# コンピュータ構成と設計（１）

## RISC-V 命令セットアーキテクチャ

李　亜民

2022 年 9 月 26 日 (月)

# 授業のテーマ

- ## 授業のテーマ

  単一サイクル CPU、マルチサイクル CPU、パイプライン CPU、浮動小数点演算処理装置、メモリ階層、入出力システム、高性能計算

- ## 授業の到達目標

  コンピュータの物理的な仕組みと設計方法を学びます。特に、コンピュータを構成するプロセッサ内部のデータの流れ（データパス）とその制御部に関して，具体的な構成方法と設計の原理を理解します。さらに Altera Quartus II と ModelSim という実際のハードウェア設計にも使われているツールとハードウェア記述言語（Verilog HDL）を使用して簡単なプロセッサを設計し動作検証シミュレーションを行います。また、ハードウェアレベルのプログラミング言語であるアセンブラプログラミングについても学び、プロセッサの基本動作を理解します。そして、現代のコンピュータにおいて高速化の鍵となっている記憶階層について学習し、最後にネットワークや外部記憶その他の周辺装置について学びます。

- ## 授業の概要と方法

  コンピュータの設計方法を学びます。さらに実際のハードウェア設計にも使われている言語とツールを使用して CPU を設計し動作検証シミュレーションを行ないます。

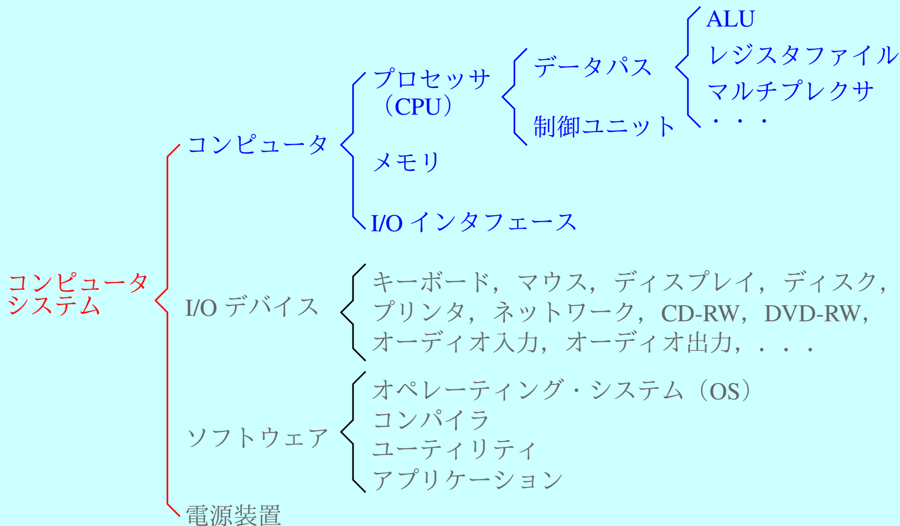
# 授業計画

1. RISC-V 命令セットアーキテクチャ (1)
2. RISC-V 命令セットアーキテクチャ (2)
3. 回路設計と Verilog HDL
4. 単一サイクル RISC-V CPU の設計 (1)
5. 単一サイクル RISC-V CPU の設計 (2)
6. 単一サイクル RISC-V CPU の設計 (3)
7. マルチサイクル RISC-V CPU の設計 (1)
8. マルチサイクル RISC-V CPU の設計 (2)
9. パイプライン RISC-V CPU の設計
10. 浮動小数点演算処理装置
11. メモリ、キャッシュ、仮想メモリ、TLB
12. 入出力システム
13. 性能評価と高性能計算システム
14. MIPS CPU の設計

# 成績評価

- 成績評価
  レポートの成績 × 70% + プロジェクトの成績 × 30%
- 課題レポートの提出：次回の授業開始前までに
  いかなる理由でも遅延提出は認めない (遅刻厳禁)
  課題レポートのフォーマット：PDF
  課題レポートのファイル名：cod-01-21k9999-名前.pdf
  (21k9999: 自分の ID；名前: 自分の名前)
  課題レポートの提出：学習支援システム (Hoppii)
- RISC-V 命令セットアーキテクチャ
  https://riscv.org/technical/specifications/

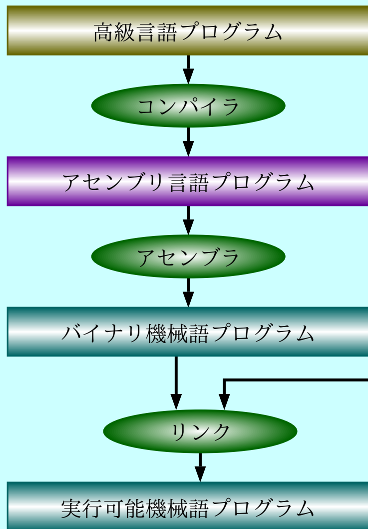
# コンピュータとコンピュータシステム



# コンピュータ言語

- ソフトウェア開発言語
  - 高級言語
    - C, C++, Java, Fortran, Python, Perl, ⋯
  - アセンブリ言語
    - Intel x86, MIPS, ARM, SPARC, RISC-V, ⋯
  - 機械語（二進数）
    - Intel x86, MIPS, ARM, SPARC, RISC-V, ⋯
- ハードウェア設計言語
  - Verilog HDL, VHDL, SystemVerilog, SystemC, Chisel, ⋯

# ソフトウェア開発の手順




```
swap(int v[], int k){
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```
swap:
    slli a1,a1,2   # k*4
    add  a0,a0,a1  # v+k*4
    lw   a4,0(a0)  # ld v[k]
    lw   a5,4(a0)  # ld v[k+1]
    sw   a5,0(a0)  # st v[k+1]
    sw   a4,4(a0)  # st v[k]
    jr   ra        # return
```

バイナリ機械語ライブラリ

```
00000000001001011001010110010011
00000000101101010000010100110011
00000000000001010010011100000011
00000000010001010010011110000011
00000000111101010010000000100011
00000000111001010010001000100011
00000000000000001000000001100111
```

# 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 命令の例

```
 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 Instruction Format

| 31 | 30–25 | 24–21 | 20 | 19–15 | 14–12 | 11–8 | 7 | 6–0 | Type |
|---|---|---|---|---|---|---|---|---|---|
| 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 |

| Type | Instructions | Example |
|---|---|---|
| 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 |

# 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 |

# 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 |

# 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。

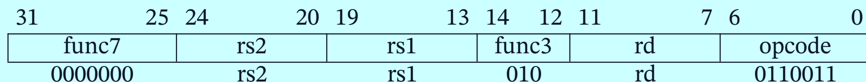
例:

```
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)



| 31 25 | 24 20 | 19 13 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| func7 | rs2 | rs1 | func3 | rd | opcode |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 |

```
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。

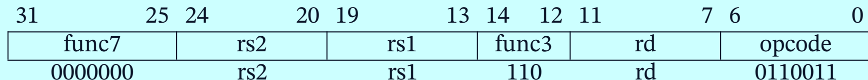
例:

```
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)



| 31 … 25 | 24 … 20 | 19 … 13 | 14 … 12 | 11 … 7 | 6 … 0 |
|---|---|---|---|---|---|
| func7 | rs2 | rs1 | func3 | rd | opcode |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 |

```
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。

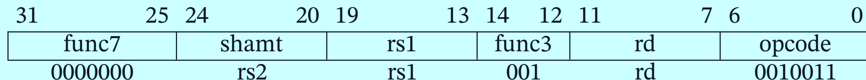
例:

```
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:

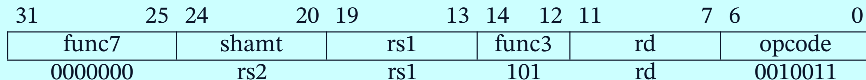
```
slli x8, x6, 4
```

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)




```
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:

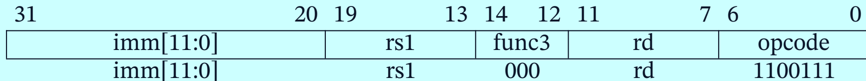
```
srai x8, x6, 4
```

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:

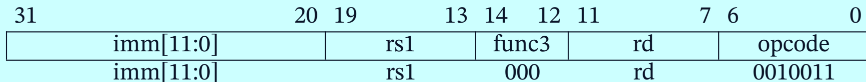
```
jalr x0, x1, 0        # = ret ra
```

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:

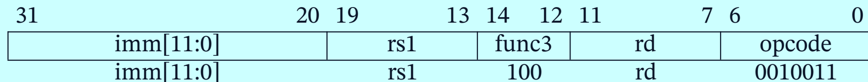
```
addi x8, x6, -1
```

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:

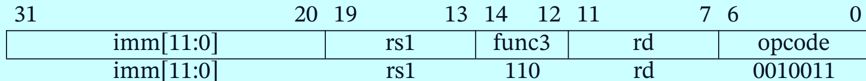
```
xori x8, x6, 0xf
```

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:

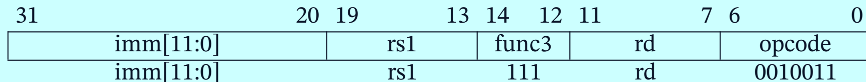
```
ori  x8, x6, 0xf
```

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)




```
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:

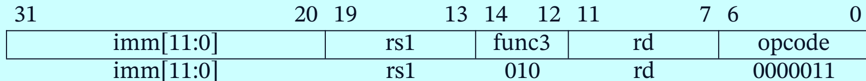
```
andi x8, x6, 0xf
```

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)




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

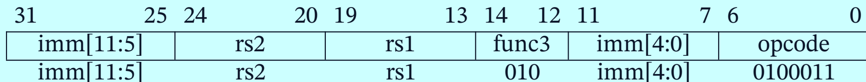
命令の意味: レジスタ rs1 に格納されている値と 2 の補数で表される即値 imm の和をメモリ・アドレスとし，そのアクセスしたメモリのデータをレジスタ rd に格納する。例:

```
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})
```

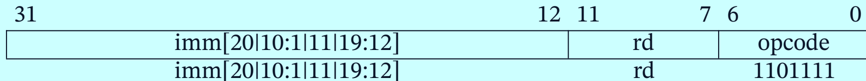
命令の意味: レジスタ rs1 に格納されている値とレジスタ rs2 に格納されている値が等しくなければ，分岐する。
分岐先アドレスは pc + $signed({imm[12:1],0}) である。例:

```
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)



| 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
```

# 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 アセンブリ言語プログラムの例

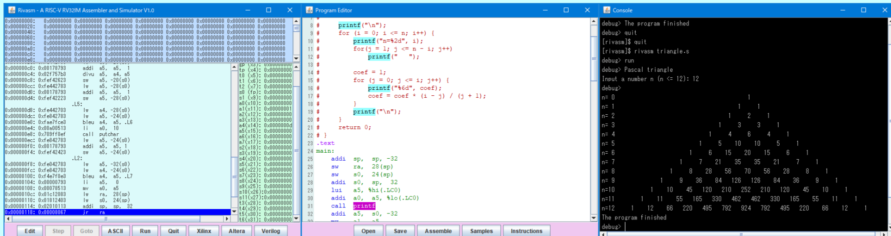
```
.text                   # code segment
main:                   # program entry
    addi x4, x0, 12     # reg[4] <= reg[0] + 12 = 12
    addi x5, x0, 13     # reg[5] <= reg[0] + 13 = 13
    add  x6, x4, x5     # reg[6] <= reg[4] + reg[5]
.end                    # end of program
```

| | |
|---|---|
| `.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 |

# Rivasm ダウンロード

- RISC-V アセンブリ言語プログラムを RISC-V 機械語に直しても、パソコンで直接に実行できません
- なぜなら、パソコンの CPU が Intel x86 機械語しかわかりません
- 従って、RISC-V シミュレータが必要になります
- Rivasm シミュレータをダウンロードします:
  https://yamin.cis.k.hosei.ac.jp/rivasm/rivasm.jar
- そして、rivasm.jar アイコンをダブルクリックし、または、ターミナルで下のコマンドを実行します:
  `java -jar rivasm.jar`

# Rivasm を実行



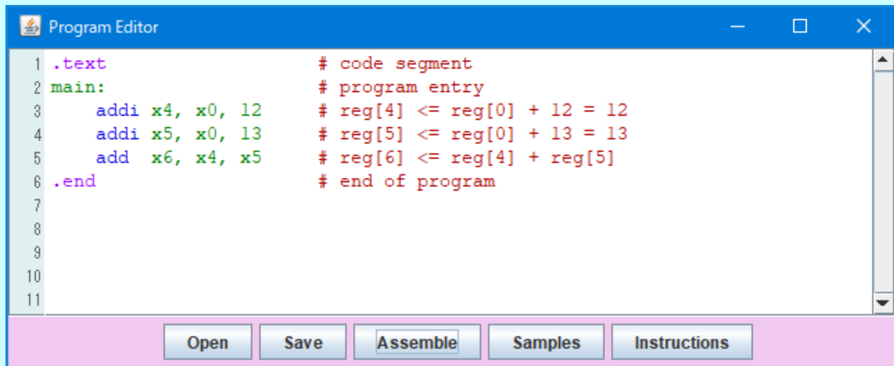

Main Window　Editor Window　Console Window

Rivasm の使い方については、

[https://yamin.cis.k.hosei.ac.jp/rivasm/](https://yamin.cis.k.hosei.ac.jp/rivasm/)

を参照ください

# add.s プログラムを編集




プログラムを編集して、
Assemble ボタンをクリック

# add.s プログラムをステップで実行

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

```
0x00000000:   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 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
```

```
                        .text                     # code segment
                        main:                     # program entry
0x00000000: 0x00c00213      addi  x4, x0, 12      # reg[4] <= reg[0] + 12 = 12
0x00000004: 0x00d00293      addi  x5, x0, 13      # reg[5] <= reg[0] + 13 = 13
0x00000008: 0x00520333      add   x6, x4, x5      # reg[6] <= reg[4] + reg[5]
                        .end                      # end of program
```

```
zero(x0):0x00000000
ra (x1): 0x000fff00
sp (x2): 0x0000ff00
gp (x3): 0x00000000
tp (x4): 0x00000000
t0 (x5): 0x00000000
t1 (x6): 0x00000000
t2 (x7): 0x00000000
```

番地　命令コード（１６進）　データメモリ　レジスタ →

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

最初の状態。
step ボタンをクリックすると、
`addi $4, $0, 12` を実行する

# add.s プログラムをステップで実行

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

```
0x00000000:   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 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
```

```
                          .text                # code segment
                          main:                # program entry
0x00000000: 0x00c00213        addi  x4, x0, 12     # reg[4] <= reg[0] + 12 = 12
0x00000004: 0x00d00293        addi  x5, x0, 13     # reg[5] <= reg[0] + 13 = 13
0x00000008: 0x00520333        add   x6, x4, x5     # reg[6] <= reg[4] + reg[5]
                          .end                 # end of program
```

```
zero(x0):0x00000000
ra (x1): 0x000fff00
sp (x2): 0x0000ff00
gp (x3): 0x00000000
tp (x4): 0x0000000c
t0 (x5): 0x00000000
t1 (x6): 0x00000000
t2 (x7): 0x00000000
```

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

`addi $4, $0, 12` を実行した。$4 = 12（16 進の c）。
step ボタンをクリックすると、
`addi $5, $0, 13` を実行する

# add.s プログラムをステップで実行

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

```
0x00000000:   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 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
```

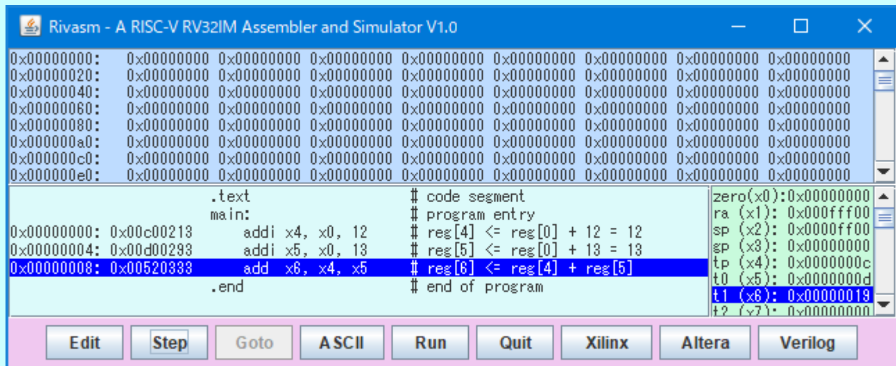
```
                            .text                  # code segment
                            main:                  # program entry
0x00000000: 0x00c00213         addi  x4, x0, 12    # reg[4] <= reg[0] + 12 = 12
0x00000004: 0x00d00293         addi  x5, x0, 13    # reg[5] <= reg[0] + 13 = 13
0x00000008: 0x00520333         add   x6, x4, x5    # reg[6] <= reg[4] + reg[5]
                            .end                   # end of program
```

```
zero(x0):0x00000000
ra (x1): 0x000fff00
sp (x2): 0x0000ff00
gp (x3): 0x00000000
tp (x4): 0x0000000c
t0 (x5): 0x0000000d
t1 (x6): 0x00000000
t2 (x7): 0x00000000
```

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

`addi $5, $0, 13` を実行した。$5 = 13（16 進の d)。
step ボタンをクリックすると、
`add  $6, $4, $5` を実行する

# add.s プログラムをステップで実行




add　$6，$4，$5 を実行した。$6 = 25（16 進の 19）。
つまり、12 + 13 = 25。
CPU がプログラムを実行する際、こんな感じ

# 課題 I (100 点)

Using li x2, 0xcafebabe can write a 32-bit immediate 0xcafebabe to reg[2]. Explain why it must be translated into two instructions. Write x = 0x91e5df78 and y = 0xa64fec53 to reg[2] and reg[3], respectively, and perform

sp (x2): 0x91e5df78
gp (x3): 0xa64fec53

reg[4] = x AND y;
reg[5] = x OR y;
reg[6] = NOT x; (Hint: for 1-bit x, $x \oplus 0 = x$; $x \oplus 1 = \overline{x}$)
reg[7] = x << 8; (shift x to the left by 8 bits)
reg[8] = x >> 8; (shift x to the right logically by 8 bits)
reg[9] = x >>> 8; (shift x to the right arithmetically by 8 bits)

Simulate your codes with Rivasm, and explain the results.

Submit your report to https://hoppii.hosei.ac.jp/portal/ before the next lecture starts. File name:
cod-01-21k9999-名前.pdf (21k9999: your ID; 名前: your name)