r_1 \leq o$, 0 sinon
`seq`(r_1, o, r_2)	Test « égal »	r_2 reçoit 1 si $r_1 = o$, 0 sinon
`load`(r_1, o, r_2)	Lecture mémoire	r_2 reçoit le contenu de l'adresse $r_1 + o$
`store`(r_1, o, r_2)	Écriture mémoire	le contenu de r_2 est écrit à l'adresse $r_1 + o$
`jmp`(o, r)	Branchement	saute à l'adresse o et stocke l'adresse de l'instruction suivant le `jmp` dans r
`braz`(r, a)	Branchement si zéro	saute à l'adresse a si $r = 0$
`branz`(r, a)	Branchement si pas zéro	saute à l'adresse a si $r \neq 0$
`scall`(n)	Appel système	n est le numéro de l'appel
`stop`	Arrêt de la machine	fin du programme

ISS : fonction *execute*

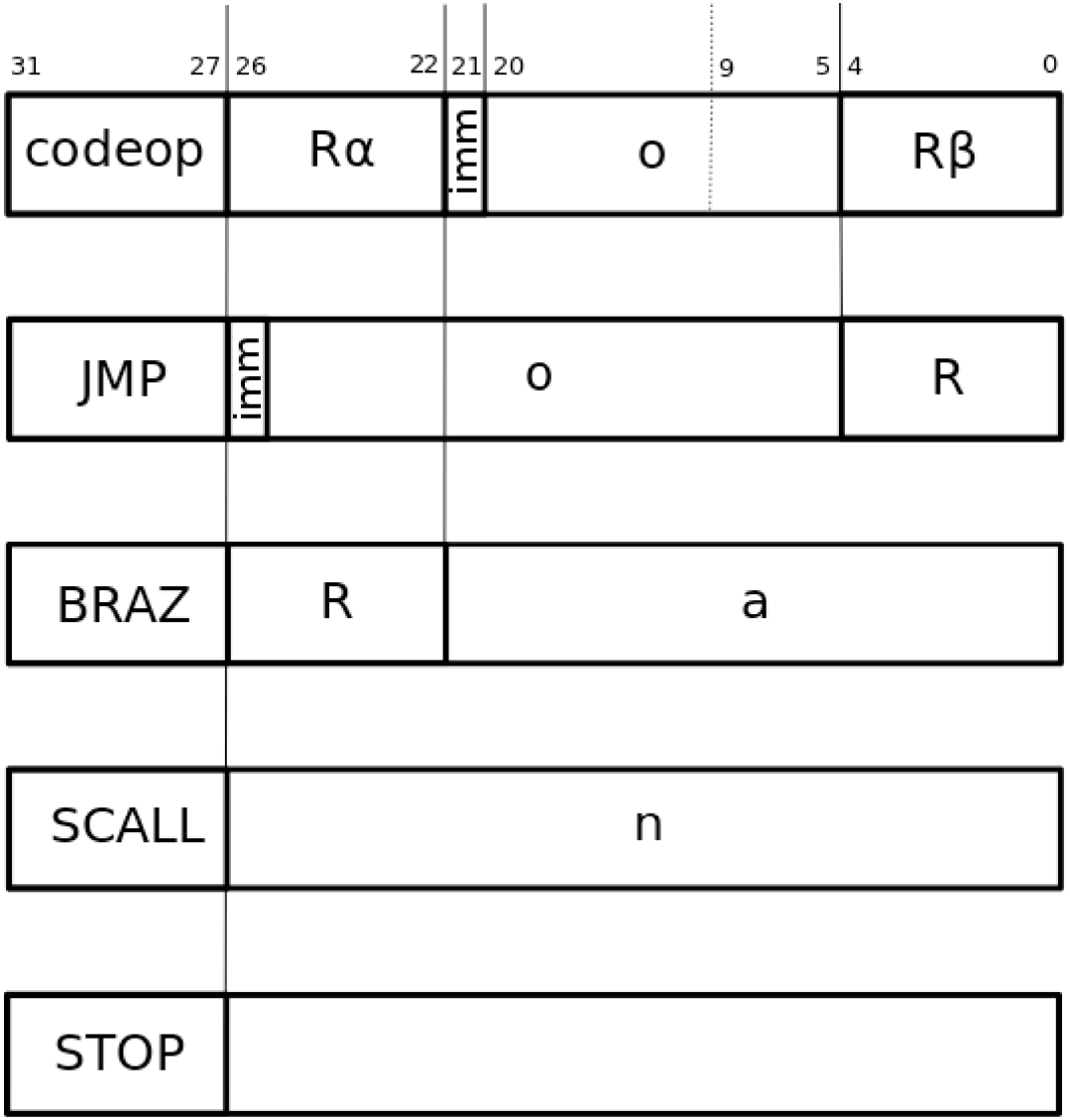
- En fonction du codeop (switch)
  - Réaliser les *actions/effets* appropriés sur
    - Les **registres**
      - Ex : ADD R1,R2,R3
    - La **mémoire** (data)
      - Ex : LOAD
      - Ex : STORE
  - Deux instructions d'*interaction* avec l'utilisateur
    - STOP
    - SCALL n
      - N=0 → READ a value on the Linux host keyboard (after enter) and store it in R1
      - N=1 → WRITE the content of R1 on the Linux host screen

Encodage binaire

Imposé !

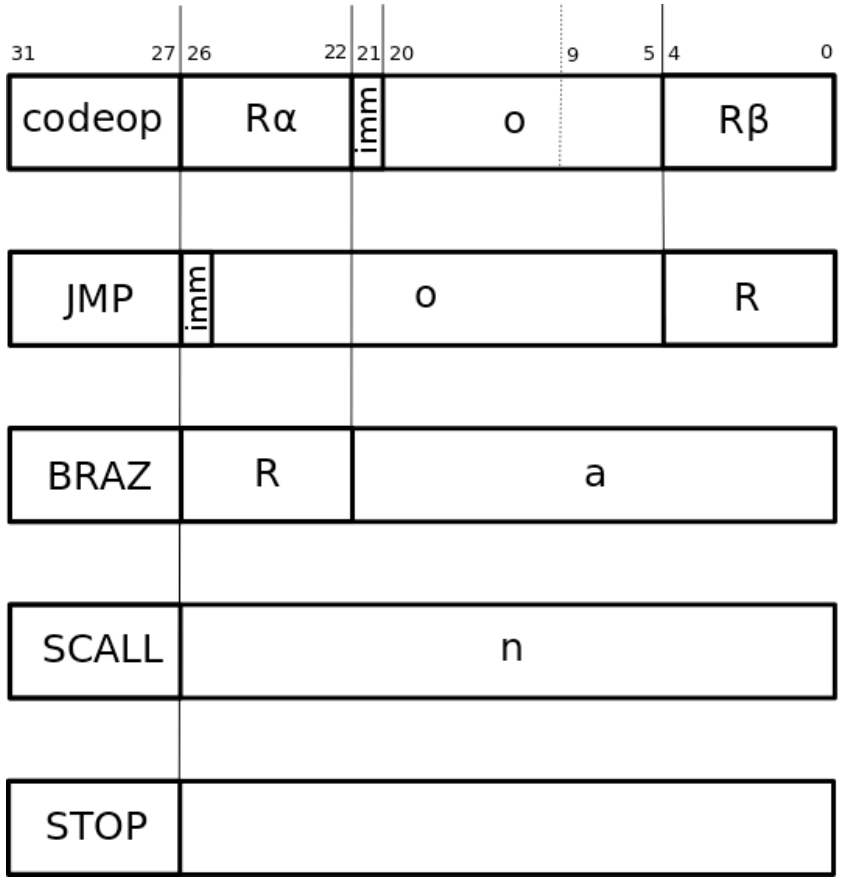
Notations : r nom de registre (`r 0`, `r 1`, …, `r 31`)
o nom de registre ou constante entière (`12`, `-34`, …)
a constante entière

Syntaxe	Instruction	Effet
`add`(r_1, o, r_2)	Addition entière	r_2 reçoit $r_1 + o$
`sub`(r_1, o, r_2)	Soustraction entière	r_2 reçoit $r_1 - o$
`mult`(r_1, o, r_2)	Multiplication entière	r_2 reçoit $r_1 * o$
`div`(r_1, o, r_2)	Quotient entier	r_2 reçoit r_1/o
`and`(r_1, o, r_2)	« Et » bit à bit	r_2 reçoit r_1 « et » o
`or`(r_1, o, r_2)	« Ou » bit à bit	r_2 reçoit r_1 « ou » o
`xor`(r_1, o, r_2)	« Ou exclusif » bit à bit	r_2 reçoit r_1 « ou exclusif » o
`shl`(r_1, o, r_2)	Décalage arithmétique logique à gauche	r_2 reçoit r_1 décalé à gauche de o bits
`shr`(r_1, o, r_2)	Décalage arithmétique logique à droite	r_2 reçoit r_1 décalé à droite de o bits
`slt`(r_1, o, r_2)	Test « inférieur »	r_2 reçoit 1 si $r_1 < o$, 0 sinon
`sle`(r_1, o, r_2)	Test « inférieur ou égal »	r_2 reçoit 1 si $r_1 \leq o$, 0 sinon
`seq`(r_1, o, r_2)	Test « égal »	r_2 reçoit 1 si $r_1 = o$, 0 sinon
`load`(r_1, o, r_2)	Lecture mémoire	r_2 reçoit le contenu de l'adresse $r_1 + o$
`store`(r_1, o, r_2)	Écriture mémoire	le contenu de r_2 est écrit à l'adresse $r_1 + o$
`jmp`(o, r)	Branchement	saute à l'adresse o et stocke l'adresse de l'instruction suivant le `jmp` dans r
`braz`(r, a)	Branchement si zéro	saute à l'adresse a si $r = 0$
`branz`(r, a)	Branchement si pas zéro	saute à l'adresse a si $r \neq 0$
`scall`(n)	Appel système	n est le numéro de l'appel
`stop`	Arrêt de la machine	fin du programme



Encodage binaire des instructions

- `ADD R7,R2,R3` se traduit en binaire par :
  - ADD → 1 (choix)
  - R7,R2,R3 → 7,0,2,3
  - (1,7,0,2,3) →




```
instr=0
instr+=1<<27
instr+=7<<22
instr+=0<<21
instr+=2<< 5
instr+=3
```

Instr vaut 163577923 = 0x9c00043

C'est un nombre. Peu importe la *représentation* de ce nombre. On parlera de *binaire.*

Manipulation bits à bits

quelques compléments utiles

- Mettre à 1 le bit 6 de la variable v :

  `v=v|(1<<6)` ou encore mieux : `v|=1<<6`

- Mettre à 0 le bit 2 de v :

  `v&=~(1<<2)`

- Inverser le bit 5 de v :

  `v^=1<<5`

- Extraire le champ 5..3 de v

  `field=(v & 56) >> 3` ou de manière plus explicite :

  `field=(v & 0x38)>>3`

Note : ~ représente l'opérateur de complément à 1 (inversion de tous les bits)

Développez *Agile* !

- Itérations courtes sur l'*ensemble des besoins*
  - On détient en permanence une solution. Dès le départ.
  - Peu importe que la solution soit incomplète
- Dans notre cas : construction de l'ISS
  - Avoir des exemples de code assembleur
    - même triviaux
  - Avoir un logiciel de traduction asm → binaire
    - Langages de script fortement recommandés ici
  - Commencer (aussi) la boucle d'interprétation.
    - Peu importe d'avoir toutes les instructions réalisées.