コンピュータ構成と設計（14）

MIPS ISA and CPU Design

李　亜民

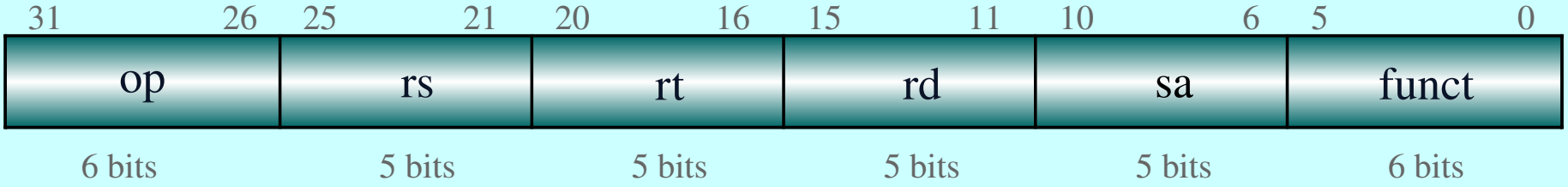
2024 年 1 月 11 日 (木)

MIPS ISA

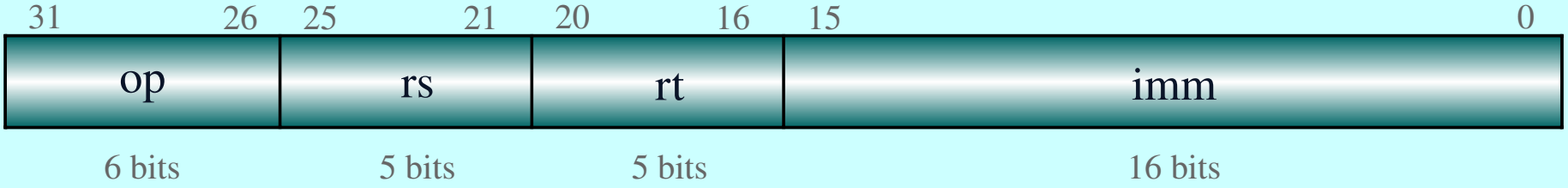
- RISC-V ISA is based on MIPS-4 ISA (>90% user instructions map one-to-one)
- MIPS: An architecture of a RISC (Reduced instruction set computers)
- MIPS - Microprocessor without Interlocked Pipeline Stages
- A project conducted by John L. Hennessy at Stanford University
- Hennessey founded MIPS Computer Systems Inc. ⇒ MIPS Technologies Inc., bought by Wave Computing Inc.
- All future designs are announced to be based on the RISC-V architecture (MIPS cores support RISC-V ISA)

MIPS 命令フォーマット

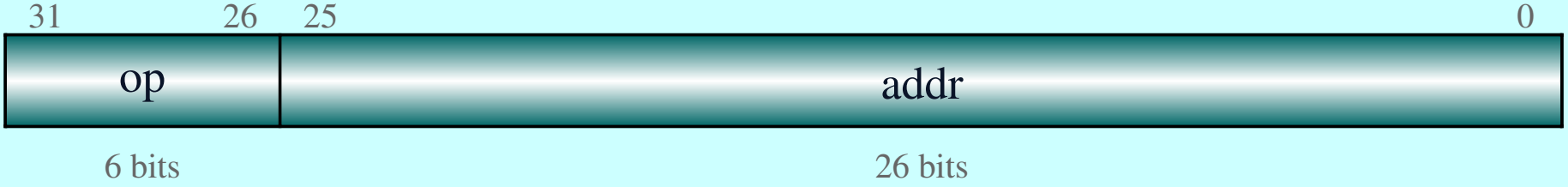
R 形式命令　(rs, rt, rd はレジスタ番号, sa はシフトさせる量)



I 形式命令　(rs, rt はレジスタ番号, imm は即値)



J 形式命令　(addr はジャンプ先のアドレス)



20 MIPS 命令の例

```
 1. add  rd, rs, rt   # reg[rd] = reg[rs]  +  reg[rt]
 2. sub  rd, rs, rt   # reg[rd] = reg[rs]  -  reg[rt]
 3. and  rd, rs, rt   # reg[rd] = reg[rs]  &  reg[rt]
 4. or   rd, rs, rt   # reg[rd] = reg[rs]  |  reg[rt]
 5. xor  rd, rs, rt   # reg[rd] = reg[rs]  ^  reg[rt]
 6. sll  rd, rt, sa   # reg[rd] = reg[rt]  << sa
 7. srl  rd, rt, sa   # reg[rd] = reg[rt]  >> sa (logical)
 8. sra  rd, rt, sa   # reg[rd] = reg[rt] >>> sa (arithmetic)
 9. jr   rs           #      PC = reg[rs]
10. addi rt, rs, imm  # reg[rt] = reg[rs]  +  (sign)imm
11. andi rt, rs, imm  # reg[rt] = reg[rs]  &  (zero)imm
12. ori  rt, rs, imm  # reg[rt] = reg[rs]  |  (zero)imm
13. xori rt, rs, imm  # reg[rt] = reg[rs]  ^  (zero)imm
14. lui  rt, imm      # reg[rt] = imm,0000000000000000
15. lw   rt, imm(rs)  # reg[rt] = memory[reg[rs]+(sign)imm]
16. sw   rt, imm(rs)  # memory[reg[rs]+(sign)imm] = reg[rt]
17. beq  rs, rt, imm  # if (reg[rs]==reg[rt]) PC=PC+4+(sign)imm*4
18. bne  rs, rt, imm  # if (reg[rs]!=reg[rt]) PC=PC+4+(sign)imm*4
19. j    addr         # PC = (PC+4)[31:28],addr,00
20. jal  addr         # PC = (PC+4)[31:28],addr,00; reg[31]=PC+4
```

20 MIPS 命令のフォーマット

```
 1. add  rd, rs, rt   # 000000  rs    rt    rd   00000 100000
 2. sub  rd, rs, rt   # 000000  rs    rt    rd   00000 100010
 3. and  rd, rs, rt   # 000000  rs    rt    rd   00000 100100
 4. or   rd, rs, rt   # 000000  rs    rt    rd   00000 100101
 5. xor  rd, rs, rt   # 000000  rs    rt    rd   00000 100110
 6. sll  rd, rt, sa   # 000000 00000  rt    rd    sa   000000
 7. srl  rd, rt, sa   # 000000 00000  rt    rd    sa   000010
 8. sra  rd, rt, sa   # 000000 00000  rt    rd    sa   000011
 9. jr   rs           # 000000  rs   00000 00000 00000 001000
10. addi rt, rs, imm  # 001000  rs    rt   _______imm________
11. andi rt, rs, imm  # 001100  rs    rt   _______imm________
12. ori  rt, rs, imm  # 001101  rs    rt   _______imm________
13. xori rt, rs, imm  # 001110  rs    rt   _______imm________
14. lui  rt, imm      # 001111 00000  rt   _______imm________
15. lw   rt, imm(rs)  # 100011  rs    rt   _______imm________
16. sw   rt, imm(rs)  # 101011  rs    rt   _______imm________
17. beq  rs, rt, imm  # 000100  rs    rt   _______imm________
18. bne  rs, rt, imm  # 000101  rs    rt   _______imm________
19. j    addr         # 000010  ____________addr_____________
20. jal  addr         # 000011  ____________addr_____________
```

1. add 命令 (Add)

```
add  rd, rs, rt  # reg[rd] = reg[rs] + reg[rt]
```

op	rs	rt	rd	sa	funct
0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0	1 0 0 0 0 0
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

命令の意味: レジスタ rs に格納されている値とレジスタ rt 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  $8, $6, $7
```

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 命令 (Subtract)

```
sub  rd, rs, rt  # reg[rd] = reg[rs] - reg[rt]
```

op	rs	rt	rd	sa	funct
0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0	1 0 0 0 1 0
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

命令の意味: レジスタ rs に格納されている値とレジスタ rt 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  $8, $6, $7
```

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. and 命令 (And)

```
and  rd, rs, rt  # reg[rd] = reg[rs] & reg[rt]
```

op	rs	rt	rd	sa	funct
0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0	1 0 0 1 0 0
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

命令の意味: レジスタ rs に格納されている値とレジスタ rt 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  $8, $6, $7
```

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 0010  ( 2)
```

4. or 命令 (Or)

```
or   rd, rs, rt  # reg[rd] = reg[rs] | reg[rt]
```

op	rs	rt	rd	sa	funct
0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0	1 0 0 1 0 1
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

命令の意味: レジスタ rs に格納されている値とレジスタ rt 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   $8, $6, $7
```

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 0011  ( 3)
```

5. xor 命令 (Exclusive Or)

```
xor  rd, rs, rt  # reg[rd] = reg[rs] ^ reg[rt]
```

op	rs	rt	rd	sa	funct
0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0	1 0 0 1 1 0
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

命令の意味: レジスタ rs に格納されている値とレジスタ rt 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  $8, $6, $7
```

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 0001  ( 1)
```

6. sll 命令 (Shift Left Logical)

```
sll  rd, rt, sa  # reg[rd] = reg[rt]  << sa
```

op	rs	rt	rd	sa	funct
0 0 0 0 0 0	0 0 0 0 0	rt	rd	sa	0 0 0 0 0 0
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

命令の意味: レジスタ rt に格納されている値を sa 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:

```
sll  $8, $6, 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
```

7. srl 命令 (Shift Right Logical)

```
srl  rd, rt, sa  # reg[rd] = reg[rt]  >> sa (logical)
```

op	rs	rt	rd	sa	funct
0 0 0 0 0 0	0 0 0 0 0	rt	rd	sa	0 0 0 0 1 0
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

命令の意味: レジスタ rt に格納されている値を sa 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:

```
srl  $8, $6, 4
```

For example,

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

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

8. sra 命令 (Shift Right Arithmetic)

```
sra  rd, rt, sa  # reg[rd] = reg[rt] >>> sa (arithmetic)
```

op	rs	rt	rd	sa	funct
0 0 0 0 0 0	0 0 0 0 0	rt	rd	sa	0 0 0 0 1 1
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

命令の意味: レジスタ rt に格納されている値を sa 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:

```
sra  $8, $6, 4
```

For example,

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

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

9. jr 命令 (Jump Register)

```
jr   rs            # PC = reg[rs]
```

op	rs	rt	rd	sa	funct
000000	rs	00000	00000	00000	001000
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

命令の意味: レジスタ rs に格納されている値を PC に格納する。

例:

```
jr   $31
```

For example,

```
if reg[31] = 1000 0000 0000 0000 0000 0000 0000 0000

then    PC = 1000 0000 0000 0000 0000 0000 0000 0000
```

10. addi 命令 (Add Immediate)

```
addi rt, rs, imm # reg[rt] = reg[rs] + (sign)imm
```

op	rs	rt	imm
0 0 1 0 0 0	rs	rt	imm
6-bit	5-bit	5-bit	16-bit

命令の意味: レジスタ rs 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 rt に格納する。
例:

```
addi $8, $6, -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)
```

11. andi 命令 (And Immediate)

```
andi rt, rs, imm # reg[rt] = reg[rs] & (zero)imm
```

op	rs	rt	imm
0 0 1 1 0 0	rs	rt	imm
6-bit	5-bit	5-bit	16-bit

命令の意味: レジスタ rs に格納されている値と符号なし即値 imm の論理積を取り，その結果をレジスタ rt に格納する。
例:

```
andi $8, $6, 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
```

12. ori 命令 (Or Immediate)

```
ori  rt, rs, imm # reg[rt] = reg[rs] | (zero)imm
```

op	rs	rt	imm
0 0 1 1 0 1	rs	rt	imm
6-bit	5-bit	5-bit	16-bit

命令の意味: レジスタ rs に格納されている値と符号なし即値 imm の論理和を取り，その結果をレジスタ rt に格納する。
例:

```
ori  $8, $6, 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 1111
```

13. xori 命令 (Exclusive Or Immediate)

```
xori rt, rs, imm # reg[rt] = reg[rs] ^ (zero)imm
```

op	rs	rt	imm
0 0 1 1 1 0	rs	rt	imm
6-bit	5-bit	5-bit	16-bit

命令の意味: レジスタ rs に格納されている値と符号なし即値 imm の排他的論理和を取り，その結果をレジスタ rt に格納する。
例:

```
xori $8, $6, 0xffff
```

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
```

14. lw 命令 (Load Word)

```
lw   rt, imm(rs) # reg[rt] = memory[reg[rs]+(sign)imm]
```

op	rs	rt	imm
1 0 0 0 1 1	rs	rt	imm
6-bit	5-bit	5-bit	16-bit

命令の意味: レジスタ rs 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 rt に格納する。例:

```
lw   $8, 4($6)
```

For example,

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

15. sw 命令 (Store Word)

```
sw   rt, imm(rs) # memory[reg[rs]+(sign)imm] = reg[rt]
```

op	rs	rt	imm
1 0 1 0 1 1	rs	rt	imm
6-bit	5-bit	5-bit	16-bit

命令の意味: レジスタ rs 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 rt に格納されている値を書き込む。例:

```
sw   $8, 4($6)
```

For example,

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

16. beq 命令 (Branch on Equal)

```
beq rs,rt,imm # if(reg[rs]==reg[rt]) PC=PC+4+(sign)imm*4
```

op	rs	rt	imm
0 0 0 1 0 0	rs	rt	imm
6-bit	5-bit	5-bit	16-bit

命令の意味: レジスタ rs に格納されている値とレジスタ rt 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 $\mathrm{PC} + 4 + (\mathrm{sign})\mathrm{imm} \times 4$ である。例:

```
beq  $8, $0, label
```

For example,

```
if   reg[8] == 0 (equal, note: reg[0] === 0)
then PC = PC + 4 + ((sign)imm<<2)
else PC = PC + 4
```

17. bne 命令 (Branch on Not Equal)

```
bne rs,rt,imm # if(reg[rs]!=reg[rt]) PC=PC+4+(sign)imm*4
```

op	rs	rt	imm
0 0 0 1 0 1	rs	rt	imm
6-bit	5-bit	5-bit	16-bit

命令の意味: レジスタ rs に格納されている値とレジスタ rt 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 $\mathrm{PC} + 4 + (\mathrm{sign})\mathrm{imm} \times 4$ である。例:

```
bne  $8, $0, label
```

For example,

```
if   reg[8] != 0 (not equal, note: reg[0] === 0)
then PC = PC + 4 + ((sign)imm<<2)
else PC = PC + 4
```

18. lui 命令 (Load Upper Immediate)

```
lui rt, imm   # reg[rt] = imm,0000000000000000
```

op	rs	rt	imm
001111	00000	rt	imm
6-bit	5-bit	5-bit	16-bit

命令の意味: 即値 imm を 16 ビット左に論理シフトして，その結果をレジスタ rt に格納する。
例:

```
lui  $8, 0xffff
```

For example,

```
reg[8] = 0xffff0000
```

19. j 命令 (Jump)

```
j    addr # PC = (PC+4)[31:28],addr,00
```

op	addr
0 0 0 0 1 0	addr
6-bit	26-bit

命令の意味: 26 ビットの即値を 2 ビット左にシフトした 28 ビット結果をジャンプ先アドレスの下位 28 ビットとしてジャンプする。ジャンプ先アドレスの上位 4 ビットは呼出し時の PC の上位 4 ビットと同じものを使う。例:

```
j     label
```

For example,

```
PC = label
```

20. jal 命令 (Jump And Link)

```
jal addr # PC = (PC+4)[31:28],addr,00; reg[31] = PC + 4;
```

op	addr
0 0 0 0 1 1	addr
6-bit	26-bit

命令の意味: サブルーチンをコールする。j 命令と同じくジャンプする。この際に PC + 4 の値をレジスタ 31 に格納する (戻りアドレス)。
例:

```
jal  subroutine
```

For example,

```
reg[31] = PC + 4
PC = subroutine
```

20 MIPS 命令

```
 1. add  rd, rs, rt   # reg[rd] = reg[rs]  +  reg[rt]
 2. sub  rd, rs, rt   # reg[rd] = reg[rs]  -  reg[rt]
 3. and  rd, rs, rt   # reg[rd] = reg[rs]  &  reg[rt]
 4. or   rd, rs, rt   # reg[rd] = reg[rs]  |  reg[rt]
 5. xor  rd, rs, rt   # reg[rd] = reg[rs]  ^  reg[rt]
 6. sll  rd, rt, sa   # reg[rd] = reg[rt]  << sa
 7. srl  rd, rt, sa   # reg[rd] = reg[rt]  >> sa (logical)
 8. sra  rd, rt, sa   # reg[rd] = reg[rt] >>> sa (arithmetic)
 9. jr   rs           #      PC = reg[rs]
10. addi rt, rs, imm  # reg[rt] = reg[rs]  +  (sign)imm
11. andi rt, rs, imm  # reg[rt] = reg[rs]  &  (zero)imm
12. ori  rt, rs, imm  # reg[rt] = reg[rs]  |  (zero)imm
13. xori rt, rs, imm  # reg[rt] = reg[rs]  ^  (zero)imm
14. lui  rt, imm      # reg[rt] = imm,0000000000000000
15. lw   rt, imm(rs)  # reg[rt] = memory[reg[rs]+(sign)imm]
16. sw   rt, imm(rs)  # memory[reg[rs]+(sign)imm] = reg[rt]
17. beq  rs, rt, imm  # if (reg[rs]==reg[rt]) PC=PC+4+(sign)imm*4
18. bne  rs, rt, imm  # if (reg[rs]!=reg[rt]) PC=PC+4+(sign)imm*4
19. j    addr         # PC = (PC+4)[31:28],addr,00
20. jal  addr         # PC = (PC+4)[31:28],addr,00; reg[31]=PC+4
```

MIPS アセンブリ言語プログラムの例

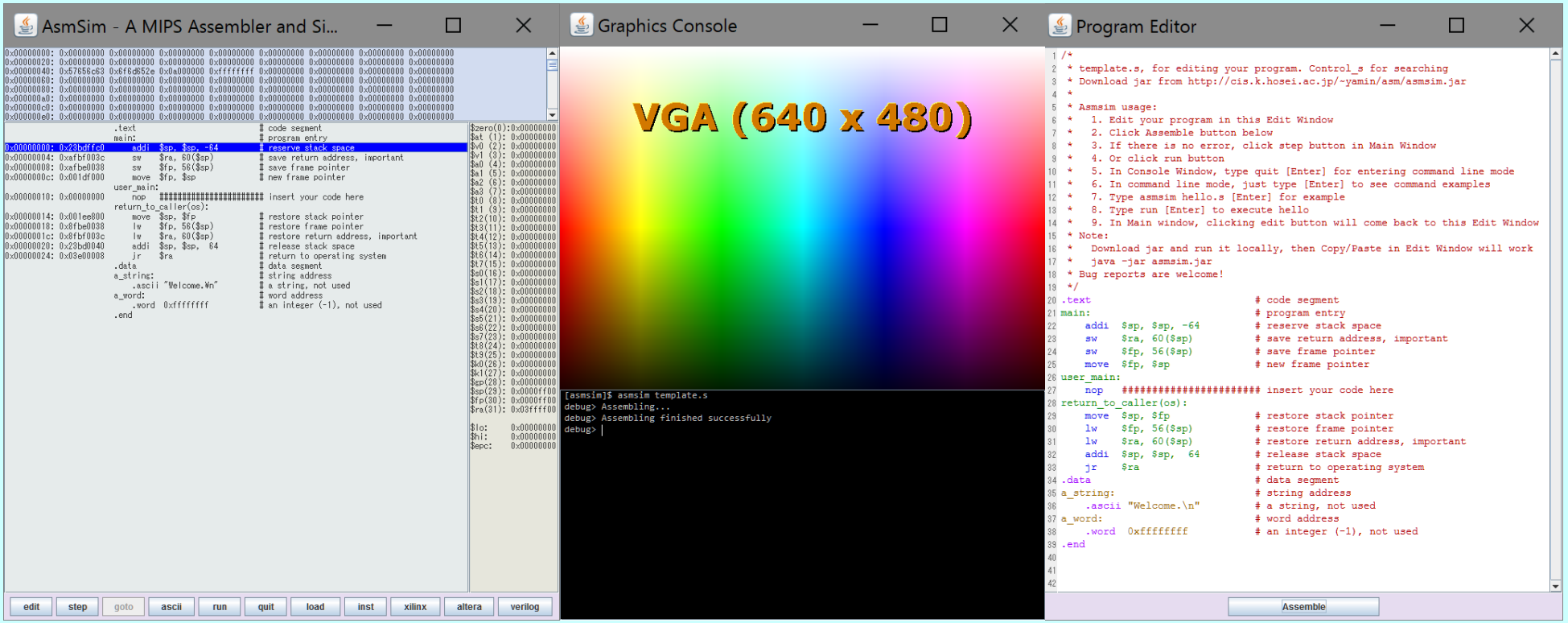
```
.text                       # code segment
main:                       # program entry
    addi $4, $0, 12         # reg[4] <= reg[0] + 12 = 12
    addi $5, $0, 13         # reg[5] <= reg[0] + 13 = 13
    add  $6, $4, $5         # reg[6] <= reg[4] + reg[5]
.end                        # end of program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

`.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`	MIPS 命令
`$0`, `$4`, `$5`, `$6`	MIPS レジスタ(レジスタ `$0` は常に 0 である)
`.end`	おわり

AsmSim ダウンロード

- MIPS アセンブリ言語プログラムを MIPS 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 x86 機械語しかわかりません
- 従って、MIPS シミュレータが必要になります
- AsmSim シミュレータをダウンロードします:
  https://yamin.cis.k.hosei.ac.jp/lectures/asm/asmsim.jar
- そして、asmsim.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 asmsim.jar`

AsmSim を実行



Main Window　　Console Window　　Editor Window

AsmSim の使い方については、
https://yamin.cis.k.hosei.ac.jp/lectures/asm/ を参照ください

add.s プログラムを編集

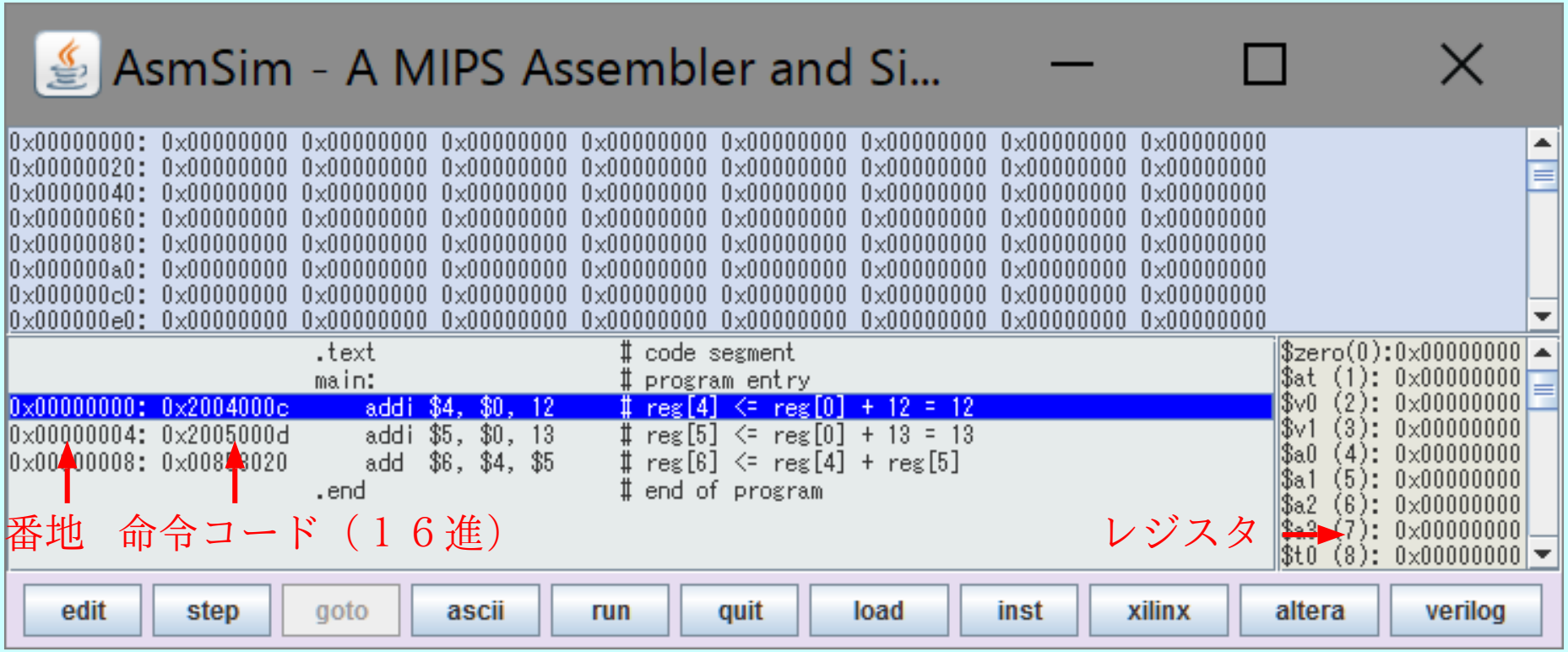
Program Editor

```
1 .text                    # code segment
2 main:                    # program entry
3     addi $4, $0, 12      # reg[4] <= reg[0] + 12 = 12
4     addi $5, $0, 13      # reg[5] <= reg[0] + 13 = 13
5     add  $6, $4, $5      # reg[6] <= reg[4] + reg[5]
6 .end                     # end of program
7
```

Assemble

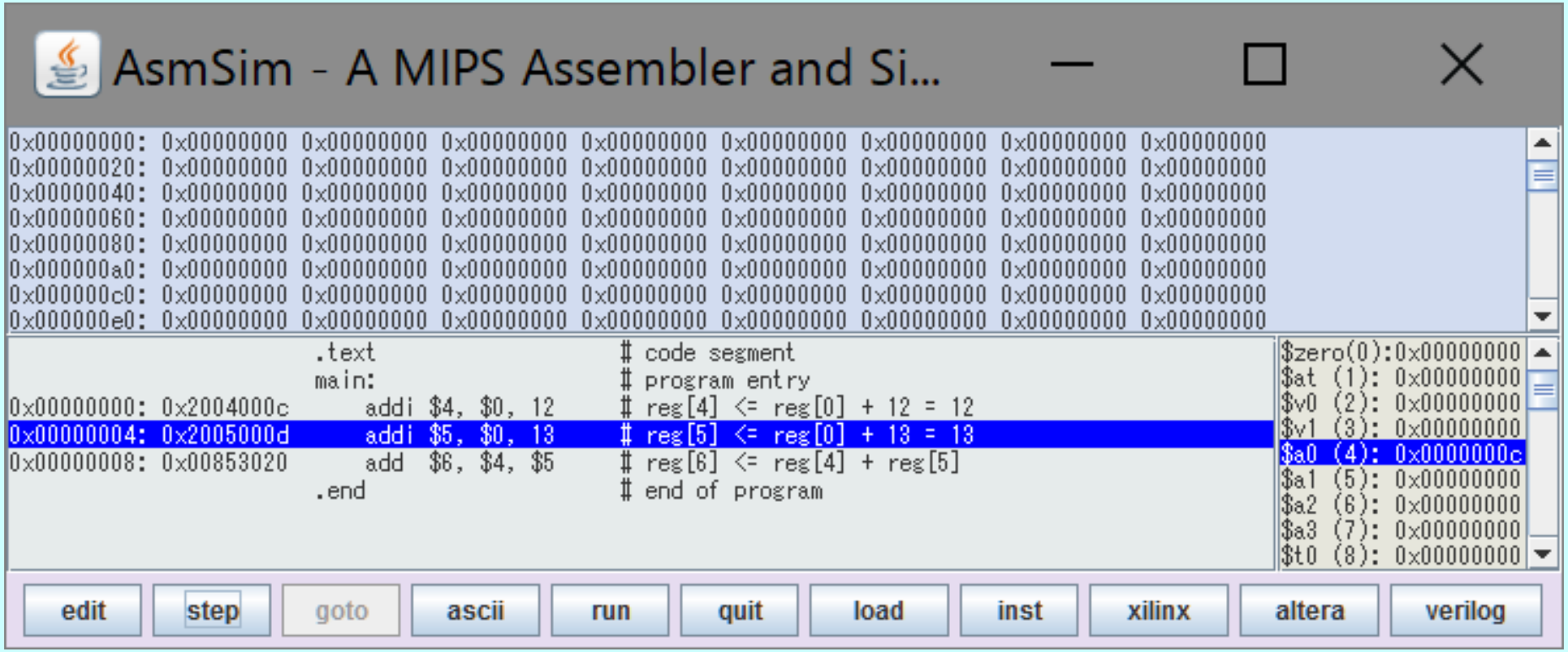
Assemble ボタンをクリック

add.s プログラムをステップで実行



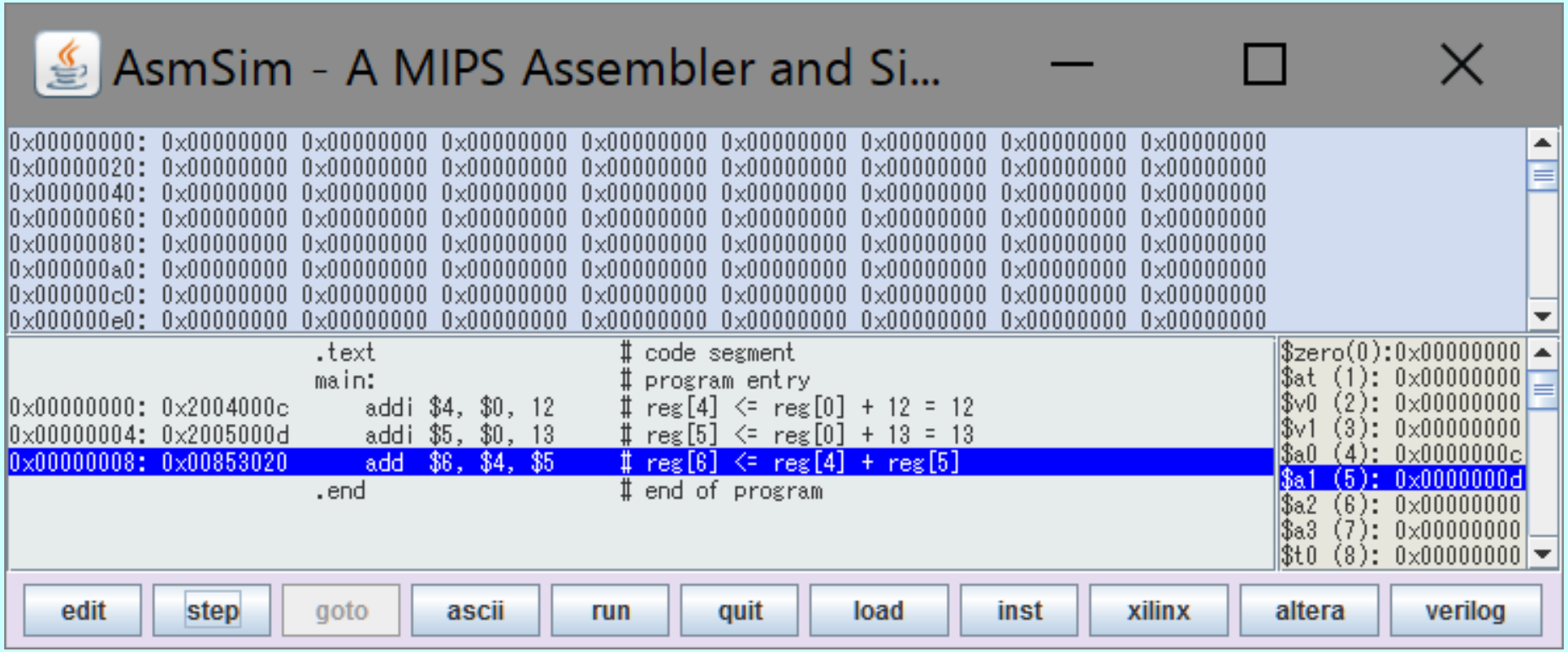
最初の状態。step ボタンをクリックすると、
`addi $4, $0, 12` を実行する

add.s プログラムをステップで実行



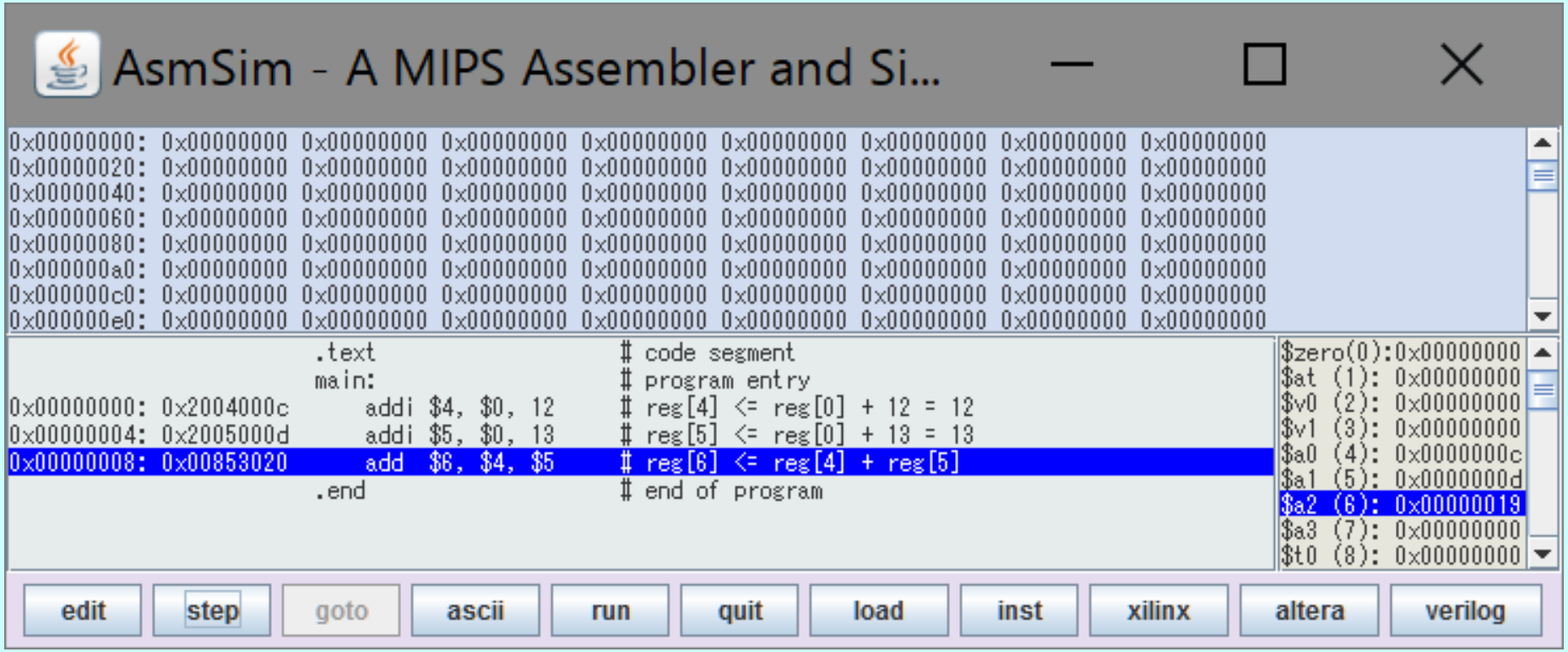
$4 = 12。step ボタンをクリックすると、
`addi $5, $0, 13` を実行する

add.s プログラムをステップで実行



$5 = 13。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



最後の結果、$6 = 25（16 進の 19 = 1 × 16 + 9 = 25）
つまり、12 + 13 = 25。CPU がプログラムを実行する際、こんな感じ

20 MIPS 命令

```
 1. add  rd, rs, rt   # reg[rd] = reg[rs]  +  reg[rt]
 2. sub  rd, rs, rt   # reg[rd] = reg[rs]  -  reg[rt]
 3. and  rd, rs, rt   # reg[rd] = reg[rs]  &  reg[rt]
 4. or   rd, rs, rt   # reg[rd] = reg[rs]  |  reg[rt]
 5. xor  rd, rs, rt   # reg[rd] = reg[rs]  ^  reg[rt]
 6. sll  rd, rt, sa   # reg[rd] = reg[rt]  << sa
 7. srl  rd, rt, sa   # reg[rd] = reg[rt]  >> sa (logical)
 8. sra  rd, rt, sa   # reg[rd] = reg[rt] >>> sa (arithmetic)
 9. jr   rs           #      PC = reg[rs]
10. addi rt, rs, imm  # reg[rt] = reg[rs]  +  (sign)imm
11. andi rt, rs, imm  # reg[rt] = reg[rs]  &  (zero)imm
12. ori  rt, rs, imm  # reg[rt] = reg[rs]  |  (zero)imm
13. xori rt, rs, imm  # reg[rt] = reg[rs]  ^  (zero)imm
14. lui  rt, imm      # reg[rt] = imm,0000000000000000
15. lw   rt, imm(rs)  # reg[rt] = memory[reg[rs]+(sign)imm]
16. sw   rt, imm(rs)  # memory[reg[rs]+(sign)imm] = reg[rt]
17. beq  rs, rt, imm  # if (reg[rs]==reg[rt]) PC=PC+4+(sign)imm*4
18. bne  rs, rt, imm  # if (reg[rs]!=reg[rt]) PC=PC+4+(sign)imm*4
19. j    addr         # PC = (PC+4)[31:28],addr,00
20. jal  addr         # PC = (PC+4)[31:28],addr,00; reg[31]=PC+4
```

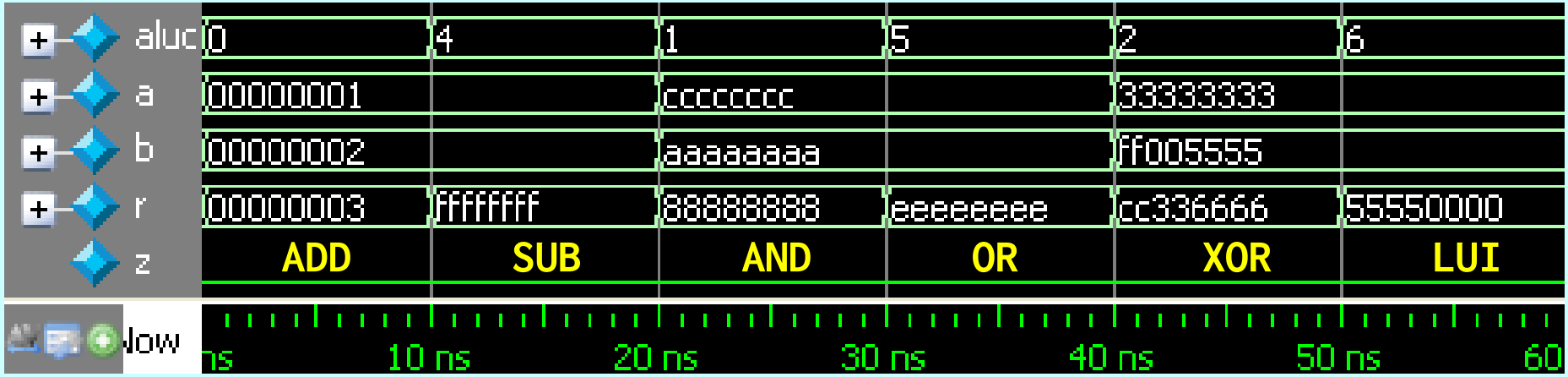
20 MIPS 命令 — 計算種類

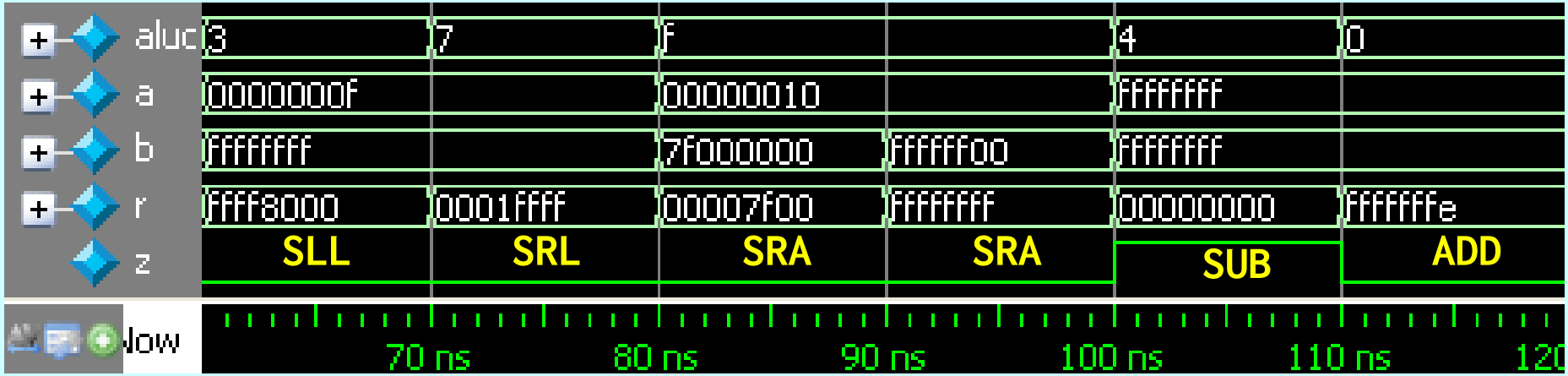
```
 1. add  rd, rs, rt   # ADD ........................................(1)
 2. sub  rd, rs, rt   # SUB ........................................(2)
 3. and  rd, rs, rt   # AND ........................................(3)
 4. or   rd, rs, rt   # OR  ........................................(4)
 5. xor  rd, rs, rt   # XOR ........................................(5)
 6. sll  rd, rt, sa   # SLL ........................................(6)
 7. srl  rd, rt, sa   # SRL ........................................(7)
 8. sra  rd, rt, sa   # SRA ........................................(8)
 9. jr   rs           # X   (No calculation)
10. addi rt, rs, imm  # Same as ADD.................................(1)
11. andi rt, rs, imm  # Same as AND.................................(3)
12. ori  rt, rs, imm  # Same as OR ..................................(4)
13. xori rt, rs, imm  # Same as XOR.................................(5)
14. lui  rt, imm      # LUI ........................................(9)
15. lw   rt, imm(rs)  # Same as ADD.................................(1)
16. sw   rt, imm(rs)  # Same as ADD.................................(1)
17. beq  rs, rt, imm  # Same as SUB.................................(2)
18. bne  rs, rt, imm  # Same as SUB.................................(2)
19. j    addr         # X   (No calculation)
20. jal  addr         # X   (No calculation)
```

ALU の回路設計 alu.v

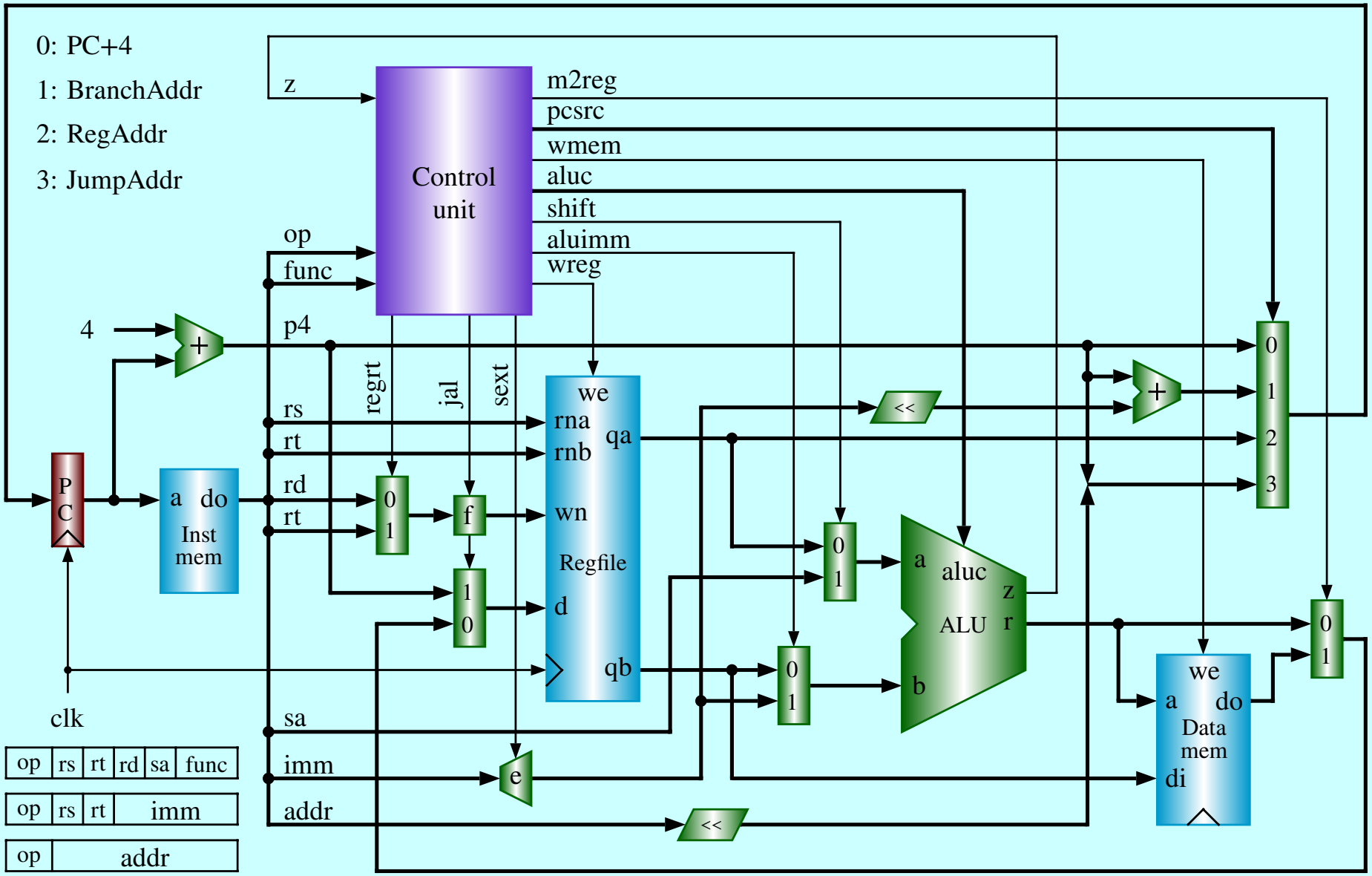
```
module alu (a,b,aluc,r,z);            // 32-bit alu with a zero flag
    input  [31:0] a, b;               // inputs: a, b
    input   [3:0] aluc;               // input:  alu control: // aluc[3:0]:
    output [31:0] r;                  // output: alu result   // x 0 0 0  ADD
    output        z;                  // output: zero flag    // x 1 0 0  SUB
    wire   [31:0] d_and = a & b;                              // x 0 0 1  AND
    wire   [31:0] d_or  = a | b;                              // x 1 0 1  OR
    wire   [31:0] d_xor = a ^ b;                              // x 0 1 0  XOR
    wire   [31:0] d_lui = {b[15:0],16'h0};                    // x 1 1 0  LUI
    wire   [31:0] d_and_or  = aluc[2]? d_or  : d_and;         // 0 0 1 1  SLL
    wire   [31:0] d_xor_lui = aluc[2]? d_lui : d_xor;         // 0 1 1 1  SRL
    wire   [31:0] d_as, d_sh;                                 // 1 1 1 1  SRA

    // addsub32   (a,b,sub,    s);
    addsub32 as32 (a,b,aluc[2],d_as);                         // add/sub

    // shift      (d,sa,    right,  arith,  sh);
    shift shifter (b,a[4:0],aluc[2],aluc[3],d_sh);            // shift

    // mux4x32  (a0,  a1,      a2,       a3,  s,        y);
    mux4x32 res (d_as,d_and_or,d_xor_lui,d_sh,aluc[1:0],r);   // alu result

    assign z = ~|r;                                           // z = (r == 0)
endmodule
```

ALU のシミュレーション波形





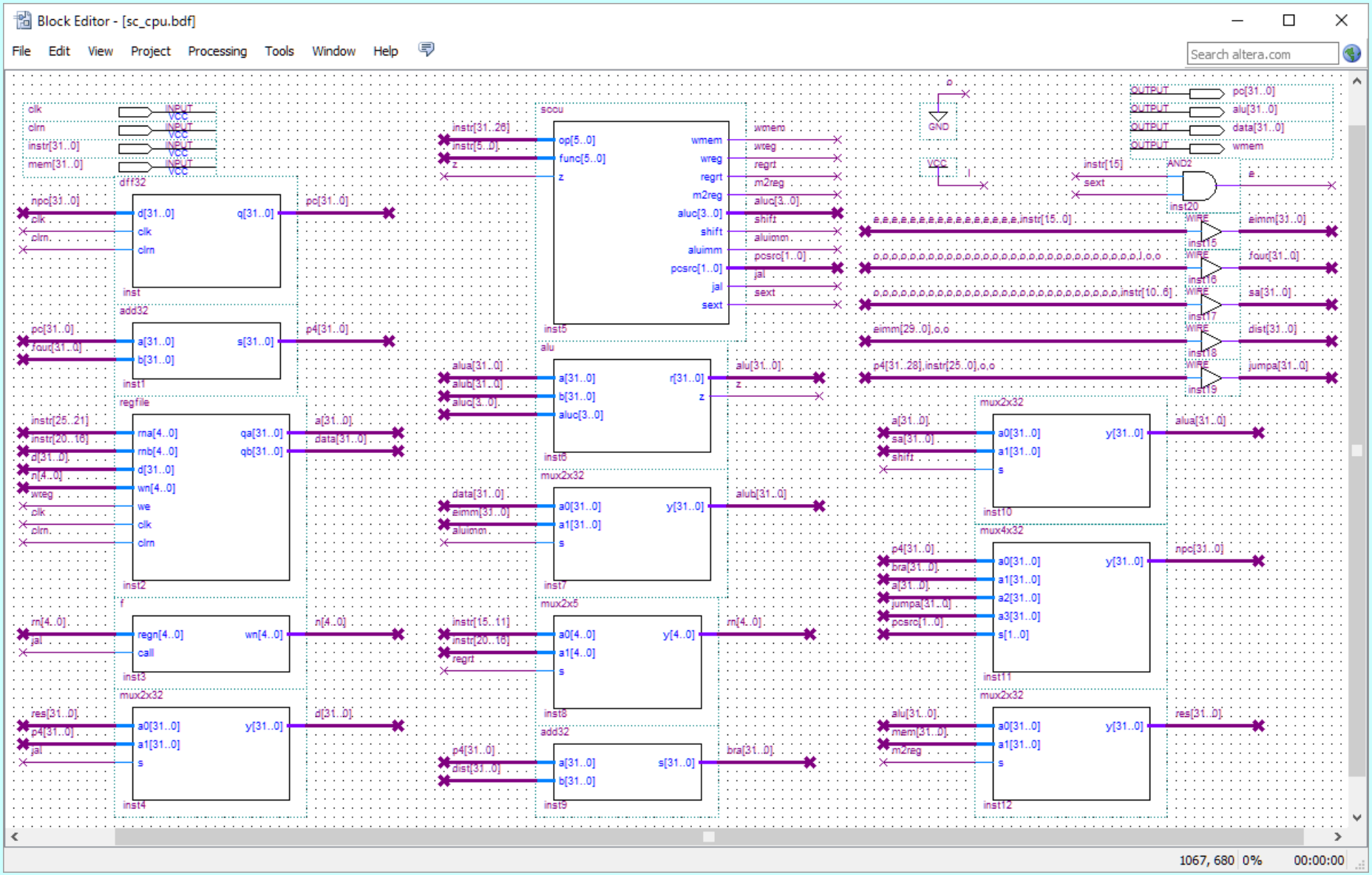
単一サイクル CPU 全体の回路図



制御ユニット設計の回路

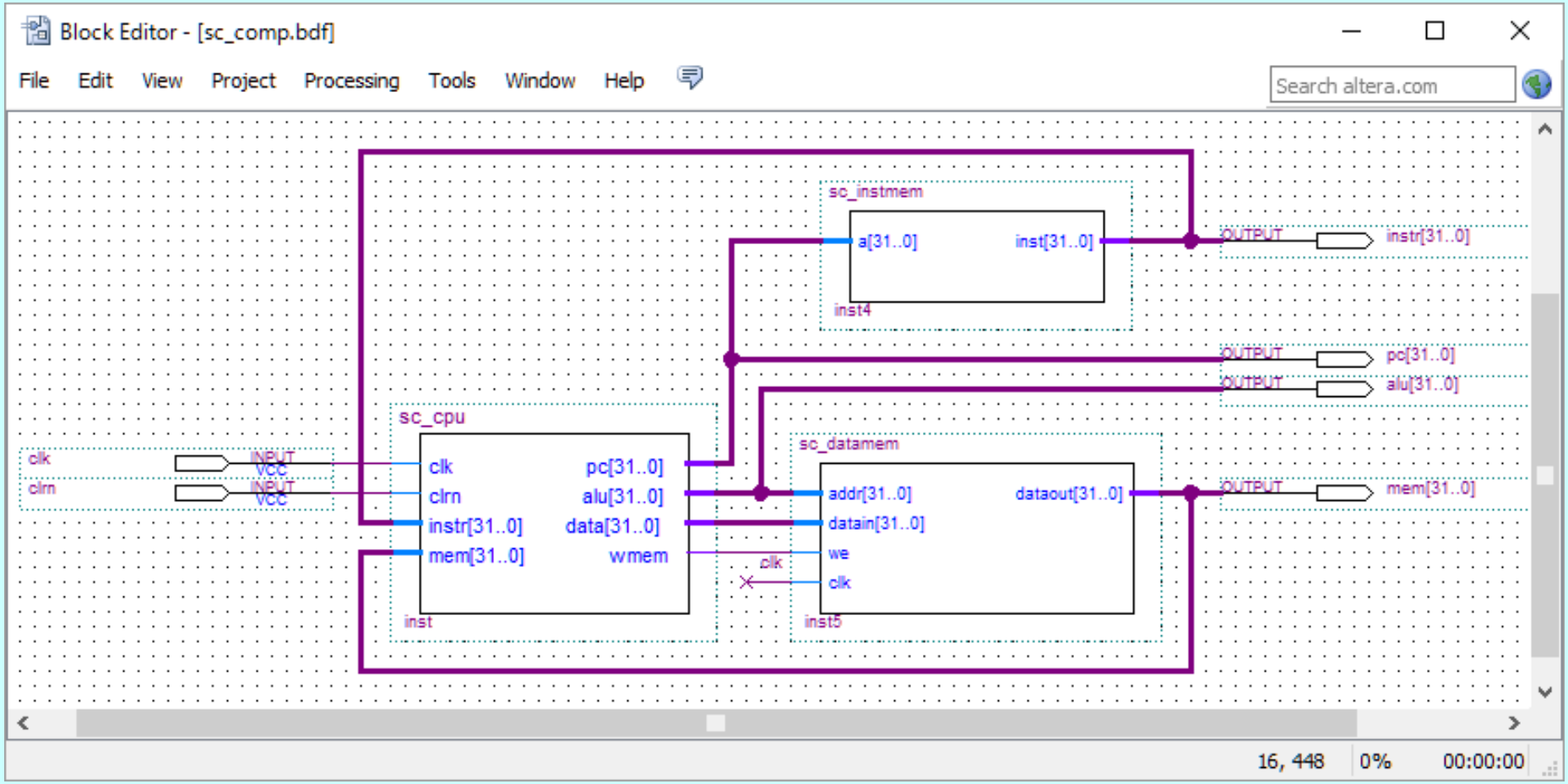
```
module sccu (op,func,z,wmem,wreg,regrt,m2reg,aluc,shift,aluimm,pcsrc,jal,sext);//control unit
  input  [5:0] op, func;                          // op, func
  input        z;                                 // alu zero tag
  output [3:0] aluc;                              // alu operation control
  output [1:0] pcsrc;                             // select pc source
  output       wreg;                              // write regfile
  output       regrt;                             // dest reg number is rt
  output       m2reg;                             // instruction is an lw
  output       shift;                             // instruction is a shift
  output       aluimm;                            // alu input b is an i32
  output       jal;                               // instruction is a jal
  output       sext;                              // is sign extension
  output       wmem;                              // write data memory
  // decode instructions
  wire rtype  = ~|op;                                          // r format
  wire i_add  = rtype & func[5] &~func[4] &~func[3] &~func[2] &~func[1] &~func[0];
  // ...
  wire i_addi = ~op[5] &~op[4] & op[3] &~op[2] &~op[1] &~op[0];  // i format
  // ...
  assign aluc[3] = i_sra;
  assign wreg = i_add | i_sub  | i_and | i_or   | i_xor  |
                i_sll | i_srl  | i_sra | i_addi | i_andi |
                i_ori | i_xori | i_lui | i_lw   | i_jal;
  // ...
endmodule
```

CPU 全体の回路図


Block Editor - [sc_cpu.bdf]
File Edit View Project Processing Tools Window Help
Search altera.com
clk
clrn
instr[31..0]
mem[31..0]
INPUT VCC
dff32
npc[31..0]
d[31..0]
q[31..0]
pc[31..0]
inst
add32
a[31..0]
s[31..0]
b[31..0]
four[31..0]
p4[31..0]
inst1
regfile
instr[25..21]
instr[20..16]
rna[4..0]
rnb[4..0]
wn[4..0]
qa[31..0]
qb[31..0]
a[31..0]
data[31..0]
wreg
we
inst2
regn[4..0]
call
jal
inst3
mux2x32
res[31..0]
a0[31..0]
a1[31..0]
y[31..0]
inst4
scu
op[5..0]
func[5..0]
z
wmem
wreg
regrt
m2reg
aluc[3..0]
shift
aluimm
pcsrc[1..0]
sext
inst5
alu
alua[31..0]
alub[31..0]
inst6
mux2x32
eimm[31..0]
inst7
mux2x5
rd[4..0]
a0[4..0]
a1[4..0]
y[4..0]
inst8
add32
dist[31..0]
bra[31..0]
inst9
GND
VCC
AND2
inst20
OUTPUT
pc[31..0]
alu[31..0]
data[31..0]
wmem
inst15
inst16
inst17
inst18
inst19
WIRE
sa[31..0]
jumpa[31..0]
mux2x32
inst10
mux4x32
npc[31..0]
inst11
inst12
1067, 680 0% 00:00:00

CPU とメモリの回路

単一サイクル MIPS CPU + 命令メモリ + データメモリ



テスト・プログラム

```
module scinstmem (a,inst);                  // instruction memory, rom
    input  [31:0] a;                        // byte address
    output [31:0] inst;                     // instruction

    wire   [31:0] rom [0:31];               // rom cells: 32 words * 32 bits

    // rom[word_addr] = instruction         // (pc) label   instruction

    assign rom[5'h00] = 32'h3c010000;       // (00) main:   lui  $1, 0
    assign rom[5'h01] = 32'h34240050;       // (04)         ori  $4, $1, 80
    assign rom[5'h02] = 32'h20050004;       // (08)         addi $5, $0,  4
    assign rom[5'h03] = 32'h0c000018;       // (0c) call:   jal  sum
    assign rom[5'h04] = 32'hac820000;       // (10)         sw   $2, 0($4)
    assign rom[5'h05] = 32'h8c890000;       // (14)         lw   $9, 0($4)
    assign rom[5'h06] = 32'h01244022;       // (18)         sub  $8, $9, $4
    assign rom[5'h07] = 32'h20050003;       // (1c)         addi $5, $0,  3
    assign rom[5'h08] = 32'h20a5ffff;       // (20) loop2:  addi $5, $5, -1
    assign rom[5'h09] = 32'h34a8ffff;       // (24)         ori  $8, $5, 0xffff
    assign rom[5'h0A] = 32'h39085555;       // (28)         xori $8, $8, 0x5555
    assign rom[5'h0B] = 32'h2009ffff;       // (2c)         addi $9, $0, -1
    assign rom[5'h0C] = 32'h312affff;       // (30)         andi $10,$9, 0xffff
    assign rom[5'h0D] = 32'h01493025;       // (34)         or   $6, $10, $9
```

テスト・プログラム

```
    assign rom[5'h0E] = 32'h01494026;    // (38)         xor  $8, $10, $9
    assign rom[5'h0F] = 32'h01463824;    // (3c)         and  $7, $10, $6
    assign rom[5'h10] = 32'h10a00001;    // (40)         beq  $5, $0, shift
    assign rom[5'h11] = 32'h08000008;    // (44)         j    loop2
    assign rom[5'h12] = 32'h2005ffff;    // (48) shift:  addi $5, $0, -1
    assign rom[5'h13] = 32'h000543c0;    // (4c)         sll  $8, $5, 15
    assign rom[5'h14] = 32'h00084400;    // (50)         sll  $8, $8, 16
    assign rom[5'h15] = 32'h00084403;    // (54)         sra  $8, $8, 16
    assign rom[5'h16] = 32'h000843c2;    // (58)         srl  $8, $8, 15
    assign rom[5'h17] = 32'h08000017;    // (5c) finish: j    finish
    assign rom[5'h18] = 32'h00004020;    // (60) sum:    add  $8, $0, $0
    assign rom[5'h19] = 32'h8c890000;    // (64) loop:   lw   $9, 0($4)
    assign rom[5'h1A] = 32'h20840004;    // (68)         addi $4, $4,  4
    assign rom[5'h1B] = 32'h01094020;    // (6c)         add  $8, $8, $9
    assign rom[5'h1C] = 32'h20a5ffff;    // (70)         addi $5, $5, -1
    assign rom[5'h1D] = 32'h14a0fffb;    // (74)         bne  $5, $0, loop
    assign rom[5'h1E] = 32'h00081000;    // (78)         sll  $2, $8, 0
    assign rom[5'h1F] = 32'h03e00008;    // (7c)         jr   $31

    assign inst = rom[a[6:2]];           // use word address to read rom

endmodule
```

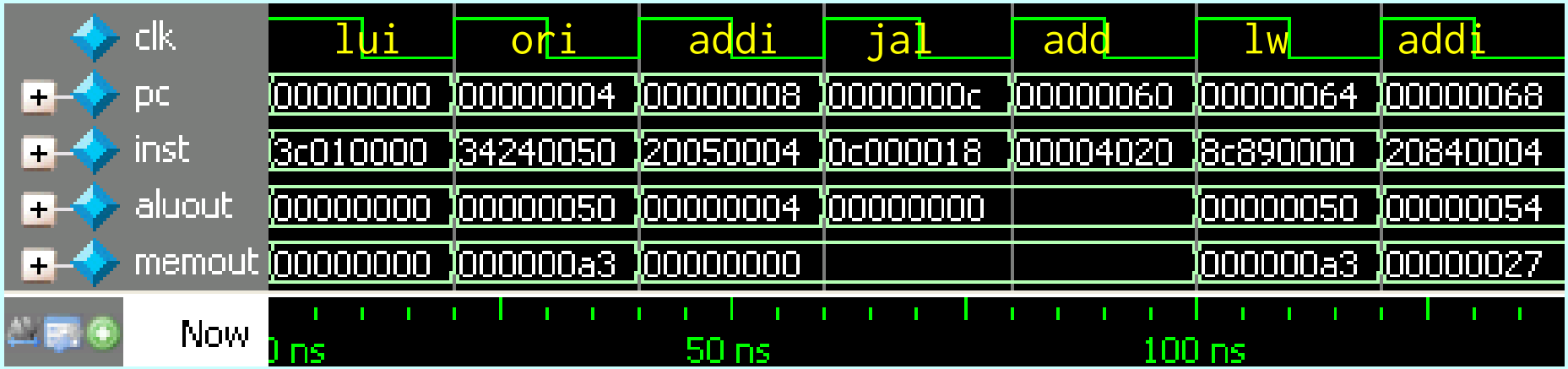
テスト・データ

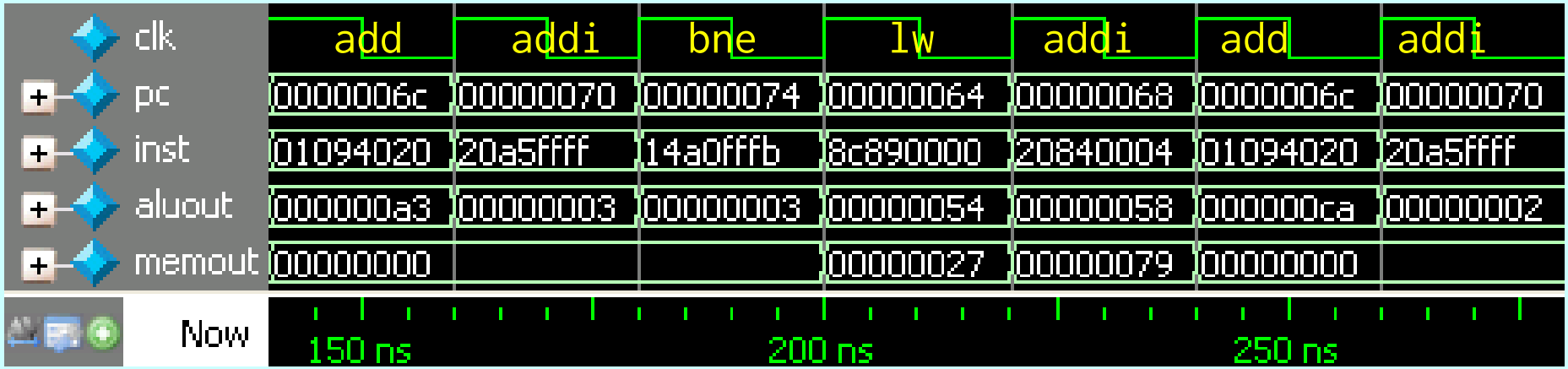
```
module scdatamem (addr,datain,we,clk,dataout);        // data memory, ram
    input         clk;                    // clock
    input         we;                     // write enable
    input  [31:0] datain;                 // data in (to memory)
    input  [31:0] addr;                   // ram address
    output [31:0] dataout;                // data out (from memory)
    reg    [31:0] ram [0:31];             // ram cells: 32 words * 32 bits
    assign dataout = ram[addr[6:2]];      // use word address to read ram
    always @ (posedge clk)
        if (we) ram[addr[6:2]] = datain; // use word address to write ram
    integer i;
    initial begin                         // initialize memory
        for (i = 0; i < 32; i = i + 1)
            ram[i] = 0;
        // ram[word_addr] = data          // (byte_addr) item in data array
        ram[5'h14] = 32'h000000a3;        // (50)  data[0]    0 +  A3 =  A3
        ram[5'h15] = 32'h00000027;        // (54)  data[1]   a3 +  27 =  ca
        ram[5'h16] = 32'h00000079;        // (58)  data[2]   ca +  79 = 143
        ram[5'h17] = 32'h00000115;        // (5c)  data[3]  143 + 115 = 258
        // ram[5'h18] should be 0x00000258, the sum stored by sw instruction
    end
endmodule
```

テストベンチ

```
`timescale 1ns/1ns
module sc_comp_tb;
    reg         clk,clrn;
    wire [31:0] inst,pc,aluout,memout;

    sc_comp cpu (.clk(clk),
                 .clrn(clrn),
                 .instr(inst),
                 .pc(pc),
                 .alu(aluout),
                 .mem(memout));

    initial begin
              clrn = 0;
              clk  = 1;
        #1    clrn = 1;
        #1399 $stop; // 1400 ns
    end

    always #10 clk = !clk;

endmodule
```

シミュレーション


clk
pc
inst
aluout
memout
Now
lui
ori
addi
jal
add
lw
addi
00000000
00000004
00000008
0000000c
00000060
00000064
00000068
3c010000
34240050
20050004
0c000018
00004020
8c890000
20840004
00000000
00000050
00000004
00000000
00000050
00000054
00000000
000000a3
00000000
000000a3
00000027
0 ns
50 ns
100 ns

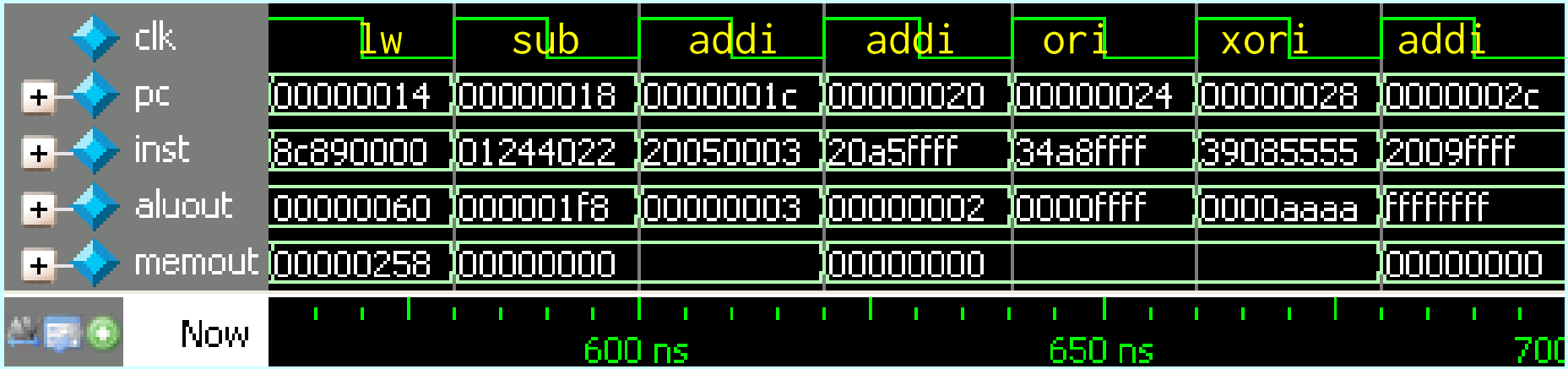

clk
pc
inst
aluout
memout
Now
add
addi
bne
lw
addi
add
addi
0000006c
00000070
00000074
00000064
00000068
0000006c
00000070
01094020
20a5ffff
14a0fffb
8c890000
20840004
01094020
20a5ffff
000000a3
00000003
00000003
00000054
00000058
000000ca
00000002
00000000
00000027
00000079
00000000
150 ns
200 ns
250 ns

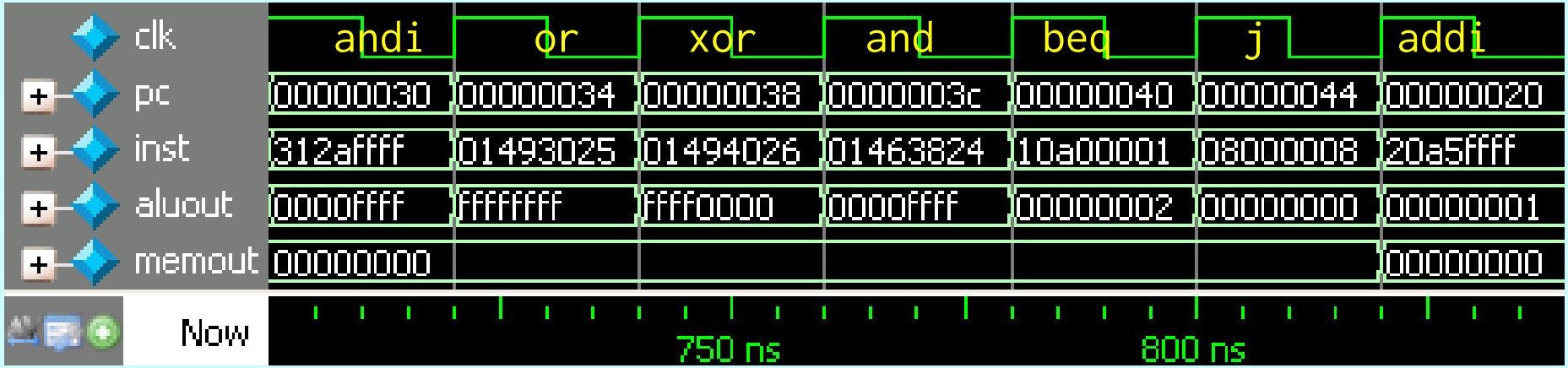
シミュレーション

clk	bne	lw	addi	add	addi	bne	lw
pc	00000074	00000064	00000068	0000006c	00000070	00000074	00000064
inst	14a0fffb	8c890000	20840004	01094020	20a5ffff	14a0fffb	8c890000
aluout	00000002	00000058	0000005c	00000143	00000001	00000001	0000005c
memout	00000000	00000079	00000115	00000000			00000115
Now	300 ns			350 ns			400 ns

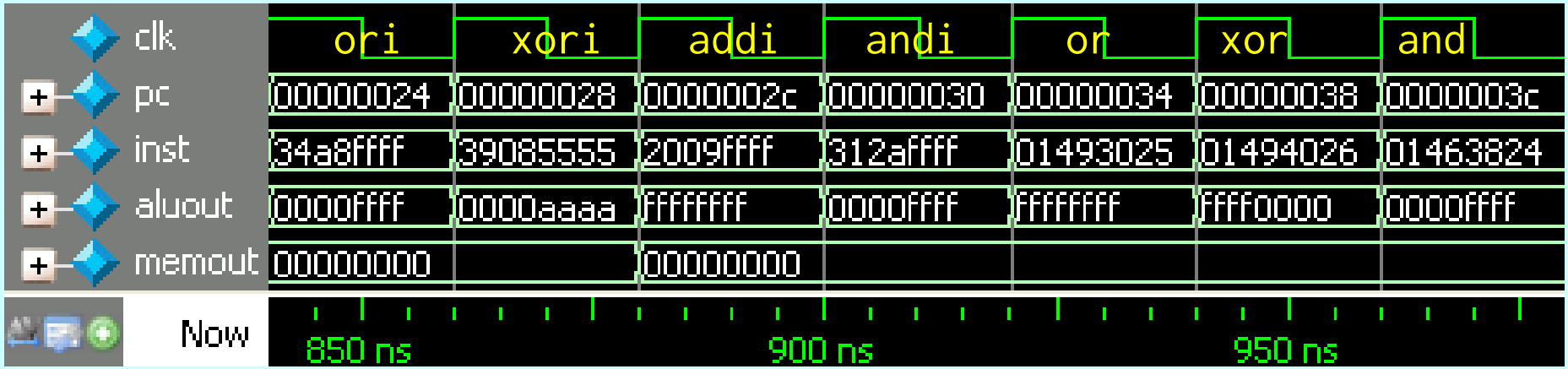
clk	addi	add	addi	bne	sll	jr	sw
pc	00000068	0000006c	00000070	00000074	00000078	0000007c	00000010
inst	20840004	01094020	20a5ffff	14a0fffb	00081000	03e00008	ac820000
aluout	00000060	00000258	00000000	00000000	00000258	00000010	00000060
memout	00000000	00000079	00000000		00000079	00000000	
Now	450 ns			500 ns			550 ns

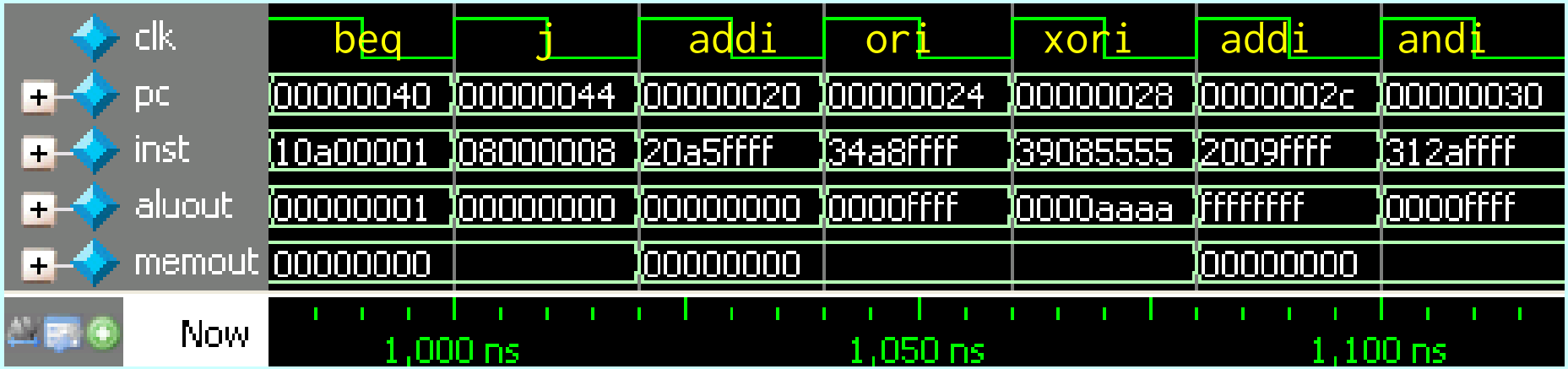
シミュレーション


clk
pc
inst
aluout
memout
Now
lw
sub
addi
addi
ori
xori
addi
00000014
00000018
0000001c
00000020
00000024
00000028
0000002c
8c890000
01244022
20050003
20a5ffff
34a8ffff
39085555
2009ffff
00000060
000001f8
00000003
00000002
0000ffff
0000aaaa
ffffffff
00000258
00000000
00000000
00000000
600 ns
650 ns

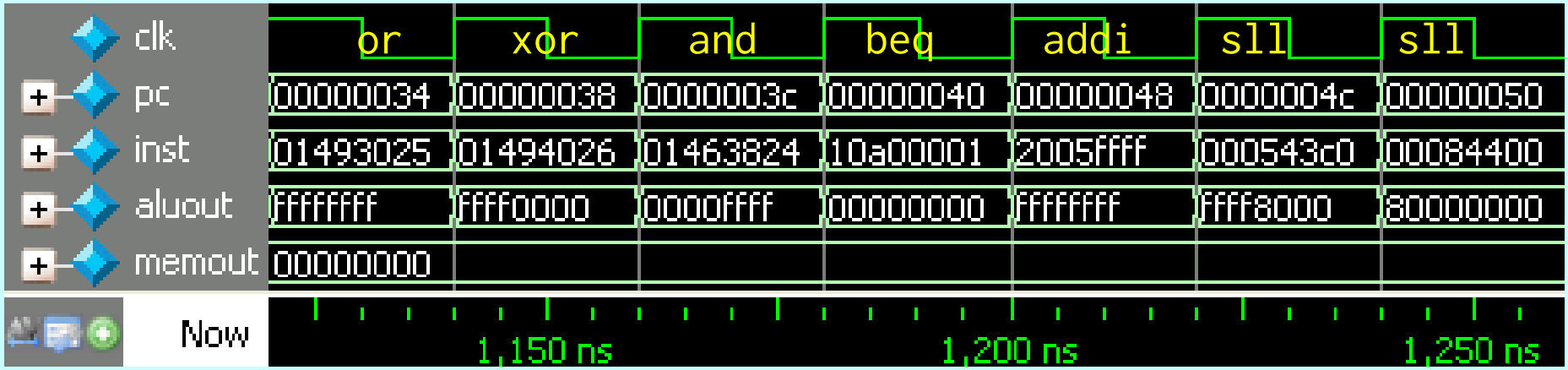

clk
pc
inst
aluout
memout
Now
andi
or
xor
and
beq
j
addi
00000030
00000034
00000038
0000003c
00000040
00000044
00000020
312affff
01493025
01494026
01463824
10a00001
08000008
20a5ffff
0000ffff
ffffffff
ffff0000
0000ffff
00000002
00000000
00000001
00000000
00000000
750 ns
800 ns

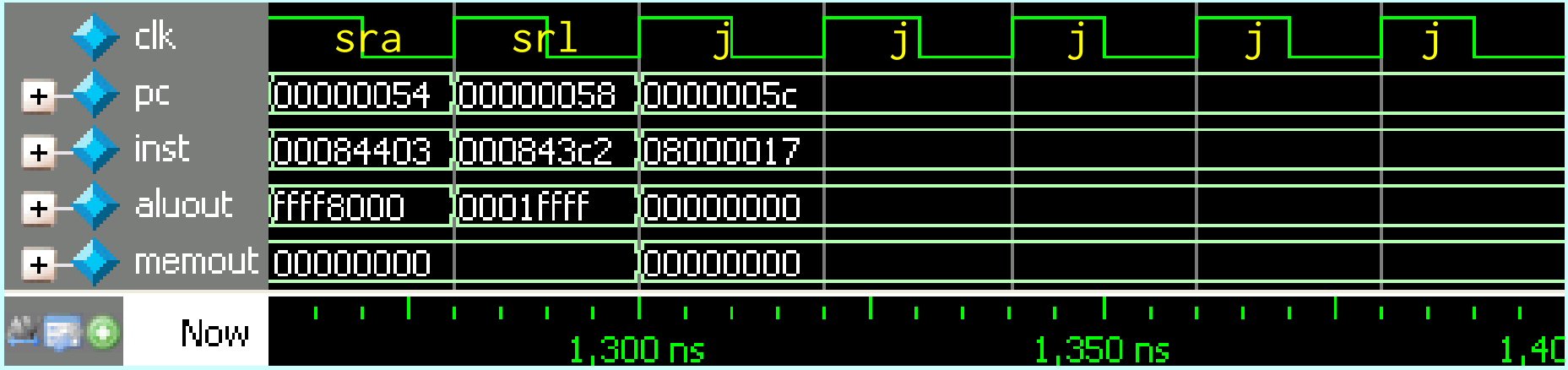
シミュレーション


clk
pc
inst
aluout
memout
Now
ori
xori
addi
andi
or
xor
and
00000024
00000028
0000002c
00000030
00000034
00000038
0000003c
34a8ffff
39085555
2009ffff
312affff
01493025
01494026
01463824
0000ffff
0000aaaa
ffffffff
0000ffff
ffffffff
ffff0000
0000ffff
00000000
00000000
850 ns
900 ns
950 ns

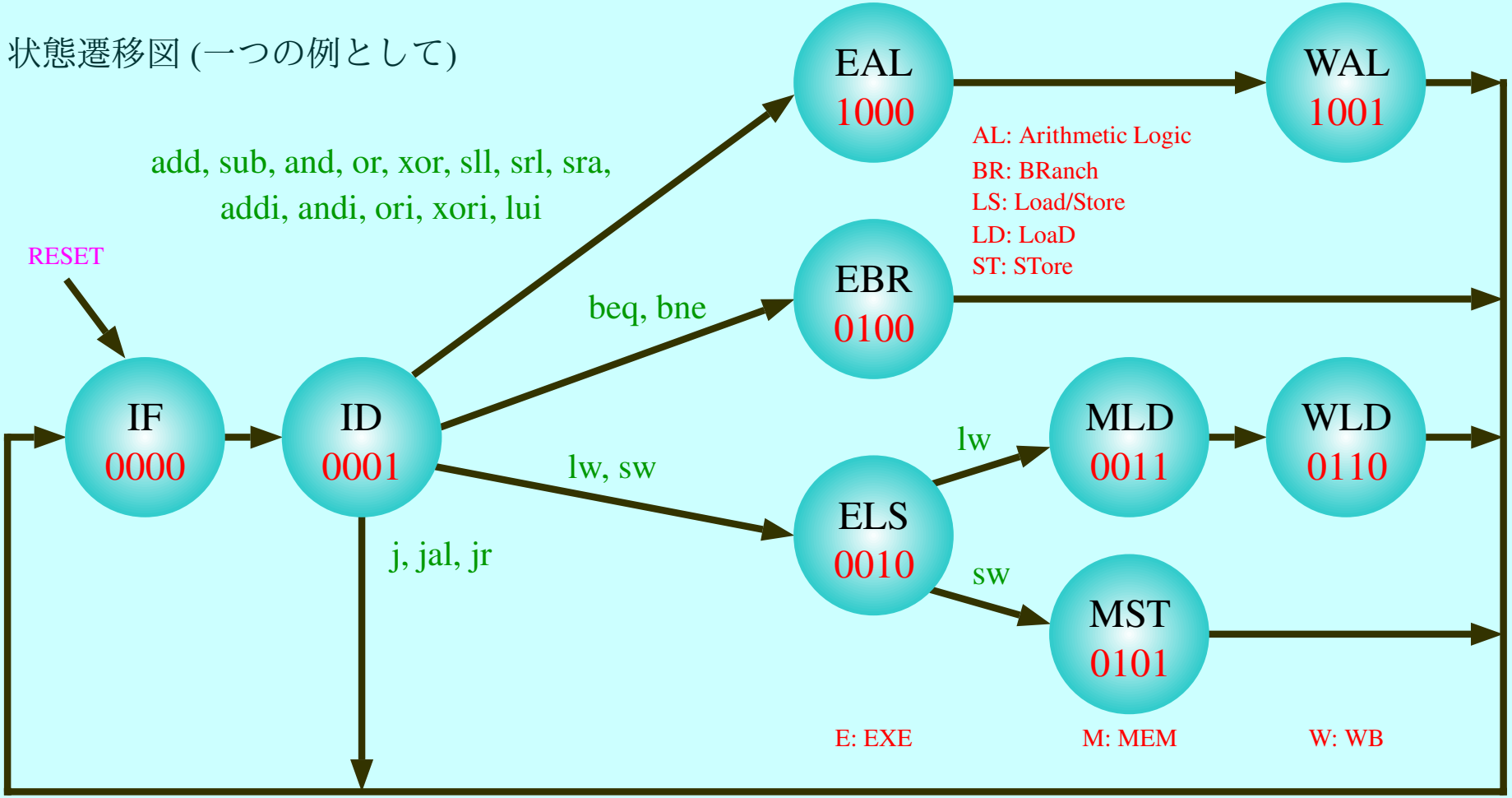

clk
pc
inst
aluout
memout
Now
beq
j
addi
ori
xori
addi
andi
00000040
00000044
00000020
00000024
00000028
0000002c
00000030
10a00001
08000008
20a5ffff
34a8ffff
39085555
2009ffff
312affff
00000001
00000000
00000000
0000ffff
0000aaaa
ffffffff
0000ffff
00000000
00000000
00000000
1,000 ns
1,050 ns
1,100 ns

シミュレーション


clk
pc
inst
aluout
memout
Now
or
xor
and
beq
addi
sll
sll
00000034
00000038
0000003c
00000040
00000048
0000004c
00000050
01493025
01494026
01463824
10a00001
2005ffff
000543c0
00084400
ffffffff
ffff0000
0000ffff
00000000
ffffffff
ffff8000
80000000
00000000
1,150 ns
1,200 ns
1,250 ns

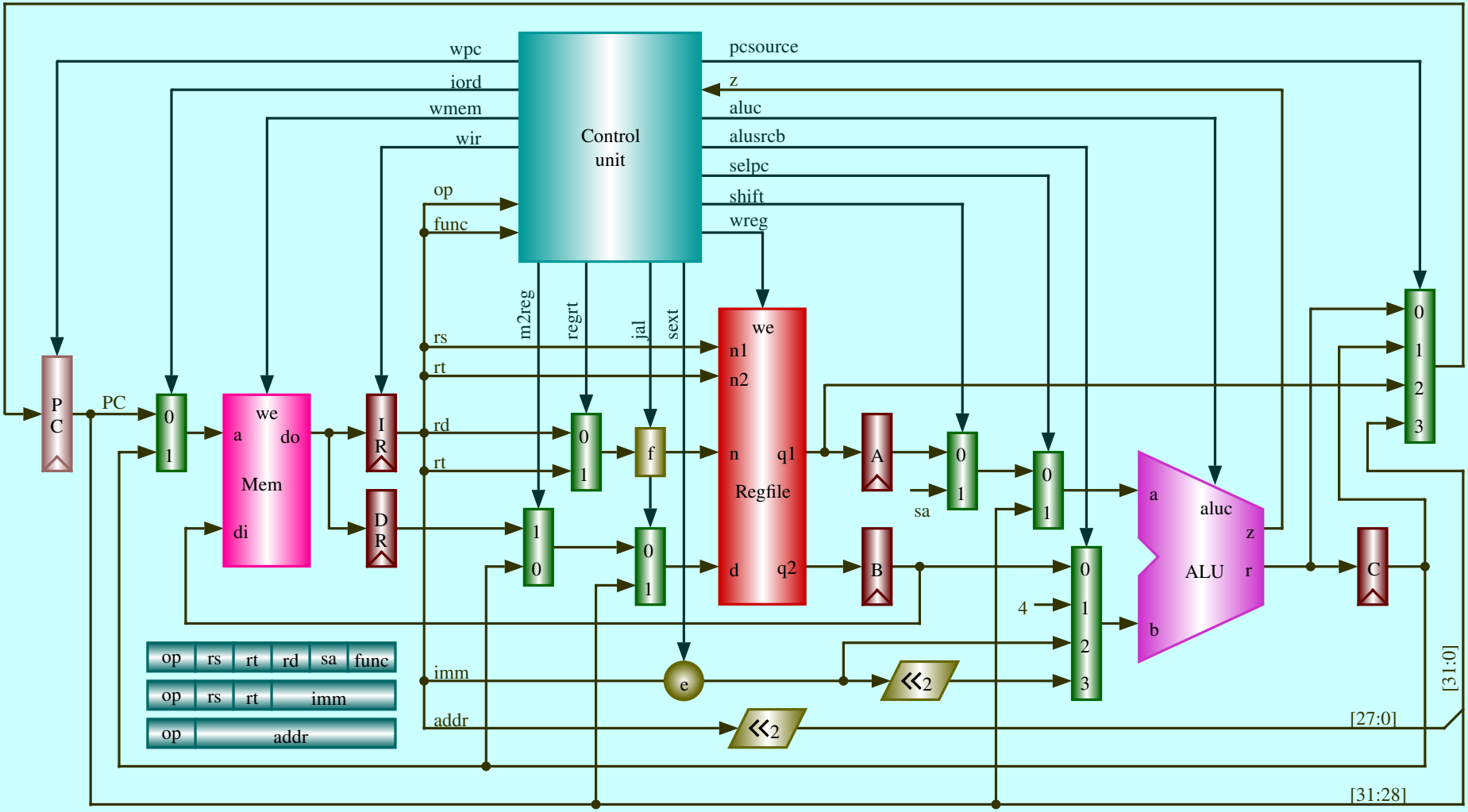

clk
pc
inst
aluout
memout
Now
sra
srl
j
j
j
j
j
00000054
00000058
0000005c
00084403
000843c2
08000017
ffff8000
0001ffff
00000000
00000000
00000000
1,300 ns
1,350 ns

マルチサイクルの実装：有限状態機械

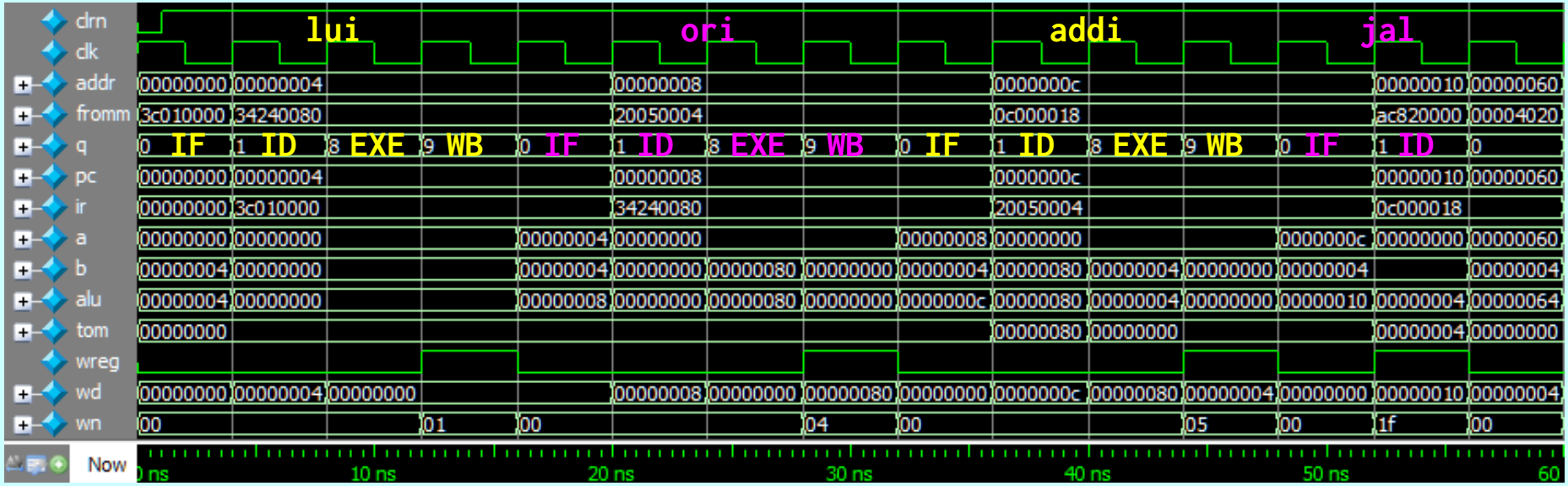


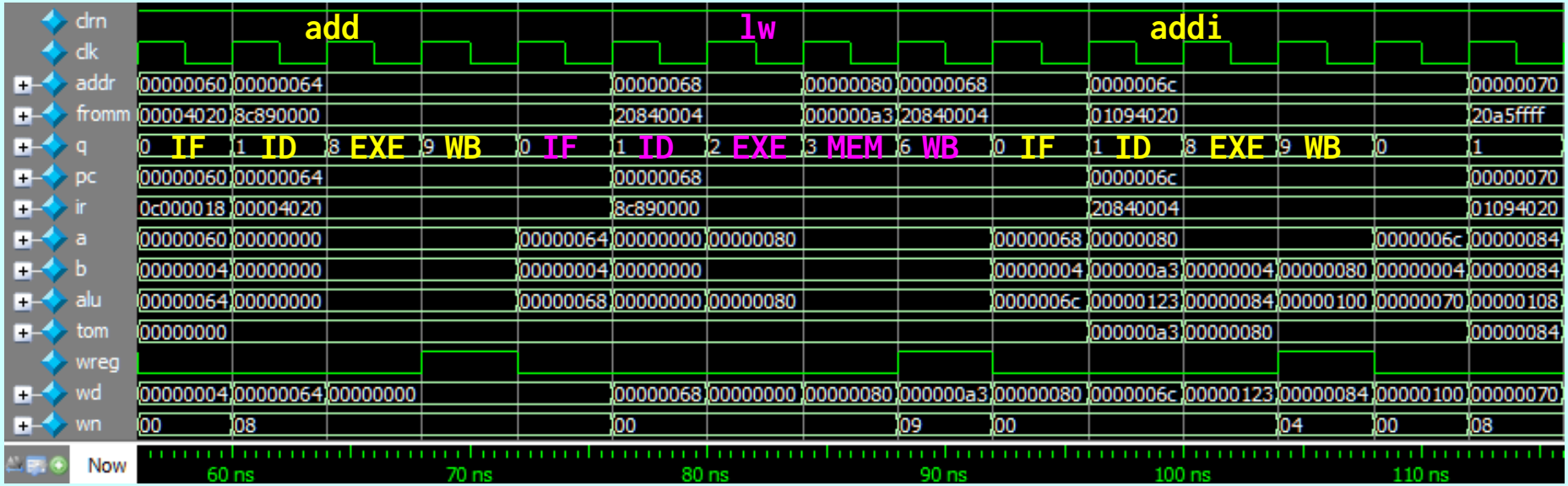
j, jal, jr: 2 サイクル、beq, bne: 3 サイクル、lw: 5 サイクル、それ以外: 4 サイクル

マルチサイクル MIPS CPU + メモリ

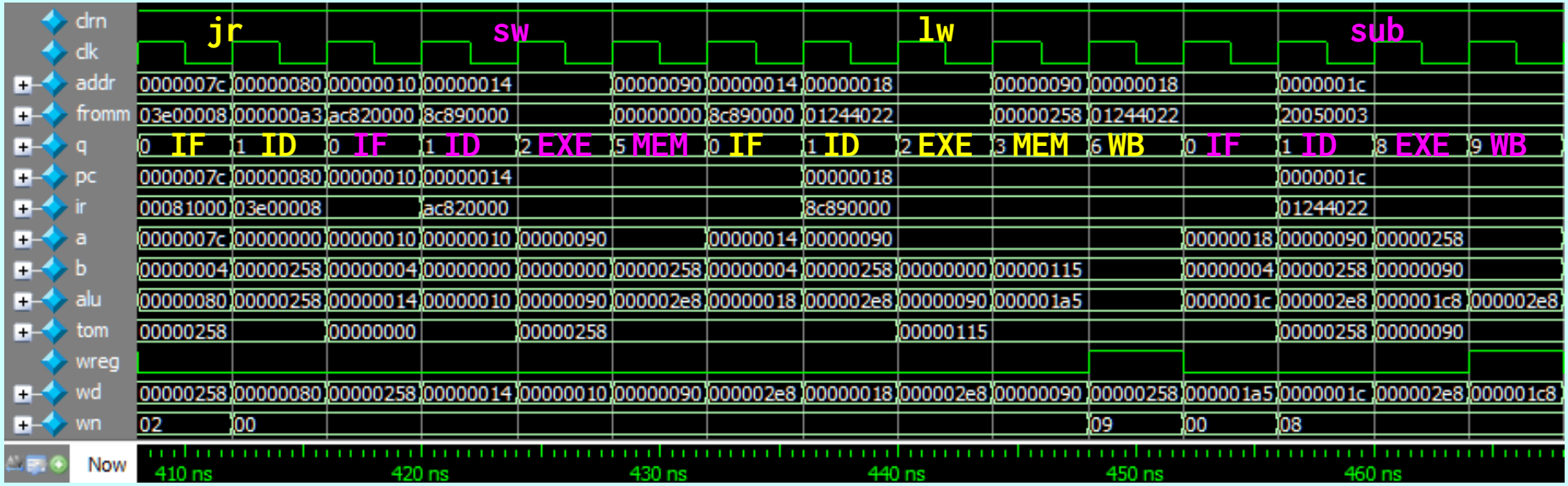


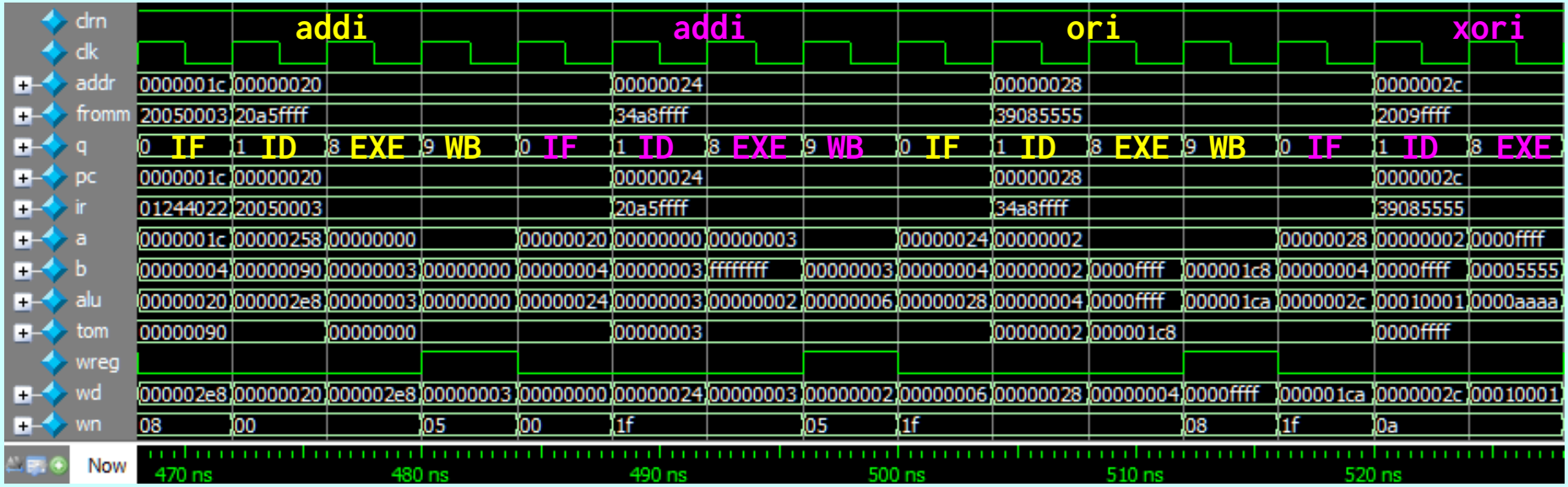
マルチサイクル MIPS CPU 波形





マルチサイクル MIPS CPU 波形



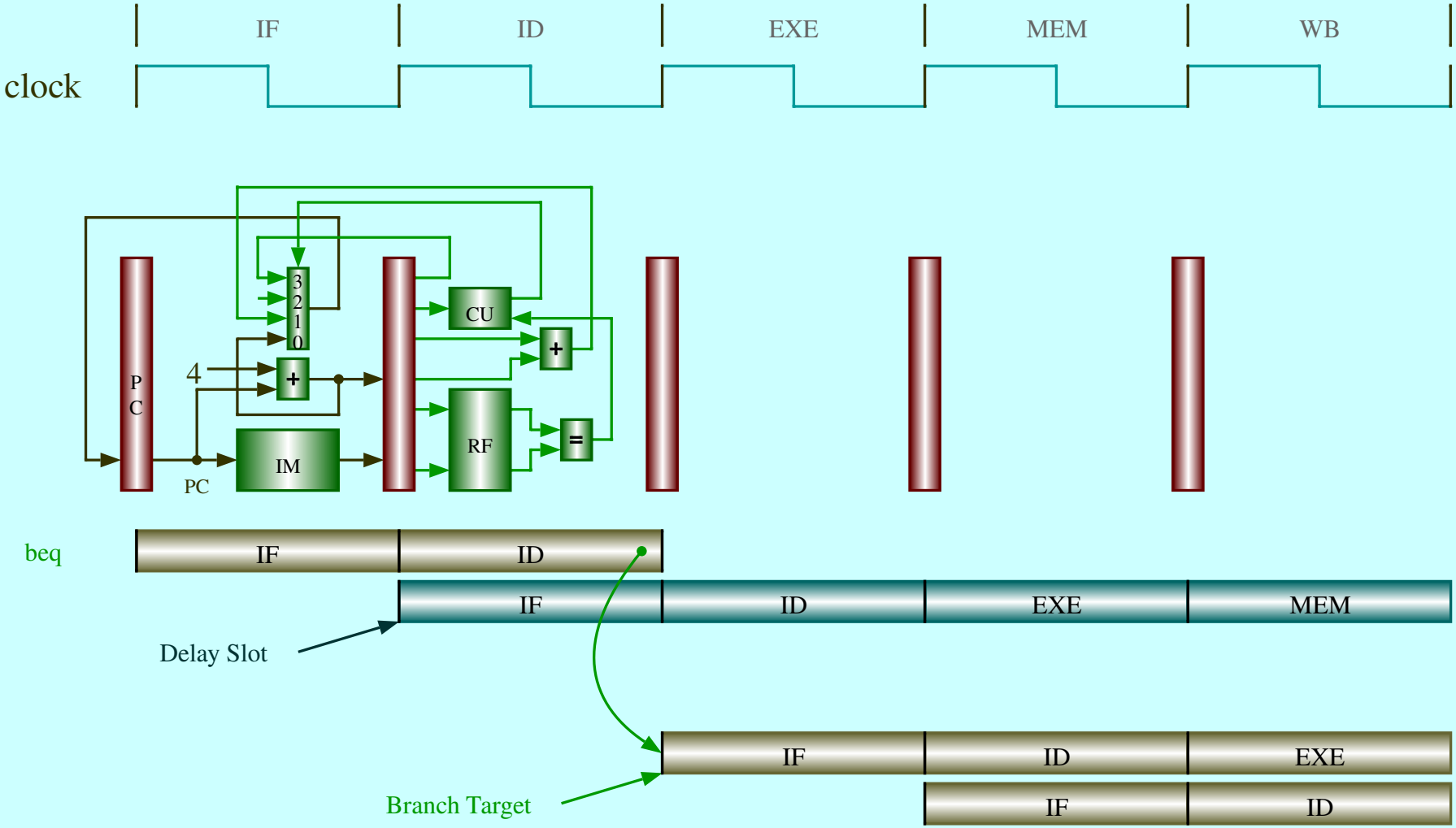


パイプライン CPU: 20 MIPS 命令の例

```
 1. add  rd, rs, rt   # reg[rd] = reg[rs]  +  reg[rt]
 2. sub  rd, rs, rt   # reg[rd] = reg[rs]  -  reg[rt]
 3. and  rd, rs, rt   # reg[rd] = reg[rs]  &  reg[rt]
 4. or   rd, rs, rt   # reg[rd] = reg[rs]  |  reg[rt]
 5. xor  rd, rs, rt   # reg[rd] = reg[rs]  ^  reg[rt]
 6. sll  rd, rt, sa   # reg[rd] = reg[rt]  << sa
 7. srl  rd, rt, sa   # reg[rd] = reg[rt]  >> sa (logical)
 8. sra  rd, rt, sa   # reg[rd] = reg[rt] >>> sa (arithmetic)
 9. jr   rs           #      PC = reg[rs]
10. addi rt, rs, imm  # reg[rt] = reg[rs]  +  (sign)imm
11. andi rt, rs, imm  # reg[rt] = reg[rs]  &  (zero)imm
12. ori  rt, rs, imm  # reg[rt] = reg[rs]  |  (zero)imm
13. xori rt, rs, imm  # reg[rt] = reg[rs]  ^  (zero)imm
14. lui  rt, imm      # reg[rt] = imm,0000000000000000
15. lw   rt, imm(rs)  # reg[rt] = memory[reg[rs]+(sign)imm]
16. sw   rt, imm(rs)  # memory[reg[rs]+(sign)imm] = reg[rt]
17. beq  rs, rt, imm  # if (reg[rs]==reg[rt]) PC=PC+4+(sign)imm*4
18. bne  rs, rt, imm  # if (reg[rs]!=reg[rt]) PC=PC+4+(sign)imm*4
19. j    addr         # PC = (PC+4)[31:28],addr,00
20. jal  addr         # PC = (PC+4)[31:28],addr,00; reg[31]=PC+8
```

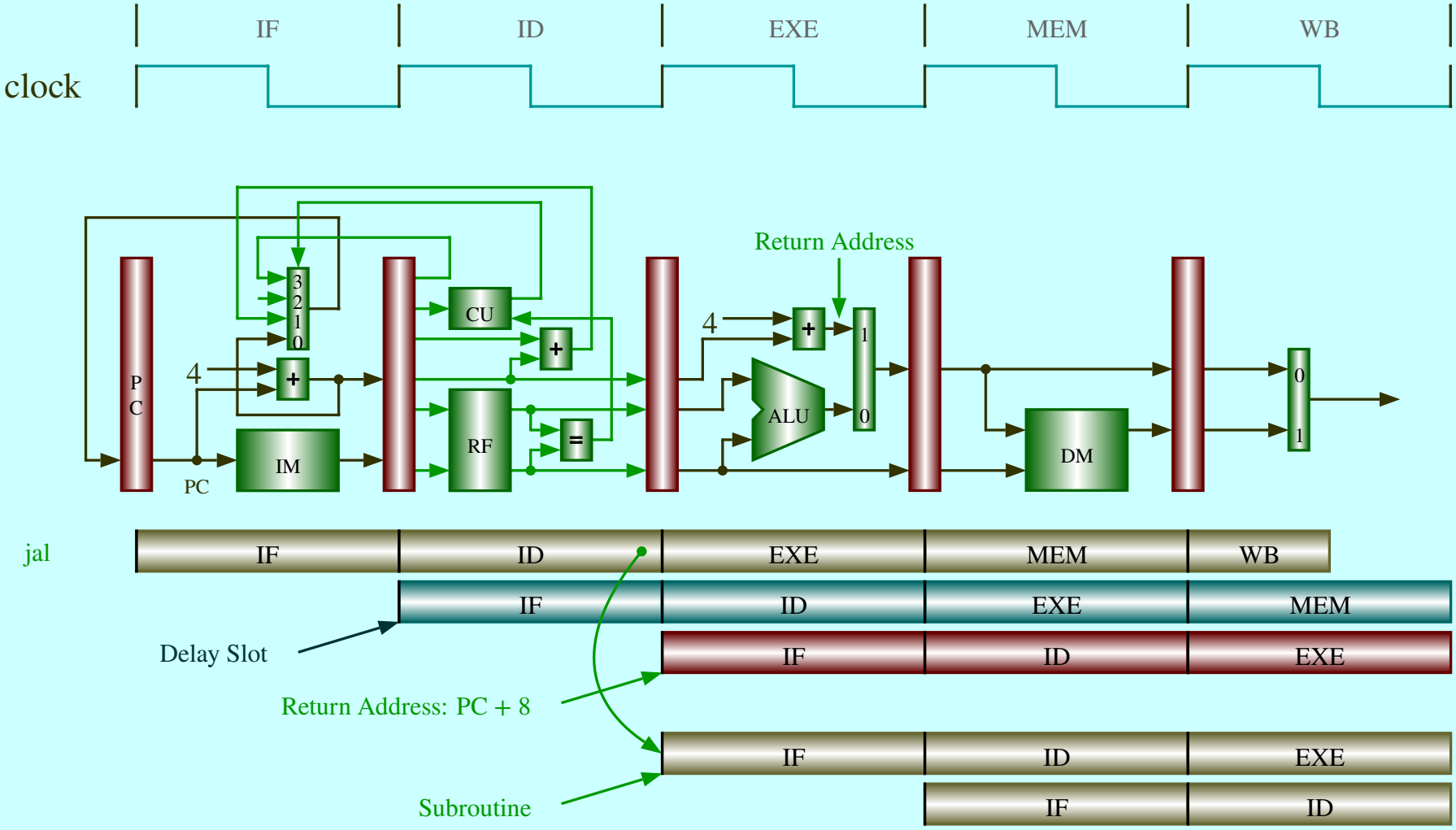
パイプライン CPU: 遅延分岐

遅延分岐方式では、分岐命令の次の命令(先読みした命令)を無条件に実行してから分岐をする

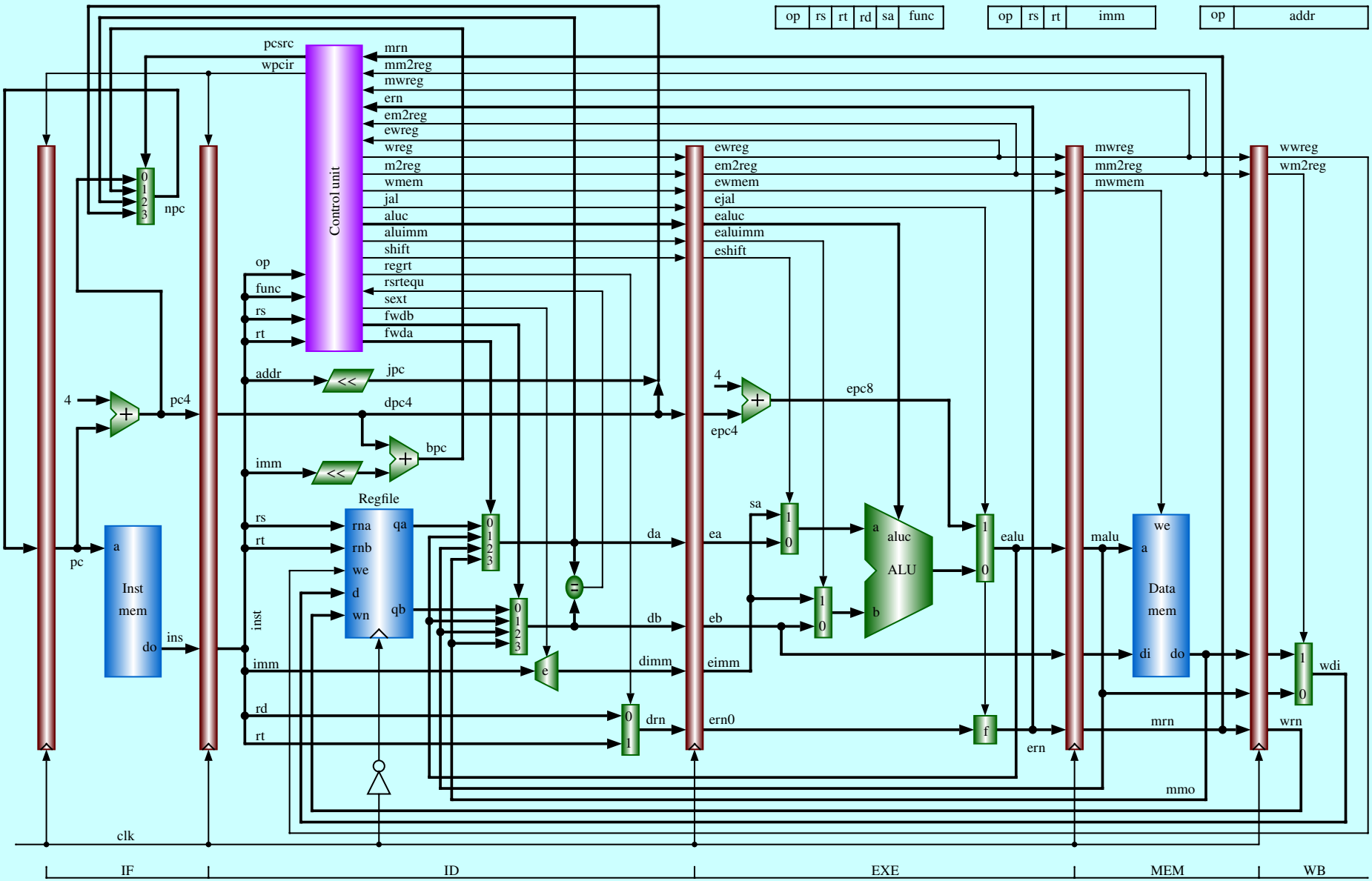


パイプライン CPU: jal 命令

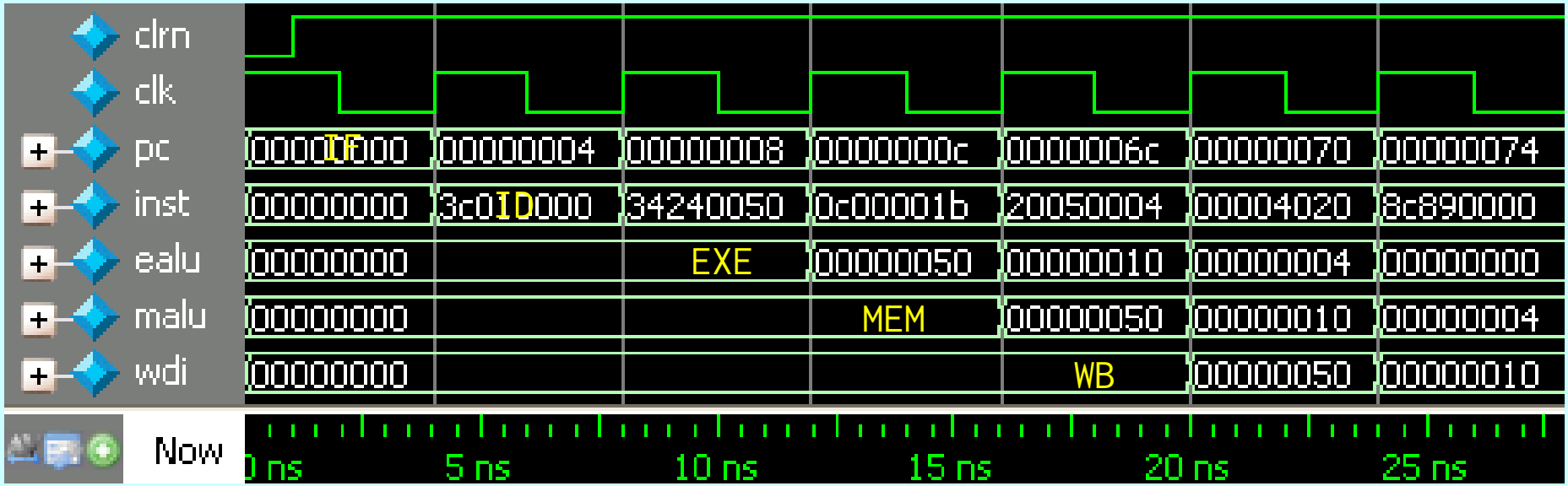
サブルーチンから戻るべき場所のアドレスをレジスタ31に保存しながらジャンプする



パイプライン CPU 回路

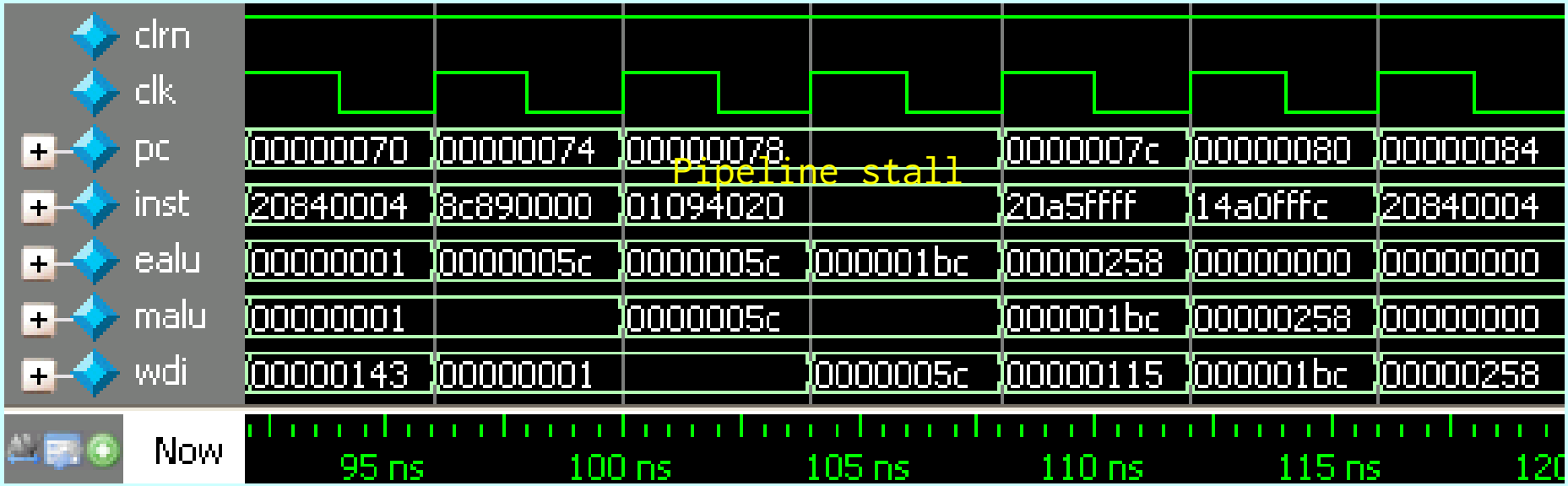


パイプライン CPU 波形 (Call)



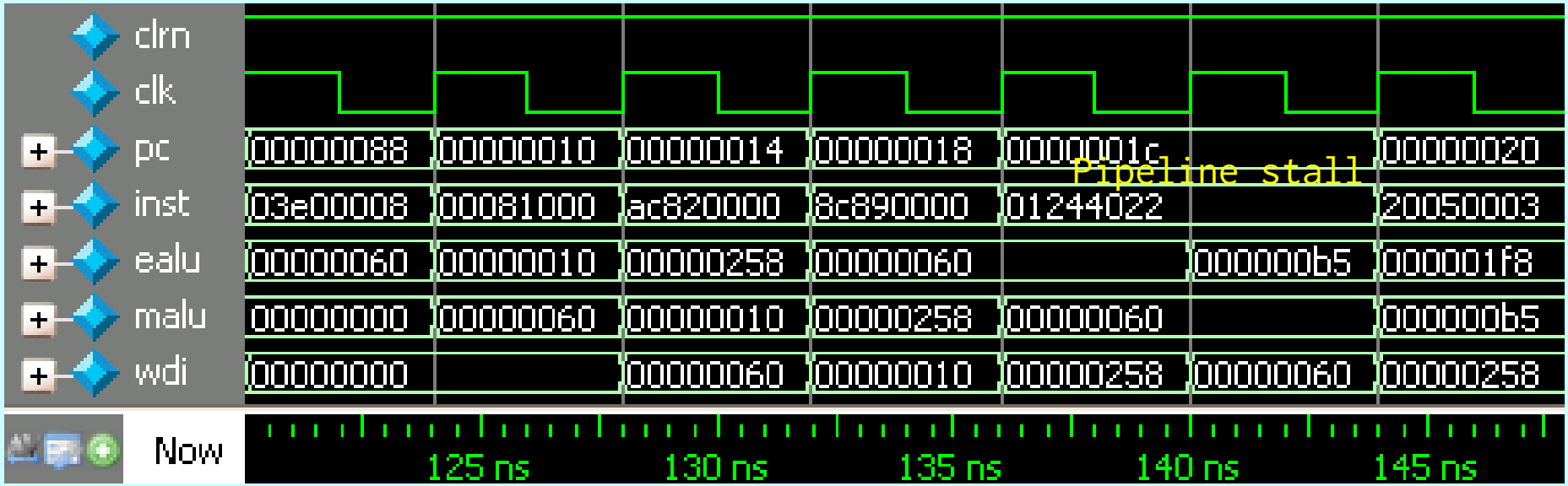

```
assign rom[6'h00] = 32'h3c010000;  // (00) main:   lui  $1, 0
assign rom[6'h01] = 32'h34240050;  // (04)         ori  $4, $1, 80
assign rom[6'h02] = 32'h0c00001b;  // (08) call:   jal  sum
assign rom[6'h03] = 32'h20050004;  // (0c) dslot1: addi $5, $0,  4
assign rom[6'h1b] = 32'h00004020;  // (6c) sum:    add  $8, $0, $0
assign rom[6'h1c] = 32'h8c890000;  // (70) loop:   lw   $9, 0($4)
assign rom[6'h1d] = 32'h01094020;  // (74) stall:  add  $8, $8, $9
```

パイプライン CPU 波形 (Pipeline Stall)




```
assign rom[6'h1c] = 32'h8c890000;  // (70) loop:   lw   $9, 0($4)
assign rom[6'h1d] = 32'h01094020;  // (74) stall:  add  $8, $8, $9
assign rom[6'h1e] = 32'h20a5ffff;  // (78)         addi $5, $5, -1
assign rom[6'h1f] = 32'h14a0fffc;  // (7c)         bne  $5, $0, loop
assign rom[6'h20] = 32'h20840004;  // (80) dslot5: addi $4, $4,  4
assign rom[6'h21] = 32'h03e00008;  // (84)         jr   $31
```

パイプライン CPU 波形 (Return)




```
assign rom[6'h22] = 32'h00081000;  // (88) dslot6: sll  $2, $8, 0
assign rom[6'h04] = 32'hac820000;  // (10) return: sw   $2, 0($4)
assign rom[6'h05] = 32'h8c890000;  // (14)         lw   $9, 0($4)
assign rom[6'h06] = 32'h01244022;  // (18)         sub  $8, $9, $4
assign rom[6'h07] = 32'h20050003;  // (1c)         addi $5, $0,  3
assign rom[6'h08] = 32'h20a5ffff;  // (20) loop2:  addi $5, $5, -1
```

課題 XIV オプション (+200 点)

1. オプション（+100 点）: MIPS マルチサイクルコンピュータを設計とシミュレーションしなさい。
2. オプション（+100 点）: MIPS パイプラインコンピュータを設計とシミュレーションしなさい。