Projet Systèmes numériques

Pierre-Alexandre Bazin, Mathis Bouverot & Arthur Rousseau

19 janvier 2021

ISA - Mémoire

Tous les entiers manipulés sont signés et sur 16 bits. La RAM a des adresses sur 16 bits, et ses mots sont de longueur 16 bits. L'entièreté des 2^{16} mots de la RAM sont adressables. Une partie de la RAM est réservée aux entrées et sorties du processeur.

`adresse de début`	`fonction`
0	entrées
8	sorties
16	aucune

ISA - Registres

Il y a 8 registres nommés rax, rbx, rcx, rdx, rex, rfx, rgx, rz (registre nul). Ces registres sont sur 16 bits.
Codage des registres :

rz	rax	rbx	rcx	rdx	rex	rfx	rgx
000	001	010	011	100	101	110	111

ISA - Instructions

Les instructions sont sur 32 bits. Leur schéma est le suivant :

`imm`	`funct7`	`funct3`	`rs2`	`rs1`	`rd`	`opcode`

avec

- `opcode` sur 3 bits
- `funct3` sur 3 bits
- `funct7` sur 1 bit
- `rd`, `rs2`, `rs1` sur 3 bits
- `imm` sur 16 bits

ISA - Instructions

Inst	Nom	Opcode	funct3	funct7	Description
`add`	ADD	011	000	0	`rd = rs1 + rs2`
`sub`	SUB	011	000	1	`rd = rs1 - rs2`
`or`	OR	011	100	0	`rd = rs1 or rs2`
`nand`	NAND	011	100	1	`rd = rs1 nand rs2`
`xor`	XOR	011	001	0	`rd = rs1 xor rs2`
`nxor`	NXOR	011	001	1	`rd = rs1 nxor rs2`
`and`	AND	011	101	0	`rd = (rs1 and rs2)`
`nor`	NOR	011	101	1	`rd = rs1 nor rs2`
`sll`	Shift left logical	011	011	0	`rd = rs1 « rs2`
`srl`	Shift right logical	011	010	0	`rd = rs1 » rs2`
`sra`	Shift right Arith	011	010	1	`rd = rs1 » rs2`
`seq`	Set equal	011	111	1	`rd = (rs1 = rs2)?1:0`
`slt`	Set less than	011	110	1	`rd = (rs1 < rs2)?1:0`

ISA - Instructions

Immediate

Inst	Nom	Opcode	funct3	funct7	Description
`addi`	ADD	001	000	0	`rd = rs1 + imm`
`subi`	SUB	001	000	1	`rd = rs1 - imm`
`ori`	OR	001	100	0	`rd = rs1 or imm`
`nandi`	NAND	001	100	1	`rd = rs1 nand imm`
`xori`	XOR	001	001	0	`rd = rs1 xor imm`
`nxori`	NXOR	001	001	1	`rd = rs1 nxor imm`
`andi`	AND	001	101	0	`rd = rs1 and imm`
`nori`	NOR	001	101	1	`rd = rs1 nor imm`
`slli`	Shift left logical	001	011	0	`rd = rs1 « imm`
`srli`	Shift right logical	001	010	0	`rd = rs1 » imm`
`srai`	Shift right Arith	001	010	1	`rd = rs1 » imm`
`seqi`	Set equal	001	111	1	`rd = (rs1 = imm)?1:0`
`slti`	Set less than	001	110	1	`rd = (rs1 < imm)?1:0`

ISA - Instructions

Inst	Nom	Opcode	funct3	funct7	Description
`lw`	Load word	000	000	0	`rd=M[rs1+imm]`
`sw`	Store word	010	000	0	`M[rs1+imm]=rs2`
`beq`	Branch $==$	110	100	1	`if(rs1 == rs2) PC=imm`
`bne`	Branch $!=$	110	101	1	`if(rs1 != rs2) PC=imm`
`ble`	Branch $\leqslant$	110	110	1	`if(rs1 <= rs2) PC=imm`
`blt`	Branch $<$	110	010	1	`if(rs1 < rs2) PC=imm`
`bge`	Branch $\geqslant$	110	011	1	`if(rs1 >= rs2) PC=imm`
`bgt`	Branch $>$	110	111	1	`if(rs1 > rs2) PC=imm`
`jal`	Jump and link	111	000	0	`rd=PC+4; PC=imm`
`jalr`	Jump and link reg	101	000	0	`rd=PC+4; PC=rs1+imm`
`jr`	Jump reg	100	000	0	`PC=rs1+imm`
`jmp`	Jump	110	001	1	`PC=imm`

Processeur

3 composants principaux :

- reg_file - gestion des registres assembleur
- alu - l'ALU
- main - lit les instructions binaires dans la ROM et les exécute

Processeur - registres assembleur

le composant reg_file permet de lire dans 2 registres et d'écrire dans un troisième
7 registres ; le registre rz est remplacé par une constante nulle dans la netlist ; écrire dedans ne fait donc rien

Processeur - ALU

L'ALU prend en entrée deux entiers signés de 16 bits ainsi que funct3 (3 bits) et funct7 (1 bit) disant l'opération à faire
funct3 indique le circuit à utiliser (via un mux 8 entrées) ; funct7 commande des éventuelles inversions préalables
L'ALU renvoie aussi des flags comparant les deux entiers ; cependant ils ne sont bien positionnés que si funct7 $= 1$

Processeur - circuit principal

1 registre supplémentaire pc indiquant la tête de lecture dans la ROM ; il est incrémenté à chaque parcours de la netlist
on ne peut pas y accéder directement comme les autres registres ; on peut cependant le modifier via les instructions binaires jump
après avoir lu la rom, le circuit principal utilise reg_file et l'ALU pour effectuer l'opération
les bits de l'opcode donnent directement des informations sur le type d'instruction, ce qui évite d'utiliser un mux 8 entrées

- par exemple le bit de poids fort dit si l'instruction commade un jump

Processeur - sortie

L'écriture dans rz ne faisant rien, on l'utilise pour la sortie (en plus des adresses RAM de sortie)

- le processeur renvoie print_en : 1 si on essaie d'écrire dans rz et print_data : ce qu'on a assayé d'écrire dans rz
- le simulateur peut ensuite utiliser print_en et print_data
- par exemple, clock_simulator actualise l'horloge lorsque print_en $= 1$

Assembleur

Entrée : programme (texte).
Sortie : contenu initial de la ROM.
Syntaxe proche de x86-64 : deux sections (text/data).

Assembleur - Exemple

```
; Exemple de programme minias
    .section text
L1:
    add %rax %rbx 42
    mov x %rax
    jmp L1
    .section data
x:
    .space 2
y:
```

Assembleur - Instructions

Les instructions assembleur sont plus 'haut niveau' :

- Pas de distinction entre add/addi, sub/subi, etc.
- Une seule instruction `mov` pour toutes les copies :
  - 43 : immediate (entier)
  - %rax : registre
  - (%rax) : mémoire
- Les instructions de type jump sont regroupées : il en reste 2 dans l'assembleur (jmp et jal).

Assembleur - Labels

Il y a des labels pour éviter de manipuler directement des adresses ROM/RAM.

- Dans la section text : permet de faire des jump.
- Dans la section data : permet d'accéder à la RAM avec l'instruction `mov`.