コンピュータ構成と設計（２）

RISC-V 命令セットアーキテクチャ

李　亜民

2022 年 10 月 3 日 (月)

RISC-V アセンブリ言語プログラムの例

```
.text                     # code segment
main:                     # program entry
    addi x4, x0, 12       # reg[4] <= reg[0] + 12 = 12
    addi x5, x0, 13       # reg[5] <= reg[0] + 13 = 13
    add  x6, x4, x5       # reg[6] <= reg[4] + reg[5]
.end                      # end of program
```

`.text`	コードセグメント
`# ***`	コメント
`main:`	ラベル(ラベル `main:` はプログラム入り口)
`addi`, `add`	RISC-V 命令
`x0`, `x4`, `x5`, `x6`	RISC-V レジスタ(レジスタ `x0` は常に 0 である)
`.end`	おわり

RISC-V Instruction Format

31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
funct7		shamt		rs1	funct3	rd		opcode	H-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type
imm[31:12]						rd		opcode	U-type

- R-type: add, sub, slt, xor, or, and — add rd, rs1, rs2
- H-type: slli, srli, srai — slli rd, rs1, shamt
- I-type: jalr, addi, xori, ori, andi, lw — jalr rd, rs1, imm; addi rd, rs1, imm; lw rd, imm(rs1)
- S-type: sw — sw rs2, imm(rs1)
- B-type: beq, bne — beq rs1, rs2, label
- J-type: jal — jal rd, label
- U-type: lui — lui rd, imm

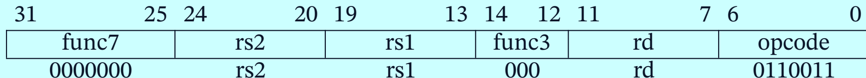
RV32I Base Instruction Set Encoding

0000000	rs2	rs1	000	rd	0110011	1.	add
0100000	rs2	rs1	000	rd	0110011	2.	sub
0000000	rs2	rs1	010	rd	0110011	3.	slt
0000000	rs2	rs1	100	rd	0110011	4.	xor
0000000	rs2	rs1	110	rd	0110011	5.	or
0000000	rs2	rs1	111	rd	0110011	6.	and
0000000	shamt	rs1	001	rd	0010011	7.	slli
0000000	shamt	rs1	101	rd	0010011	8.	srli
0100000	shamt	rs1	101	rd	0010011	9.	srai
imm[11:0]		rs1	000	rd	1100111	10.	jalr
imm[11:0]		rs1	000	rd	0010011	11.	addi
imm[11:0]		rs1	100	rd	0010011	12.	xori
imm[11:0]		rs1	110	rd	0010011	13.	ori
imm[11:0]		rs1	111	rd	0010011	14.	andi
imm[11:0]		rs1	010	rd	0000011	15.	lw
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	16.	sw
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	17.	beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	18.	bne
imm[20|10:1|11|19:12]				rd	1101111	19.	jal
imm[31:12]				rd	0110111	20.	lui

20 RISC-V 命令のまとめ

```
 1. add  rd, rs1, rs2    # rd <- rs1 + rs2
 2. sub  rd, rs1, rs2    # rd <- rs1 - rs2
 3. slt  rd, rs1, rs2    # rd <- rs1 < rs2 (signed)
 4. xor  rd, rs1, rs2    # rd <- rs1 ^ rs2
 5. or   rd, rs1, rs2    # rd <- rs1 | rs2
 6. and  rd, rs1, rs2    # rd <- rs1 & rs2
 7. slli rd, rs1, shamt  # rd <- rs1 << shamt
 8. srli rd, rs1, shamt  # rd <- rs1 >> shamt
 9. srai rd, rs1, shamt  # rd <- rs1 >>>shamt
10. jalr rd, rs1, imm    # rd <- pc+4; pc <- rs1+imm
11. addi rd, rs1, imm    # rd <- rs1 + imm
12. xori rd, rs1, imm    # rd <- rs1 ^ imm
13. ori  rd, rs1, imm    # rd <- rs1 | imm
14. andi rd, rs1, imm    # rd <- rs1 & imm
15. lw   rd,  imm(rs1)   # rd <- memory[rs1+imm]
16. sw   rs2, imm(rs1)   # memory[rs1+imm] <- rs2
17. beq  rs1, rs2, label # if (rs1==rs2) pc <- label
18. bne  rs1, rs2, label # if (rs1!=rs2) pc <- label
19. jal  rd,  label      # rd <- pc+4;   pc <- label
20. lui  rd,  imm        # rd <- imm,000000000000
```

1. add (Add)



31　　25	24　　20	19　　13	14　12	11　　7	6　　0
func7	rs2	rs1	func3	rd	opcode
0000000	rs2	rs1	000	rd	0110011

```
add  rd, rs1, rs2    # rd <- rs1 + rs2
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値を加算して，その結果をレジスタ rd に格納する。

例:

```
add  x8, x6, x7
```

For example,

```
if   reg[6] = 2
     reg[7] = 3
then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0010  ( 2)
            + 0000 0000 0000 0000 0000 0000 0000 0011  ( 3)
            = 0000 0000 0000 0000 0000 0000 0000 0101  ( 5)
```

2. sub (Subtraction)

31 25	24 20	19 13	14 12	11 7	6 0
func7	rs2	rs1	func3	rd	opcode
0100000	rs2	rs1	000	rd	0110011

```
sub  rd, rs1, rs2    # rd <- rs1 - rs2
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値を減算して，その結果をレジスタ rd に格納する。

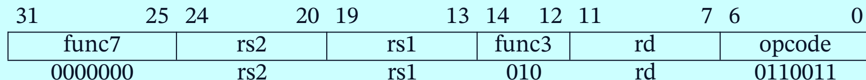
例:

```
sub  x8, x6, x7
```

For example,

```
if   reg[6] = 2
     reg[7] = 3
then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0010  ( 2)
            - 0000 0000 0000 0000 0000 0000 0000 0011  ( 3)
            = 1111 1111 1111 1111 1111 1111 1111 1111  (-1)
```

3. slt (Set on Less Than)




```
slt  rd, rs1, rs2    # rd <- rs1 < rs2 (signed)
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値を比較して，その結果をレジスタ rd に格納する。

例:

```
slt  x8, x6, x7
```

For example,

```
if   reg[6] = 2
     reg[7] = 3
then reg[8] = 1 because 2 < 3 is true
if   reg[6] = 3
     reg[7] = 2
then reg[8] = 0 because 3 < 2 is false
```

4. xor (Exclusive Or)

31 — 25	24 — 20	19 — 13	14 — 12	11 — 7	6 — 0
func7	rs2	rs1	func3	rd	opcode
0000000	rs2	rs1	100	rd	0110011

```
xor  rd, rs1, rs2    # rd <- rs1 ^ rs2
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値の排他的論理和を取り，その結果をレジスタ rd に格納する。

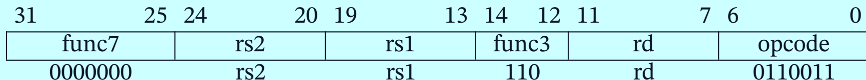
例:

```
xor  x8, x6, x7
```

For example,

```
if   reg[6] = 6
     reg[7] = 3
then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0110  ( 6)
            ^ 0000 0000 0000 0000 0000 0000 0000 0011  ( 3)
            = 0000 0000 0000 0000 0000 0000 0000 0101  ( 5)
```

5. or (Or)




```
or   rd, rs1, rs2    # rd <- rs1 | rs2
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値の論理和を取り，その結果をレジスタ rd に格納する。

例:

```
or   x8, x6, x7
```

For example,

```
if   reg[6] = 6
     reg[7] = 3
then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0110  ( 6)
            | 0000 0000 0000 0000 0000 0000 0000 0011  ( 3)
            = 0000 0000 0000 0000 0000 0000 0000 0111  ( 7)
```

6. and (And)

31 25	24 20	19 13	14 12	11 7	6 0
func7	rs2	rs1	func3	rd	opcode
0000000	rs2	rs1	111	rd	0110011

```
and  rd, rs1, rs2    # rd <- rs1 & rs2
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値の論理積を取り，その結果をレジスタ rd に格納する。

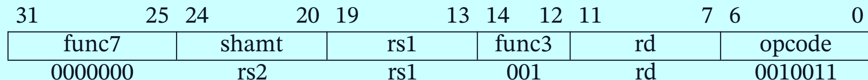
例:

```
and  x8, x6, x7
```

For example,

```
if   reg[6] = 6
     reg[7] = 3
then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0110  ( 6)
            & 0000 0000 0000 0000 0000 0000 0000 0011  ( 3)
            = 0000 0000 0000 0000 0000 0000 0000 0010  ( 2)
```

7. slli (Shift Left Logical Immediate)



31 25	24 20	19 13	14 12	11 7	6 0
func7	shamt	rs1	func3	rd	opcode
0000000	rs2	rs1	001	rd	0010011

```
slli rd, rs1, shamt  # rd <- rs1 << shamt
```

命令の意味: レジスタ rs1 に格納されている値を shamt ビット左に論理シフトして，その結果をレジスタ rd に格納する。

例:

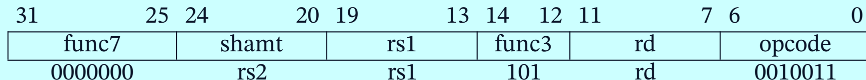
```
slli x8, x6, 4
```

For example,

```
if   reg[6] = 0000 0000 0000 0000 0000 0000 0000 0010

then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0010  << 4
            = 0000 0000 0000 0000 0000 0000 0010 0000
```

8. srli (Shift Right Logical Immediate)



31 　 25	24 　 20	19 　 13	14 　 12	11 　 7	6 　 0
func7	shamt	rs1	func3	rd	opcode
0000000	rs2	rs1	101	rd	0010011

```
srli rd, rs1, shamt  # rd <- rs1 >> shamt
```

命令の意味: レジスタ rs1 に格納されている値を shamt ビット右に論理シフトして，その結果をレジスタ rd に格納する。

例:

```
srli x8, x6, 4
```

For example,

```
if   reg[6] = 1111 0000 0000 0000 0000 0000 0000 0010

then reg[8] = 1111 0000 0000 0000 0000 0000 0000 0010  >> 4
            = 0000 1111 0000 0000 0000 0000 0000 0000
```

9. srai (Shift Right Arithmetic Imm.)

31–25	24–20	19–13	14–12	11–7	6–0
func7	shamt	rs1	func3	rd	opcode
0100000	rs2	rs1	101	rd	0010011

```
srai rd, rs1, shamt  # rd <- rs1 >>> shamt
```

命令の意味: レジスタ rs1 に格納されている値を shamt ビット右に算術シフトして，その結果をレジスタ rd に格納する。

例:

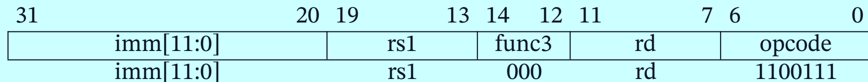
```
srai x8, x6, 4
```

For example,

```
if   reg[6] = 1111 0000 0000 0000 0000 0000 0000 0010

then reg[8] = 1111 0000 0000 0000 0000 0000 0000 0010 >>> 4
            = 1111 1111 0000 0000 0000 0000 0000 0000
```

10. jalr (Jump And Link Register)



31 20	19 13	14 12	11 7	6 0
imm[11:0]	rs1	func3	rd	opcode
imm[11:0]	rs1	000	rd	1100111

```
jalr rd, rs1, imm # rd <- pc+4; pc <- (rs1+$signed(imm[11:0]))&0xfffffffe
```

命令の意味: pc + 4 の値をレジスタ rd に格納する。レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm を加算して，最下位ビットをゼロにして，その結果をジャンプ先アドレスとしてジャンプする。例:

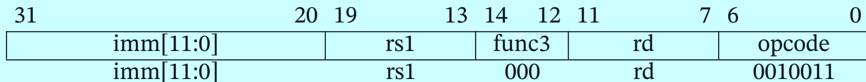
```
jalr x0, x1, 0       # = ret ra
```

For example,

```
if   reg[1] = 0000 0000 0000 0000 0000 0000 1111 0001

then pc     = 0000 0000 0000 0000 0000 0000 1111 0000
```

11. addi (Add Immediate)




```
addi rd, rs1, imm    # rd <- rs1 + $signed(imm[11:0])
```

命令の意味: レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm を加算して，その結果をレジスタ rd に格納する。

例:

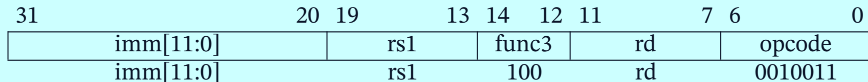
```
addi x8, x6, -1
```

For example,

```
if   reg[6] = 2

then reg[8] = 0000 0000 0000 0000 0000 0000 0000 0010  ( 2)
            + 1111 1111 1111 1111 1111 1111 1111 1111  (-1)
            = 0000 0000 0000 0000 0000 0000 0000 0001  ( 1)
```

12. xori (Exclusive Or Immediate)



31 20	19 13	14 12	11 7	6 0
imm[11:0]	rs1	func3	rd	opcode
imm[11:0]	rs1	100	rd	0010011

```
xori rd, rs1, imm    # rd <- rs1 ^ $signed(imm[11:0])
```

命令の意味: レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm の排他的論理和を取り，その結果をレジスタ rd に格納する。

例:

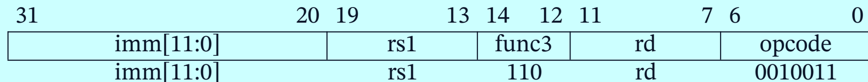
```
xori x8, x6, 0xf
```

For example,

```
if   reg[6] = xxxx xxxx xxxx xxxx xxxx xxxx xxxx 1010

then reg[8] = xxxx xxxx xxxx xxxx xxxx xxxx xxxx 1010
            ^ 0000 0000 0000 0000 0000 0000 0000 1111
            = xxxx xxxx xxxx xxxx xxxx xxxx xxxx 0101
```

13. ori (Or Immediate)



31 20	19 13	14 12	11 7	6 0
imm[11:0]	rs1	func3	rd	opcode
imm[11:0]	rs1	110	rd	0010011

```
ori  rd, rs1, imm    # rd <- rs1 | $signed(imm[11:0])
```

命令の意味: レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm の論理和を取り，その結果をレジスタ rd に格納する。

例:

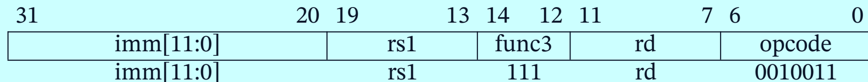
```
ori  x8, x6, 0xf
```

For example,

```
if   reg[6] = xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx

then reg[8] = xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
            | 0000 0000 0000 0000 0000 0000 0000 1111
            = xxxx xxxx xxxx xxxx xxxx xxxx xxxx 1111
```

14. andi (And Immediate)



31 　　　　　　　　20	19 　　　 13	14 　 12	11 　　　 7	6 　　　 0
imm[11:0]	rs1	func3	rd	opcode
imm[11:0]	rs1	111	rd	0010011

```
andi rd, rs1, imm    # rd <- rs1 & $signed(imm[11:0])
```

命令の意味: レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm の論理積を取り，その結果をレジスタ rd に格納する。

例:

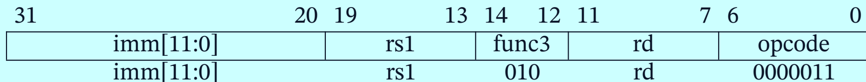
```
andi x8, x6, 0xf
```

For example,

```
if   reg[6] = xxxx xxxx xxxx xxxx xxxx xxxx xxxx 1010

then reg[8] = xxxx xxxx xxxx xxxx xxxx xxxx xxxx 1010
            & 0000 0000 0000 0000 0000 0000 0000 1111
            = 0000 0000 0000 0000 0000 0000 0000 1010
```

15. lw (Load Word)



31 — 20	19 — 13	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	func3	rd	opcode
imm[11:0]	rs1	010	rd	0000011

```
lw   rd,  imm(rs1)   # rd <- memory[rs1 + $signed(imm[11:0])]
```

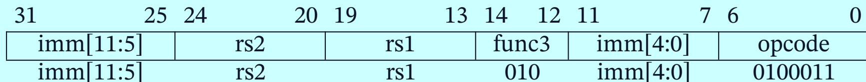
命令の意味: レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm の和をメモリ・アドレスとし，そのアクセスしたメモリのデータをレジスタ rd に格納する。例:

```
lw   x8, 4(x6)
```

For example,

```
if   reg[6] = 12
    mem[16] = 3
then reg[8] = mem[12+4] = mem[16] = 3
```

16. sw (Store Word)



31 25	24 20	19 13	14 12	11 7	6 0
imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011

```
sw   rs2, imm(rs1)   # memory[rs1 + $signed(imm[11:0])] <- rs2
```

命令の意味: レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm の和をメモリ・アドレスとし，そのアクセスしたメモリにレジスタ rs2 に格納されている値を書き込む。例:

```
sw   x8, 4(x6)
```

For example,

```
if   reg[6] = 12
     reg[8] = 3
then mem[12+4] = mem[16] = reg[8] = 3
```

17. beq (Branch on Equal)

31 — 25	24 — 20	19 — 13	14 — 12	11 — 7	6 — 0
imm[12|10:5]	rs2	rs1	func3	imm[4:1|11]	opcode
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011

```
beq  rs1, rs2, label # if (rs1 == rs2) pc <- pc + $signed({imm[12:1],0})
```

命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値が等しければ，分岐する。
分岐先アドレスは pc + $signed({imm[12:1],0}) である。例:

```
beq  x8, x0, label
```

For example,

```
if   reg[8] == 0 (equal, note: reg[0] = 0)
then pc = pc + $signed({imm[12:1],0})
else pc = pc + 4
```

18. bne (Branch on Not Equal)

31 — 25	24 — 20	19 — 13	14 — 12	11 — 7	6 — 0
imm[12|10:5]	rs2	rs1	func3	imm[4:1|11]	opcode
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011

```
bne  rs1, rs2, label # if (rs1 != rs2) pc <- pc + $signed({imm[12:1],0})
```

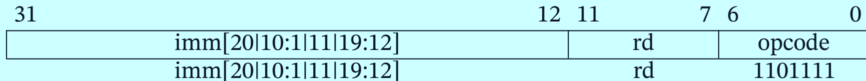
命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値が等しくなければ，分岐する。
分岐先アドレスは pc + $signed({imm[12:1],0}) である。例:

```
bne  x8, x0, label
```

For example,

```
if   reg[8] != 0 (equal, note: reg[0] = 0)
then pc = pc + $signed({imm[12:1],0})
else pc = pc + 4
```

19. jal (Jump And Link)




```
jal   rd, label        # rd <- pc + 4; pc <- pc + $signed({imm[20:1],0})
```

命令の意味: pc + 4 の値をレジスタ rd に格納する。pc に格納されている値と 2 の補数で表される即値 imm を加算して，その結果をジャンプ先アドレスとしてジャンプする。例:

```
jal   x1, subroutine   # = call subroutine
```

For example,

```
reg[1] = pc + 4        # save return address, use ret x1 to return
pc = subroutine        # jump to subroutine
```

20. lui (Load Upper Immediate)

```
lui   rd,   imm          # rd <- {imm[31:12],000000000000}
```

命令の意味: 即値 {imm[31:12],000000000000} をレジスタ rd に格納する。

例:

```
lui  x8, 0xfffff
```

For example,
reg[8] = 0xfffff000

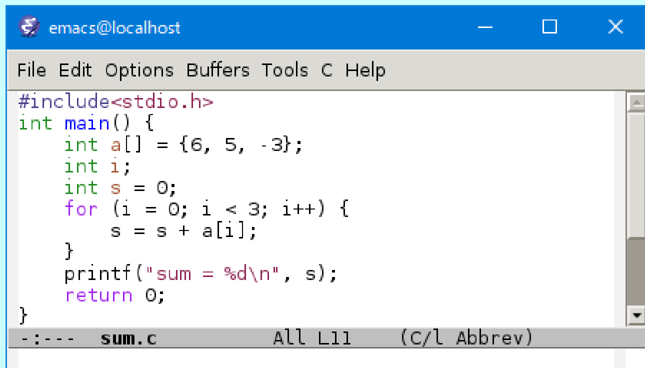
20 RISC-V 命令のまとめ

```
 1. add  rd, rs1, rs2    # rd <- rs1 + rs2
 2. sub  rd, rs1, rs2    # rd <- rs1 - rs2
 3. slt  rd, rs1, rs2    # rd <- rs1 < rs2 (signed)
 4. xor  rd, rs1, rs2    # rd <- rs1 ^ rs2
 5. or   rd, rs1, rs2    # rd <- rs1 | rs2
 6. and  rd, rs1, rs2    # rd <- rs1 & rs2
 7. slli rd, rs1, shamt  # rd <- rs1 << shamt
 8. srli rd, rs1, shamt  # rd <- rs1 >> shamt
 9. srai rd, rs1, shamt  # rd <- rs1 >>>shamt
10. jalr rd, rs1, imm    # rd <- pc+4; pc <- rs1+imm
11. addi rd, rs1, imm    # rd <- rs1 + imm
12. xori rd, rs1, imm    # rd <- rs1 ^ imm
13. ori  rd, rs1, imm    # rd <- rs1 | imm
14. andi rd, rs1, imm    # rd <- rs1 & imm
15. lw   rd,  imm(rs1)   # rd <- memory[rs1+imm]
16. sw   rs2, imm(rs1)   # memory[rs1+imm] <- rs2
17. beq  rs1, rs2, label # if (rs1==rs2) pc <- label
18. bne  rs1, rs2, label # if (rs1!=rs2) pc <- label
19. jal  rd,  label      # rd <- pc+4;   pc <- label
20. lui  rd,  imm        # rd <- imm,000000000000
```

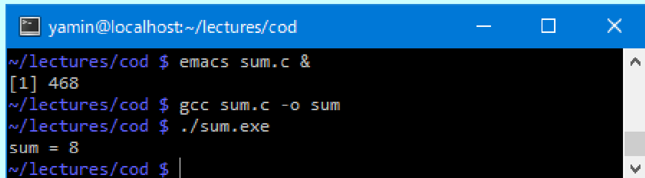
RISC-V Registers

Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate link register
x6–7	t1–2	Temporaries
x8	s0 / fp	Saved register / frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments / return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

C プログラムの例 (sum.c)

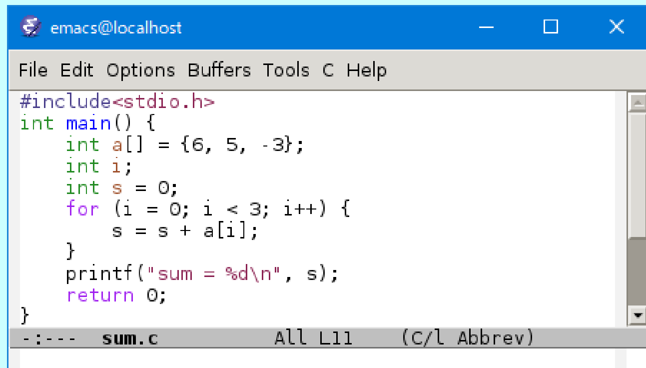


⇐ sum.c



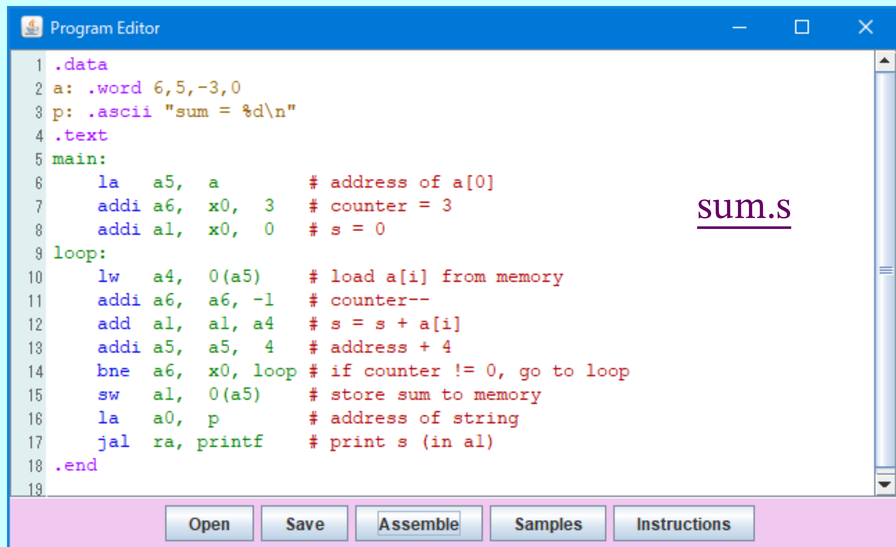
Cygwin

C プログラムの例 (sum.c)



RISC-V アセンブリ言語で実装：
配列 a？
for ループ？

RISC-V アセンブリ・プログラム



sum.s

RISC-V アセンブリ・プログラム

Rivasm - A RISC-V RV32IM Assembler and Simulator V1.0

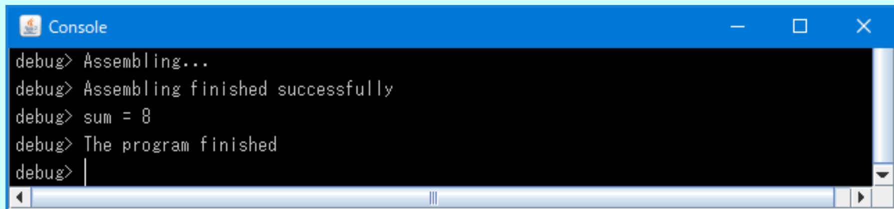
```
0x00000000:  0x00000006 0x00000005 0xfffffffd 0x00000008 0x206d7573 0x6425203d 0x0000000a 0x00000000
0x00000020:  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000040:  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000060:  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000080:  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x000000a0:  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x000000c0:  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x000000e0:  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
```

```
                                  .data
                                  a: .word 6,5,-3,0
                                  p: .ascii "sum = %d¥n"
                                  .text
                                  main:
0x00000020: 0x00000793       la   a5,  a        # address of a[0]
0x00000024: 0x00300813       addi a6,  x0,  3   # counter = 3
0x00000028: 0x00000593       addi a1,  x0,  0   # s = 0
                                  loop:
0x0000002c: 0x0007a703       lw   a4,  0(a5)    # load a[i] from memory
0x00000030: 0xfff80813       addi a6,  a6,  -1  # counter--
0x00000034: 0x00e585b3       add  a1,  a1,  a4  # s = s + a[i]
0x00000038: 0x00478793       addi a5,  a5,  4   # address + 4
0x0000003c: 0xfe0818e3       bne  a6,  x0,  loop # if counter != 0, go to loop
0x00000040: 0x00b7a023       sw   a1,  0(a5)    # store sum to memory
0x00000044: 0x01000513       la   a0,  p        # address of string
0x00000048: 0x711ff0ef       jal  ra, printf    # print s (in a1)
```

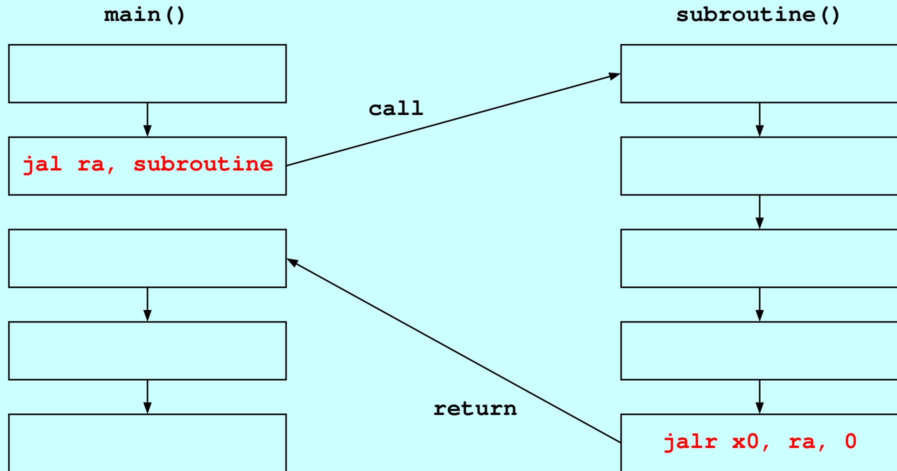
```
zero(x0):0x00000000
ra (x1): 0x0000004c
sp (x2): 0x0000ff00
gp (x3): 0x00000000
tp (x4): 0x00000000
t0 (x5): 0x00000000
t1 (x6): 0x00000000
t2 (x7): 0x00000000
s0 (fp): 0x00000000
s1 (x9): 0x00000000
a0(x10): 0x00000010
a1(x11): 0x00000008
a2(x12): 0x00000000
a3(x13): 0x00000000
a4(x14): 0xfffffffd
a5(x15): 0x0000000c
a6(x16): 0x00000000
a7(x17): 0x00000000
```

Edit | Step | Goto | ASCII | Run | Quit | Xilinx | Altera | Verilog

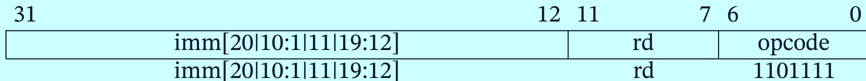
RISC-V アセンブリ・プログラム



サブルーチン呼び出し



サブルーチン呼び出し命令: jal



31 … 12	11 … 7	6 … 0
imm[20|10:1|11|19:12]	rd	opcode
imm[20|10:1|11|19:12]	rd	1101111

```
jal   rd, label     # rd <- pc+4; pc <- pc + $signed({imm[20:1],0})
```

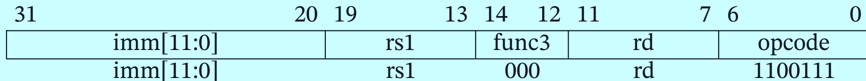
命令の意味: pc + 4 の値をレジスタ rd に格納する。pc に格納されている値と２の補数で表される即値 imm を加算して，その結果をジャンプ先アドレスとしてジャンプする。例:

```
jal   x1, subroutine   # = call subroutine
```

For example,

```
reg[1] = pc + 4        # save return address, use ret x1 to return
pc = subroutine        # jump to subroutine
```

サブルーチンから戻す命令: jalr



31 20	19 13	14 12	11 7	6 0
imm[11:0]	rs1	func3	rd	opcode
imm[11:0]	rs1	000	rd	1100111

```
jalr rd, rs1, imm # rd <- pc+4; pc <- (rs1+$signed(imm[11:0]))&0xfffffffe
```

命令の意味: pc + 4 の値をレジスタ rd に格納する。レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm を加算して，最下位ビットをゼロにして，その結果をジャンプ先アドレスとしてジャンプする。例:

```
jalr x0, x1, 0       # = ret ra
```

For example,

```
if   reg[1] = 0000 0000 0000 0000 0000 0000 1111 0001
then pc     = 0000 0000 0000 0000 0000 0000 1111 0000
```

サブルーチン呼び出し：C の例

```c
#include <stdio.h>
int sum (int a, int b) { // subroutine
    return a + b; // return
}
int main() {
    int x = 7, y = 5;
    int z;
    z = sum (x, y); // call subroutine
    printf("z = %d\n", z);
    return 0;
}
```

Compile and execute on PC:

```
$ gcc subroutine.c -o subroutine
$ ./subroutine
z = 12
```

サブルーチン呼び出し：C の例

yamin@localhost:~/lectures/cod

```
~/lectures/cod $ cat subroutine.c
#include <stdio.h>
int sum (int a, int b) { // subroutine
    return a + b; // return
}
int main() {
    int x = 7, y = 5;
    int z;
    z = sum (x, y); // call subroutine
    printf("z = %d\n", z);
    return 0;
}
~/lectures/cod $ gcc subroutine.c -o subroutine
~/lectures/cod $ ./subroutine
z = 12
~/lectures/cod $
```

サブルーチン呼び出し：RISC-V の例

```
.data
x: .word 7, 5, 0
p: .ascii "sum = %d\n"
.text
sum:                   # subroutine
    add  a0, a1, a2 # a + b in a0
    ret  ra         # return
main:
    addi sp, sp, -8 # reserve stack space
    sw   ra, 4(sp)  # save return address
    la   a3, x      # address of x
    lw   a1, 0(a3)  # load x from memory
    addi a3, a3,  4 # address of y
    lw   a2, 0(a3)  # load y from memory
    addi a3, a3,  4 # address of z
    jal  ra, sum    # call subroutine
    sw   a0, 0(a3)  # store z to memory
    addi a1, a0,  0 # a1: sum
    la   a0, p      # address of string
    jal  ra, printf # print z (in a1)
    lw   ra, 4(sp)  # restore return address
    addi sp, sp,  8 # release stack space
    ret  ra         # return
.end
```

サブルーチン呼び出し

Program Editor

```
.data
x: .word 7, 5, 0
p: .ascii "sum = %d\n"
.text
sum:                    # subroutine
    add  a0, a1, a2 # a + b in a0
    ret  ra         # return
main:
    addi sp, sp, -8 # reserve stack space
    sw   ra, 4(sp)  # save return address
    la   a3, x      # address of x
    lw   a1, 0(a3)  # load x from memory
    addi a3, a3,  4 # address of y
    lw   a2, 0(a3)  # load y from memory
    addi a3, a3,  4 # address of z
    jal  ra, sum    # call subroutine
    sw   a0, 0(a3)  # store z to memory
    addi a1, a0,  0 # a1: sum
    la   a0, p      # address of string
    jal  ra, printf # print z (in a1)
    lw   ra, 4(sp)  # restore return address
    addi sp, sp,  8 # release stack space
    ret  ra         # return
.end
```

Open | Save | Assemble | Samples | Instructions

サブルーチン呼び出し

Rivasm - A RISC-V RV32IM Assembler and Simulator V1.0

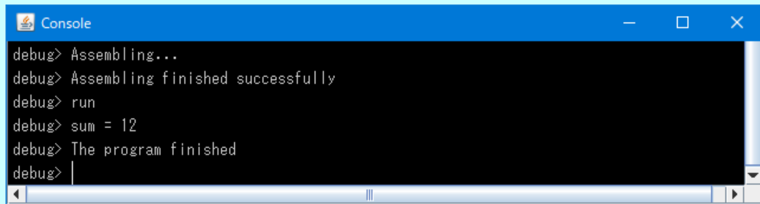
```
0x00000000:   0x00000007 0x00000005 0x0000000c 0x206d7573 0x6425203d 0x0000000a 0x00000000 0x00000000
0x00000020:   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000040:   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000060:   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000080:   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x000000a0:   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x000000c0:   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x000000e0:   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
```

```
                        .data
                        x: .word 7, 5, 0
                        p: .ascii "sum = %d¥n"
                        .text
                        sum:                 # subroutine
0x00000020: 0x00c58533     add  a0, a1, a2  # a + b in a0
0x00000024: 0x00008067     ret  ra          # return
                        main:
0x00000028: 0xff810113     addi sp, sp, -8  # reserve stack space
0x0000002c: 0x00112223     sw   ra, 4(sp)   # save return address
0x00000030: 0x00000693     la   a3, x       # address of x
0x00000034: 0x0006a583     lw   a1, 0(a3)   # load x from memory
0x00000038: 0x00468693     addi a3, a3,  4  # address of y
0x0000003c: 0x0006a603     lw   a2, 0(a3)   # load y from memory
0x00000040: 0x00468693     addi a3, a3,  4  # address of z
0x00000044: 0xfddff0ef     jal  ra, sum     # call subroutine
0x00000048: 0x00a6a023     sw   a0, 0(a3)   # store z to memory
0x0000004c: 0x00050593     addi a1, a0,  0  # a1: sum
0x00000050: 0x00c00513     la   a0, p       # address of string
0x00000054: 0x711ff0ef     jal  ra, printf  # print z (in a1)
0x00000058: 0x00412083     lw   ra, 4(sp)   # restore return address
0x0000005c: 0x00810113     addi sp, sp,  8  # release stack space
0x00000060: 0x00008067     ret  ra          # return
                        .end
```

```
zero(x0):0x00000000
ra (x1): 0x000fff00
sp (x2): 0x0000ff00
gp (x3): 0x00000000
tp (x4): 0x00000000
t0 (x5): 0x00000000
t1 (x6): 0x00000000
t2 (x7): 0x00000000
s0 (fp): 0x00000000
s1 (x9): 0x00000000
a0(x10): 0x0000000c
a1(x11): 0x0000000c
a2(x12): 0x00000005
a3(x13): 0x00000008
a4(x14): 0x00000000
a5(x15): 0x00000000
a6(x16): 0x00000000
a7(x17): 0x00000000
s2(x18): 0x00000000
s3(x19): 0x00000000
s4(x20): 0x00000000
s5(x21): 0x00000000
s6(x22): 0x00000000
s7(x23): 0x00000000
s8(x24): 0x00000000
s9(x25): 0x00000000
```

Edit | Step | Goto | ASCII | Run | Quit | Xilinx | Altera | Verilog

サブルーチン呼び出し



課題 II (100 点)

以下のプログラムを RISC-V アセンブリプログラムに変換し、Rivasm で実行しなさい。アセンブリプログラムと実行結果について説明しなさい。

```
#include <stdio.h>
int array1[] = {6, 5, -3};
int array2[] = {9, 8, -5, 1, 7};
int sum (int *a, int n) {
    int i;
    int s = 0;
    for (i = 0; i < n; i++) {
        s = s + a[i];
    }
    return s;
}
int main() {
    int z = sum (array1, 3);
    printf("z1 = %d\n", z);
    printf("z2 = %d\n", sum (array2, 5));
    return 0;
}
```

Note: Subroutine sum parameters: a1: the address of array[0], a2: n, a0: result

課題 II (100 点)

```
.data
array1: .word 6, 5, -3
array2: .word 9, 8, -5, 1, 7
p1: .ascii "z1 = %d\n"
p2: .ascii "z2 = %d\n"
.text
sum: # a1: address of array; a2: number of elements
    add  a0, x0, x0   # sum = 0
loop:
    # ... ... ... ... your codes here for sum subroutine
    ret  ra           # return, a0: result
main:
    addi sp, sp, -8   # reserve stack space
    sw   ra, 4(sp)    # save return address
    la   a1, array1   # a0: address of array1[0]
    addi a2, x0, 3    # a2: number of elements = 3
    jal  ra, sum      # call sum
    addi a1, a0,  0   # sum of array1
    la   a0, p1       # a0: address of string, for printf
    jal  ra, printf   # call printf, print sum (in a1)
    # ... ... ... ... your codes here for array2
    lw   ra, 4(sp)    # restore return address
    addi sp, sp,  8   # release stack space
    ret  ra           # return
.end
```