Department of Computer Science
Course number: CSC.T341

コンピュータ論理設計
Computer Logic Design

6. 命令セットアーキテクチャ：命令形式とデータ表現
Instruction Set Architecture:
Instruction Format and Data Representation

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

Ver. 2023-04-26b

Sample Verilog HDL code

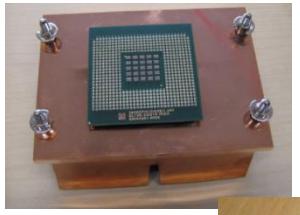
- ACRi Room のサーバーにリモートデスクトップでログインする.
  - /home/tu_kise/cld/lec6/ にサンプルのコードがあるので，Ubuntu のターミナルで次のコマンドを入力して，自分のディレクトリにコピーする.
  - /home/tu_kise は automount のディレクトリなので，アクセスしないとファイルが見えない．tabキーによる補完がうまく動作しないことがあるので注意する.

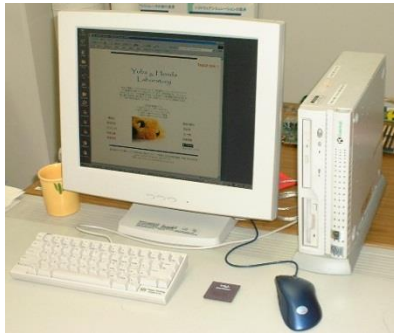
```
$ cd
$ mkdir cld
$ cd cld
$ cp /home/tu_kise/cld/lec6/* .
```

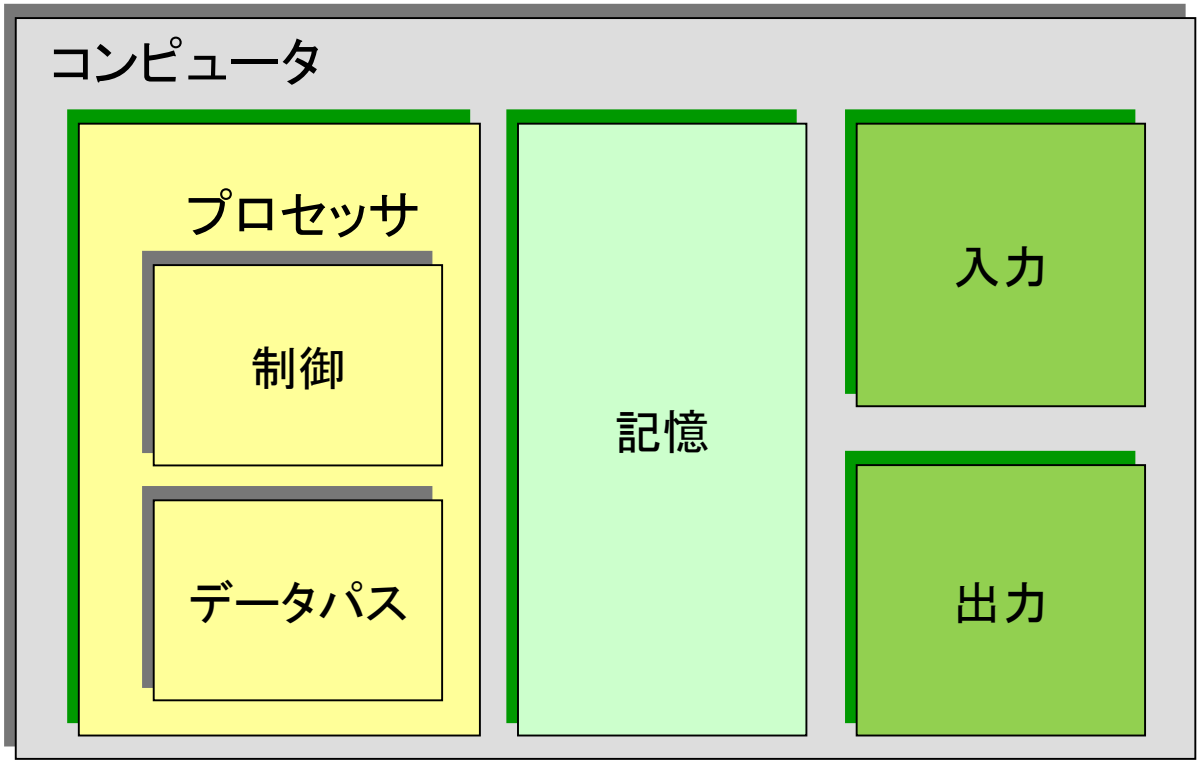
- code094.v をシミュレーションするためには.
  - リモートデスクトップで接続した Ubuntu のターミナルで次のコマンドを入力する.
  - コマンド iverilog でコンパイルして，生成される a.out を実行する.

```
$ iverilog code094.v
$ ./a.out
```

コンピュータ(ハードウェア)の古典的な要素







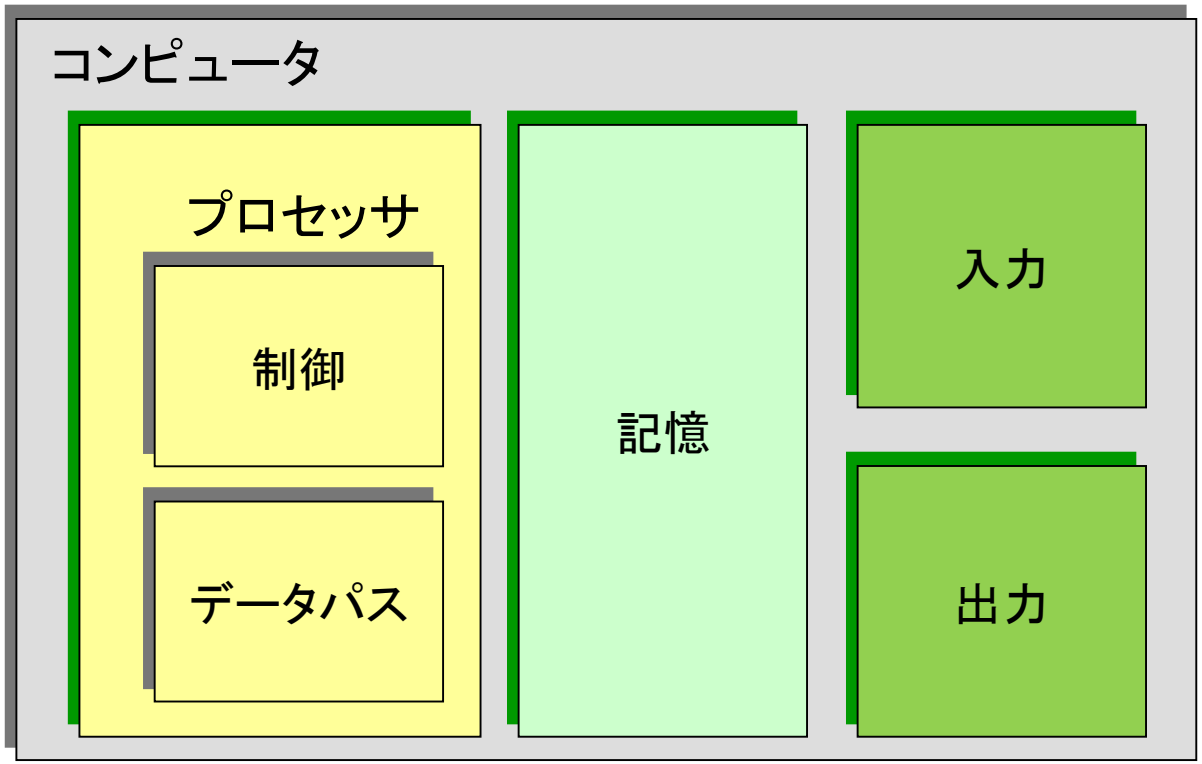
プロセッサは記憶装置から命令とデータを取り出す。入力装置はデータを記憶装置に書き込む。出力装置は記憶装置からデータを読みだす。制御装置は、データパス、記憶装置、入力装置、そして出力装置の動作を指定する信号を送る。

コンピュータの古典的な要素

コンパイラ

インタフェース　Instruction Set Architecture (ISA), 命令セットアーキテクチャ

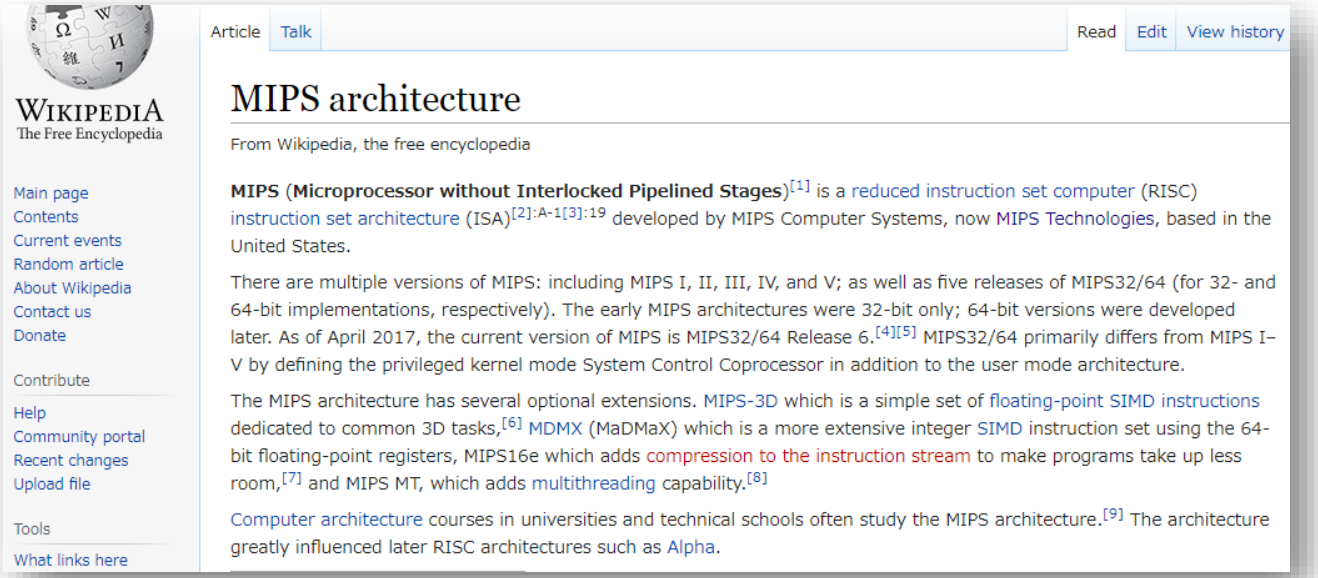
性能の評価



Two major ISA types: RISC vs CISC

- RISC (Reduced Instruction Set Computer) philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited addressing modes
  - limited operations
  - RISC: MIPS, Alpha, ARM, RISC-V, …
- CISC (Complex Instruction Set Computer) philosophy
  - ! fixed instruction lengths
  - ! load-store instruction sets
  - ! limited addressing modes
  - ! limited operations
  - CISC : DEC VAX11, Intel 80x86, …

MIPS, ARM, and RISC-V



WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute
Help
Community portal
Recent changes
Upload file

Tools
What links here

Article Talk Read Edit View history

MIPS architecture

From Wikipedia, the free encyclopedia

MIPS (**Microprocessor without Interlocked Pipelined Stages**)[1] is a reduced instruction set computer (RISC) instruction set architecture (ISA)[2]:A-1[3]:19 developed by MIPS Computer Systems, now MIPS Technologies, based in the United States.

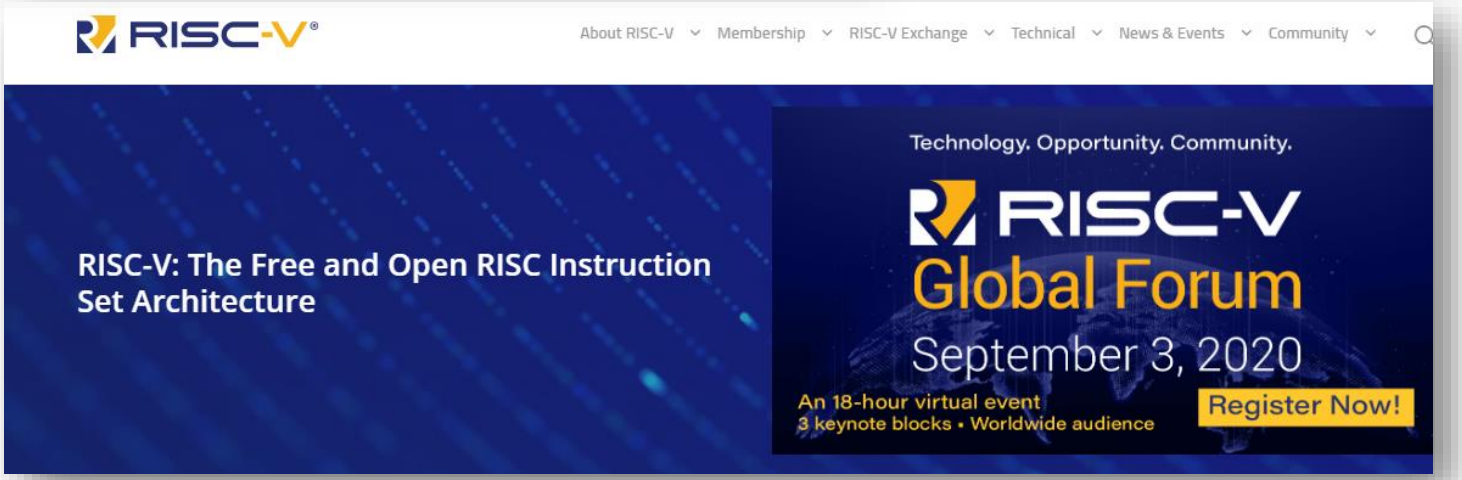
There are multiple versions of MIPS: including MIPS I, II, III, IV, and V; as well as five releases of MIPS32/64 (for 32- and 64-bit implementations, respectively). The early MIPS architectures were 32-bit only; 64-bit versions were developed later. As of April 2017, the current version of MIPS is MIPS32/64 Release 6.[4][5] MIPS32/64 primarily differs from MIPS I–V by defining the privileged kernel mode System Control Coprocessor in addition to the user mode architecture.

The MIPS architecture has several optional extensions. MIPS-3D which is a simple set of floating-point SIMD instructions dedicated to common 3D tasks,[6] MDMX (MaDMaX) which is a more extensive integer SIMD instruction set using the 64-bit floating-point registers, MIPS16e which adds compression to the instruction stream to make programs take up less room,[7] and MIPS MT, which adds multithreading capability.[8]

Computer architecture courses in universities and technical schools often study the MIPS architecture.[9] The architecture greatly influenced later RISC architectures such as Alpha.

ARM (Advanced RISC Machine)

https://en.wikipedia.org/wiki/MIPS_architecture



https://riscv.org/

C言語と RISC-V のアセンブリ言語で書いたプログラム例

main1.c

```
1  int main(){
2      int i=0, sum=1;
3      for(i=0; i<10; i++) sum = sum * i;
4      return sum;
5  }
```

main1.s

```
 1          .file   "main1.c"
 2          .option nopic
 3          .text
 4          .section        .text.startup,"ax",@progbits
 5          .align  2
 6          .globl  main
 7          .type   main, @function
 8  main:
 9          li      a0,1
10          li      a5,0
11          li      a4,10
12  .L2:
13          mul     a0,a0,a5
14          addi    a5,a5,1
15          bne     a5,a4,.L2
16          ret
17          .size   main, .-main
18          .ident  "GCC: (GNU) 11.1.0"
19          .section        .note.GNU-stack,"",@progbits
```

RISC-V の基本整数 ISA と機能拡張

ISA base and extensions (20191213)

Name	Description	Version	Status[a]
Base			
RVWMO	Weak Memory Ordering	2.0	Ratified
RV32I	Base Integer Instruction Set, 32-bit	2.1	Ratified
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers	1.9	Open
RV64I	Base Integer Instruction Set, 64-bit	2.1	Ratified
RV128I	Base Integer Instruction Set, 128-bit	1.7	Open
Extension			
M	Standard Extension for Integer Multiplication and Division	2.0	Ratified
A	Standard Extension for Atomic Instructions	2.1	Ratified
F	Standard Extension for Single-Precision Floating-Point	2.2	Ratified
D	Standard Extension for Double-Precision Floating-Point	2.2	Ratified
G	Shorthand for the base integer set (I) and above extensions (MAFD)	N/A	N/A
Q	Standard Extension for Quad-Precision Floating-Point	2.2	Ratified
L	Standard Extension for Decimal Floating-Point	0.0	Open
C	Standard Extension for Compressed Instructions	2.0	Ratified
B	Standard Extension for Bit Manipulation	0.92	Open
J	Standard Extension for Dynamically Translated Languages	0.0	Open
T	Standard Extension for Transactional Memory	0.0	Open
P	Standard Extension for Packed-SIMD Instructions	0.2	Open
V	Standard Extension for Vector Operations	0.9	Open
N	Standard Extension for User-Level Interrupts	1.1	Open
H	Standard Extension for Hypervisor	0.4	Open
ZiCSR	Control and Status Register (CSR)	2.0	Ratified
Zifencei	Instruction-Fetch Fence	2.0	Ratified
Zam	Misaligned Atomics	0.1	Open
Ztso	Total Store Ordering	0.1	Frozen

RISC-V の汎用レジスタ

XLEN = 32
for 32bit ISA

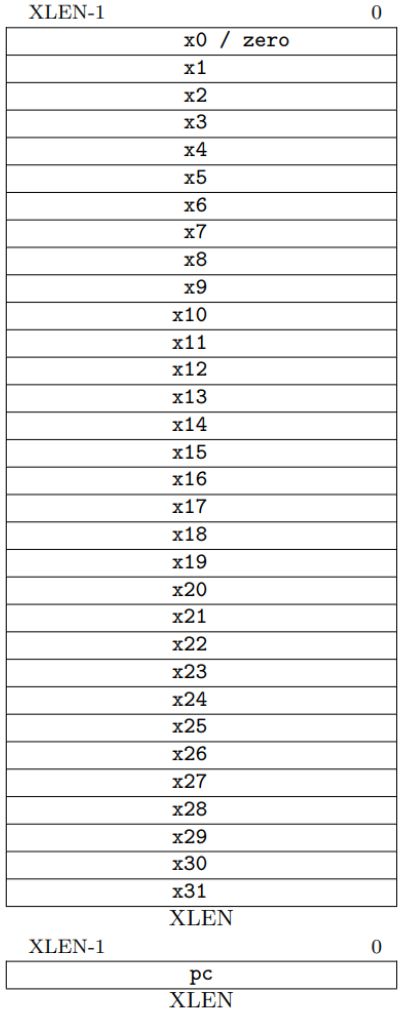


Figure 2.1: RISC-V base unprivileged integer register state.

ABI(Application Binary Interface) name

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Table 18.2: RISC-V calling convention register usage.

RISC-V の命令長

The RISC-V Instruction Set Manual
Volume I: Unprivileged ISA
Document Version 20191214-*draft*

Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu
November 12, 2021

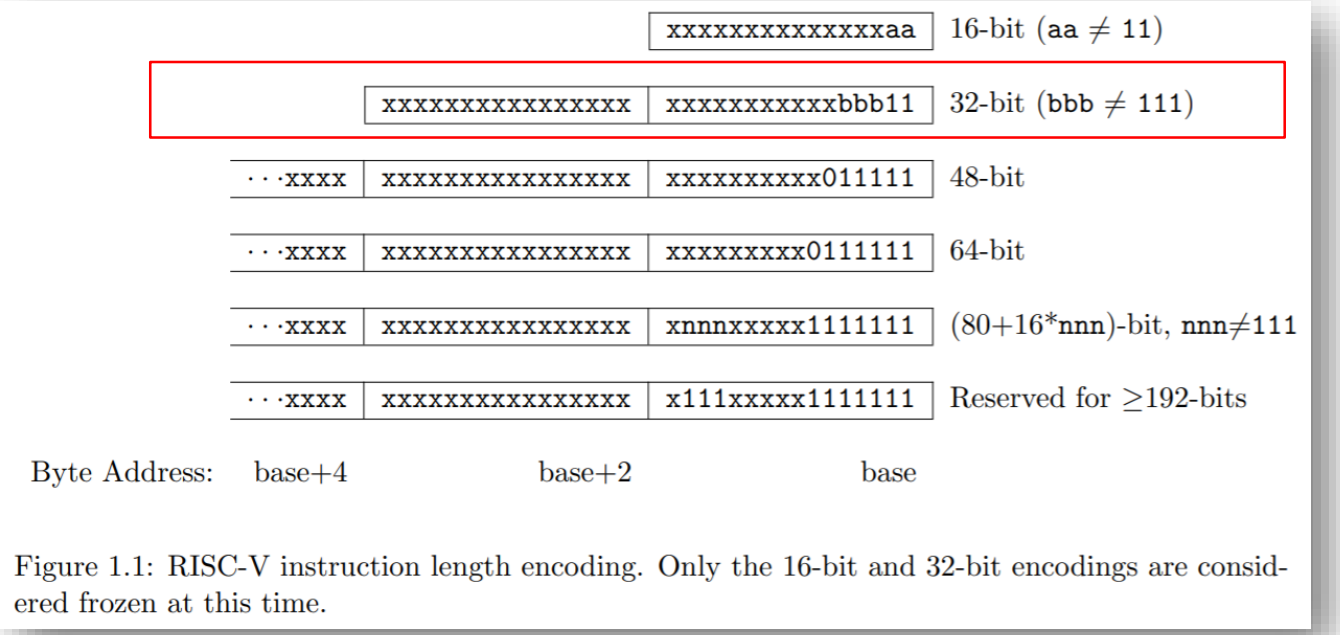


Figure 1.1: RISC-V instruction length encoding. Only the 16-bit and 32-bit encodings are considered frozen at this time.

RISC-V の命令フォーマット

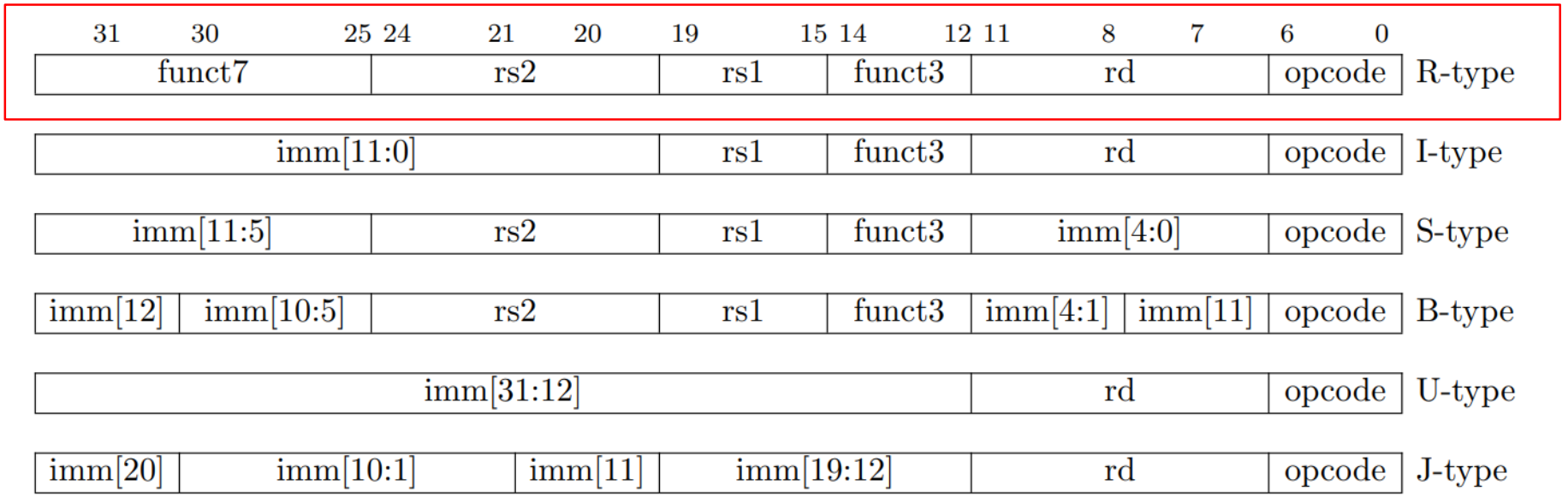


Figure 2.3: RISC-V base instruction formats showing immediate variants.

RISC-V Arithmetic Instructions

- RISC-V assembly language arithmetic statement

```
add   x7, x8, x9
sub   x7, x8, x9
```

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

```
destination <- source1 op source2
```

- Operand order is fixed (destination first)
- Those operands are all contained in the datapath's register file (`x0, ..., x31`)

Example（例題）

- `f = ( g + h ) - ( i + j )`

  f, g, h, i, j をそれぞれレジスタ x3, x4, x5, x6, x7 に割り付けるとする．上のステートメントをコンパイルした結果のRISC-Vのコードはどうなるか．

Answer

x3 x4 x5 x6 x7

- f = (g + h) − (i + j)

x10 x11

f, g, h, i, j をそれぞれレジスタ x3, x4, x5, x6, x7 に割り付けるとする. 上のステートメントをコンパイルした結果のRISC-Vのコードはどうなるか.

```
add x10, x4, x5   # x10 = (g + h)
add x11, x6, x7   # x11 = (i + j)
sub x3, x10, x11  # f   = x10 - x11
```

計算結果を保存する一時的なレジスタとして x10, x11を用いたが, 別のレジスタを用いても良い.

Machine Language - Add Instruction

- Instructions, like registers and words of data, are 32 bits long
- Arithmetic Instruction Format (R format):

```
add  x7, x8, x9
```

R format

opcode 7-bits *opcode* that specifies the operation

rs1 5-bits register file address of the first source operand

rs2 5-bits register file address of the second source operand

rd 5-bits register file address of the result's destination

funct3 and **funct7** 10-bits select the type of operation (function)

RISC-V の32ビット基本命令セット RV32I

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Table 24.1: RISC-V base opcode map, inst[1:0]=11

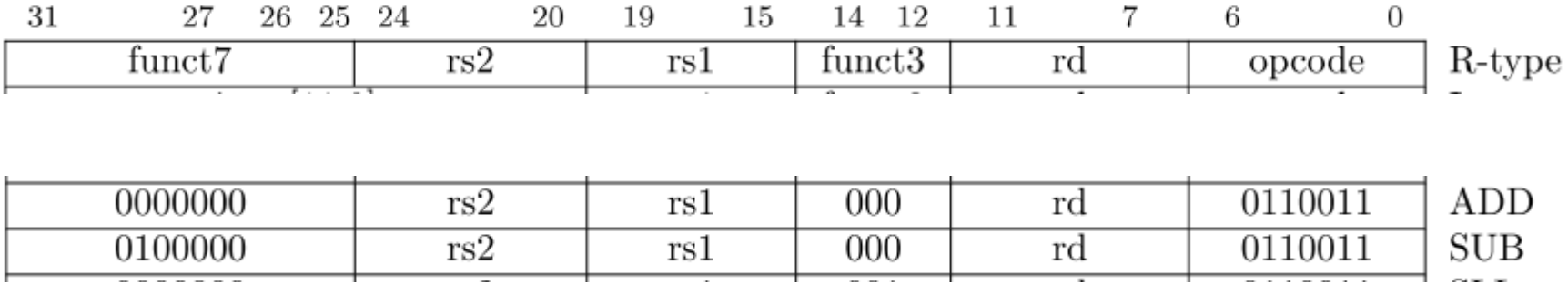
RV32I Base Instruction Set

imm[31:12]				rd	0110111 LUI
imm[31:12]				rd	0010111 AUIPC
imm[20|10:1|11|19:12]				rd	1101111 JAL
imm[11:0]		rs1	000	rd	1100111 JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011 BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011 BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011 BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011 BGE
				n[4:1|11]	1100011 BLTU
				n[4:1|11]	1100011 BGEU
				rd	0000011 LB
				rd	0000011 LH
				rd	0000011 LW
				rd	0000011 LBU
				rd	0000011 LHU
				m[4:0]	0100011 SB
				m[4:0]	0100011 SH
				m[4:0]	0100011 SW
				rd	0010011 ADDI
				rd	0010011 SLTI
imm[11:0]		rs1	011	rd	0010011 SLTIU
imm[11:0]		rs1	100	rd	0010011 XORI
imm[11:0]		rs1	110	rd	0010011 ORI
imm[11:0]		rs1	111	rd	0010011 ANDI
0000000	shamt	rs1	001	rd	0010011 SLLI
0000000	shamt	rs1	101	rd	0010011 SRLI
0100000	shamt	rs1	101	rd	0010011 SRAI
0000000	rs2	rs1	000	rd	0110011 ADD
0100000	rs2	rs1	000	rd	0110011 SUB
0000000	rs2	rs1	001	rd	0110011 SLL
0000000	rs2	rs1	010	rd	0110011 SLT
0000000	rs2	rs1	011	rd	0110011 SLTU
0000000	rs2	rs1	100	rd	0110011 XOR
0000000	rs2	rs1	101	rd	0110011 SRL
0100000	rs2	rs1	101	rd	0110011 SRA
0000000	rs2	rs1	110	rd	0110011 OR
0000000	rs2	rs1	111	rd	0110011 AND
fm | pred | succ		rs1	000	rd	0001111 FENCE
000000000000		00000	000	00000	1110011 ECALL
000000000001		00000	000	00000	1110011 EBREAK

Example（例題）

- 次のRISC-Vの命令列の機械語コードはどうなるか．それぞれの命令を2進数と16進数で表示する Verilog HDLのコードを記述して，その結果を示せ．

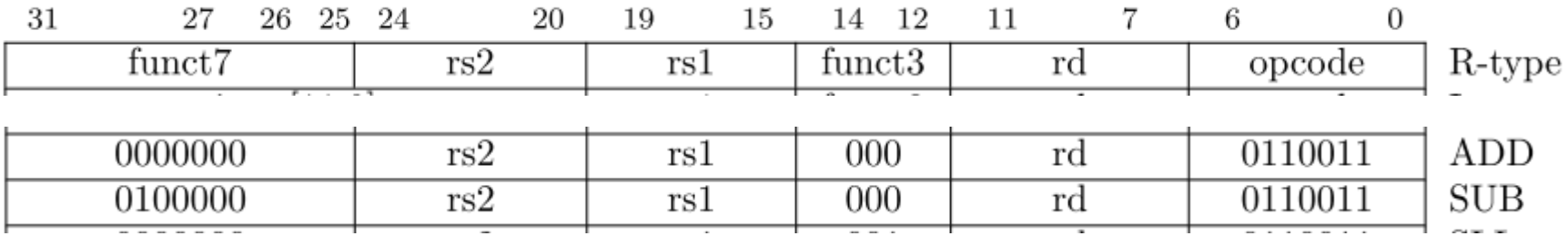
```
add x10, x4, x5     # x10 = (g + h)
add x11, x6, x7     # x11 = (i + j)
sub x3, x10, x11    # f   = x10 - x11
```



31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB

Answer

```
add x10, x4, x5      # x10 = (g + h)
add x11, x6, x7      # x11 = (i + j)
sub x3, x10, x11     # f   = x10 – x11
```



31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB

```
module m_top();
  reg [31:0] r_i1, r_i2, r_i3;
  initial begin
    r_i1 <= {7'b0000000, 5'd5,  5'd4,  3'b000, 5'd10, 7'b0110011};
    r_i2 <= {7'b0000000, 5'd7,  5'd6,  3'b000, 5'd11, 7'b0110011};
    r_i3 <= {7'b0100000, 5'd11, 5'd10, 3'b000, 5'd3,  7'b0110011};
  end
  initial #1 begin
    $display("i1: %b %x", r_i1, r_i1);
    $display("i2: %b %x", r_i2, r_i2);
    $display("i3: %b %x", r_i3, r_i3);
  end
endmodule
```

```
i1: 00000000010100100000010100110011 00520533
i2: 00000000011100110000010110110011 007305b3
i3: 01000000101101010000000110110011 40b501b3
```

code094.v

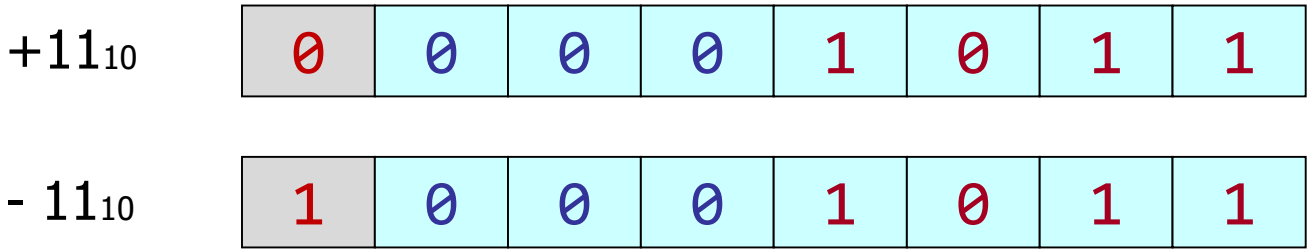
Integer（整数）Representation

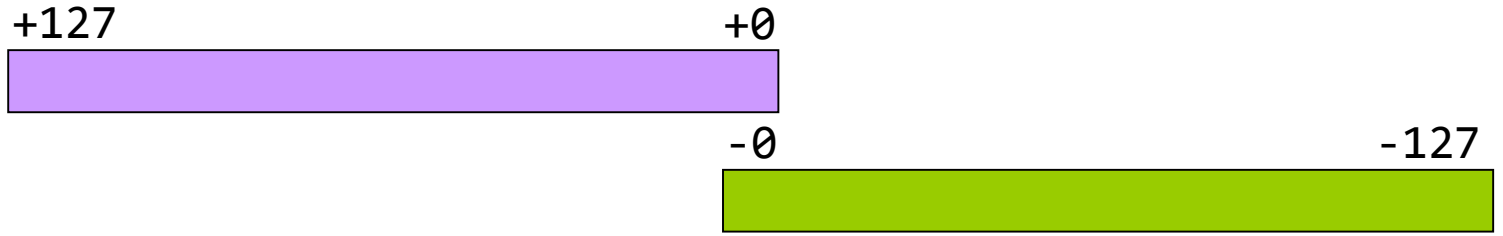
- まずは，「符号なし数」の表現を考える．
  - 欠点は負の数を表現できないこと．
- 整数を2進数の符号なし数で表現するには
  - 例えば，3_{10} であれば，11_2 として下位ビットを決める（右下の小さい数が10の場合には10進数、2の場合には2進数を示す）．
  - 上位の残ったビットを0で埋める．
  - 8ビットであれば，0～255 の256種類の整数を表現できる．

0	0	0	0	0	0	1	1

Integer（整数）Representation

- 符号つき絶対値 (sign and magnitude) による符号付き数の表現
  - 11_{10} であれば，1011_2 として下位ビットを決める．
  - ただし，最上位ビットを用いて符号を表す（符号ビット）．符号ビットが0であれば正数，1であれば負数とする．
  - 残ったビットを0で埋める．
  - 8ビットであれば，- 127 ～127 までの 255種類の整数を表現できる





Integer（整数）Representation

- 2の補数 (two's complement) による符号付き数の表現
  - 正数のビットを反転させ，1を加えたものを負数とする．
  - 8ビットであれば，- 128 ～127 までの 256種類の整数を表現できる

$0000\ 0000_2 = 0_{10}$		
$0000\ 0001_2 = +1_{10}$	$1111\ 1110_2 = -1_{10}$	$1111\ 1111_2 = -1_{10}$
$0000\ 0010_2 = +2_{10}$	$1111\ 1101_2 = -2_{10}$	$1111\ 1110_2 = -2_{10}$
…	…	…
$0111\ 1101_2 = +125_{10}$	$1000\ 0010_2 = -125_{10}$	$1000\ 0011_2 = -125_{10}$
$0111\ 1110_2 = +126_{10}$	$1000\ 0001_2 = -126_{10}$	$1000\ 0010_2 = -126_{10}$
$0111\ 1111_2 = +127_{10}$	$1000\ 0000_2 = -127_{10}$	$1000\ 0001_2 = -127_{10}$
		$1000\ 0000_2 = -128_{10}$
0と1～127の正数	1～127の正数のビット反転（1の補数表現）	ビット反転に1を加えて得る負数

Integer（整数）Representation

- 2の補数 (two's complement) による符号付き数の表現
  - 正数のビットを反転させ，1を加えたものを負数とする．
  - 8ビットであれば，- 128 ～127 までの 256種類の整数を表現できる

$0000\ 0000_2 = 0_{10}$

$0000\ 0001_2 = +1_{10}$

$0000\ 0010_2 = +2_{10}$

…

$0111\ 1101_2 = +125_{10}$

$0111\ 1110_2 = +126_{10}$

$0111\ 1111_2 = +127_{10}$

0と1～127の正数

$1111\ 1111_2 = -1_{10}$

$1111\ 1110_2 = -2_{10}$

…

$1000\ 0011_2 = -125_{10}$

$1000\ 0010_2 = -126_{10}$

$1000\ 0001_2 = -127_{10}$

$1000\ 0000_2 = -128_{10}$

ビット反転に1を加えて得る負数

Integer（整数）Representation

- 2の補数 (two's complement) 表現の特徴
  - 全てのビットを反転させて1を加えると，正負が反転する.
  - 最上位ビットは符号ビットと呼ばれ，0であれば正数，1であれば負数.
  - 符号拡張(sign extension)と呼ばれるビット長を増やす処理は，符号ビットを複製して補填すればよい.

$$x + \overline{x} = -1$$

$$\overline{x} + 1 = -x$$

Example（例題）

- 16ビットの2進数の 2_{10} と -2_{10} を32ビットの2進数に変換せよ．これらは2の補数で表現されている．

Answer

- 16ビットの2進数の 2_{10} と -2_{10} を32ビットの2進数に変換せよ. これらは2の補数で表現されている.

16ビットの2進数の 2_{10} 0000 0000 0000 0010

16ビットの2進数の -2_{10} 1111 1111 1111 1110

32ビットの2進数の 2_{10} 0000 0000 0000 0000 0000 0000 0000 0010

32ビットの2進数の -2_{10} 1111 1111 1111 1111 1111 1111 1111 1110

Verilog HDLで「2の補数」として表示

- ワイヤ型の信号の定義を wire signed とすることで，2の補数表現の符号付き整数として扱われる．

code095.v

```
module m_top();
  reg [15:0] r_data = 16'b1111111111111110;
  wire signed [15:0] w_data = r_data;

  initial #1 begin
    $display("%6d", r_data);
    $display("%6d", w_data);
  end
endmodule
```

```
 65534
    -2
```

Verilog HDLで2の補数表現の符号拡張

- 符号拡張(sign extension)と呼ばれるビット長を増やす処理は，符号ビットを複製して補填すればよい．
- 2の補数表現の16ビットの整数を32ビットの整数に符号拡張する例
  `w_data2 = {{16{w_data1[15]}}, w_data1};`

code096.v

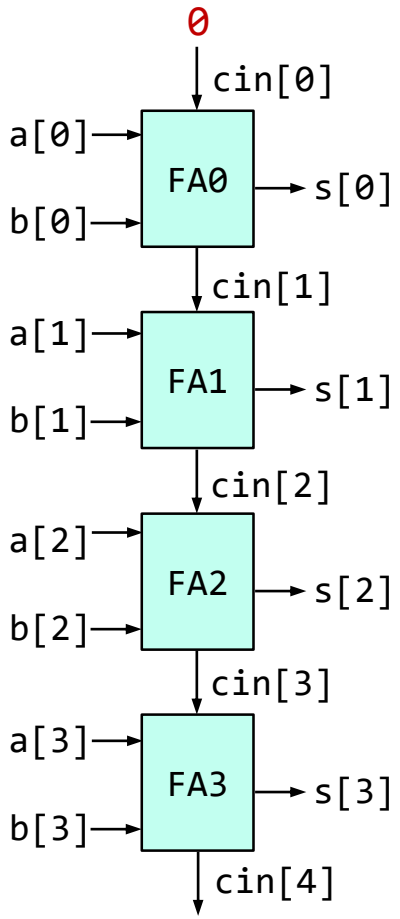
```
module m_top();
  wire signed [15:0] w_data1 = 16'b1111111111111110;
  wire signed [31:0] w_data2 = {{16{w_data1[15]}}, w_data1};

  initial #1 begin
    $display(“%5d %32b", w_data1, w_data1);
    $display("%5d %32b", w_data2, w_data2);
  end
endmodule
```

```
   -2                  1111111111111110
   -2 11111111111111111111111111111110
```

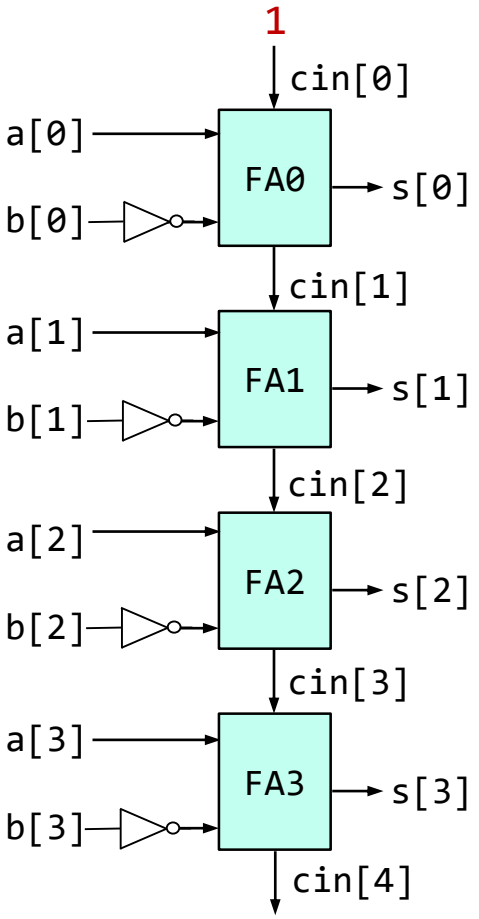
Adder and Subtractor

- 加算器を用いて，減算することでハードウエア量を節約できる
- 4-bit Ripple Carry Adder

```
s = a + b
```

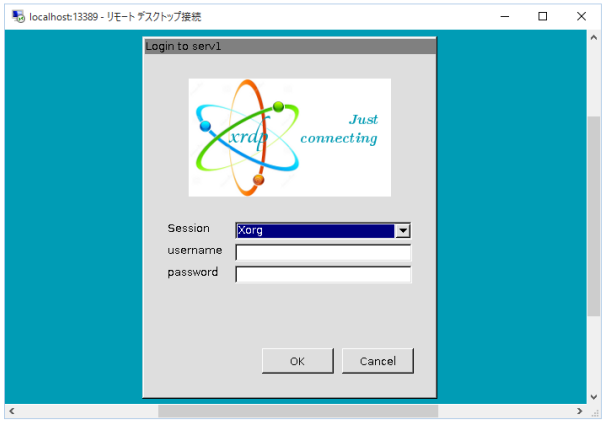


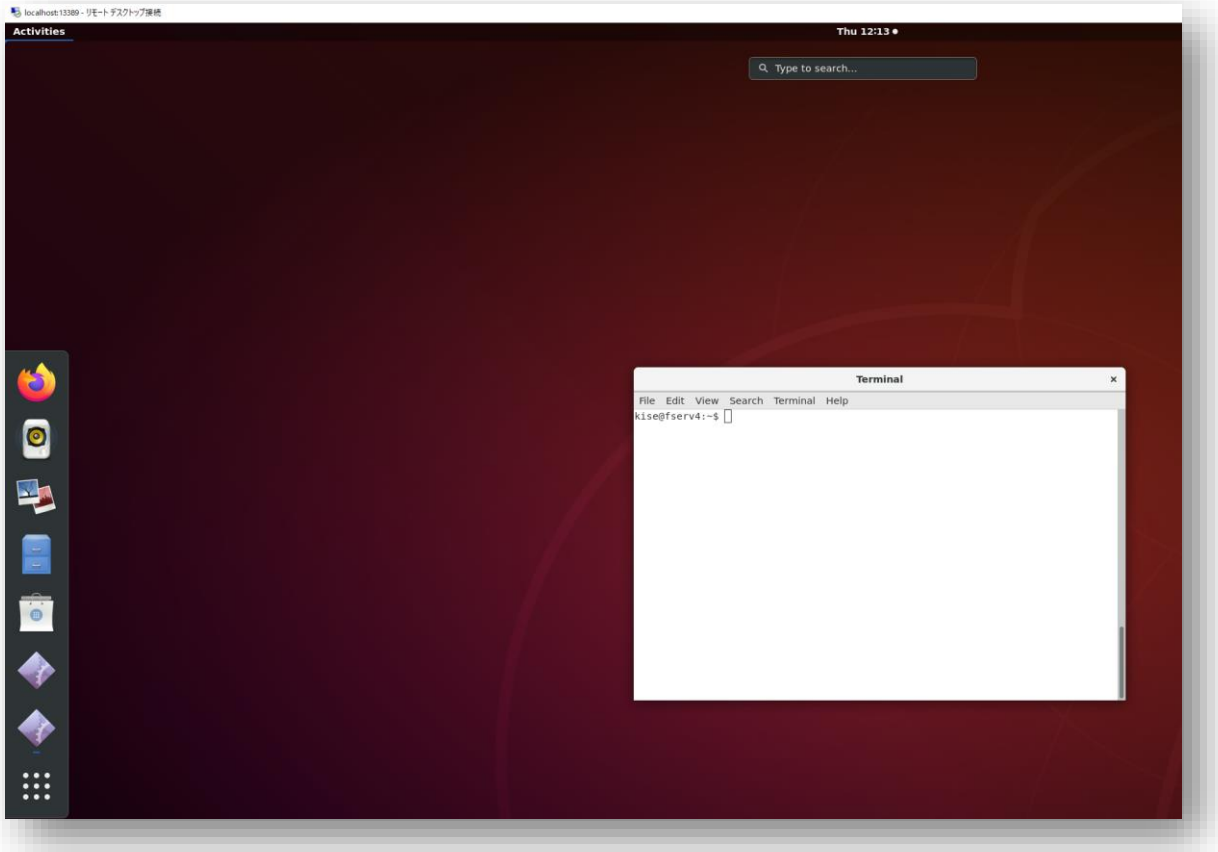

```
s = a - b
  = a + (-b)
  = a + (~b + 1)
```



ACRiルームのデモンストレーション

- LUT で実現される真理値表を確認する方法.
- Clocking Wizard を使って 20MHz のクロック信号を生成する方法.





References

- Computer Logic Design support page
  - https://www.arch.cs.titech.ac.jp/lecture/CLD/
- ACRi Room
  - https://gw.acri.c.titech.ac.jp
- ACRi Blog
  - https://www.acri.c.titech.ac.jp/wordpress/
- 情報工学系計算機室
  - http://www.csc.titech.ac.jp/
- Xilinx Vivado Design Suite
  - https://japan.xilinx.com/products/design-tools/vivado.html
- Digilent Arty A7-35T
  - https://reference.digilentinc.com/reference/programmable-logic/arty-a7/start
- Verilog HDL
  - https://ja.wikipedia.org/wiki/Verilog