

情報通信実験3

RISC-V プログラミング編

一色 剛
工学院情報通信系
isshiki@ict.e.titech.ac.jp

資料概要

1. RISC-Vプロセッサとシステム概要
2. RV32-I命令セット仕様
3. ABI : エンディアン、アラインメント、関数呼出し規則
4. 入出力装置と割込み処理機構
5. RV32プログラミング手順 : WSL(Ubuntu)
6. FPGA使用手順 : Windows-11
7. RV32プログラミング課題

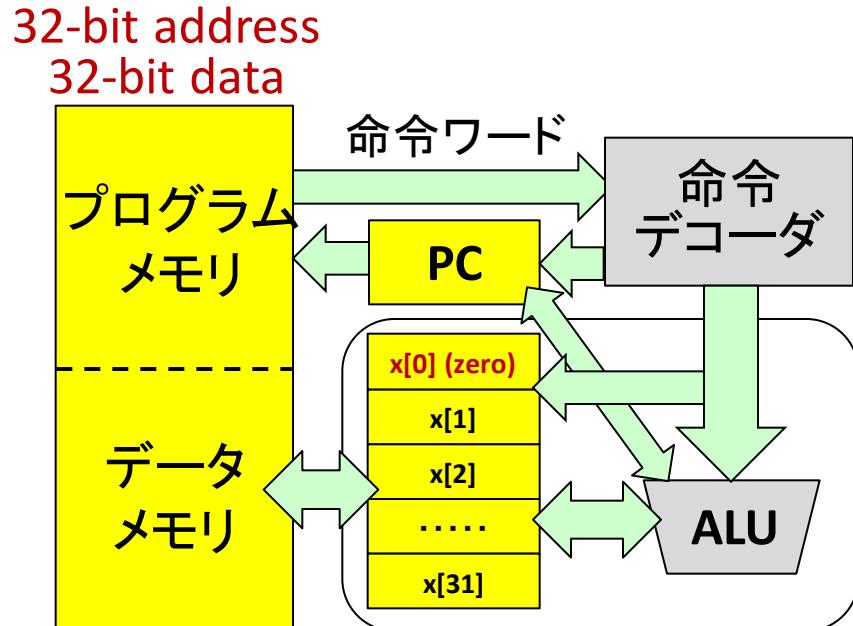
RISC-Vプロセッサ概要

<https://riscv.org/japan/>

- **技術背景**: 2010年頃UC Berkeleyで開発がスタート
- **現在**: 70カ国、3950加盟団体(企業、大学)、ライセンスフリー
オープンソース
- **命令セット**: RV32/64 (データ幅)、演算機能でモジュール化された命令セット仕様
 - 'I': 整数演算
 - 'M': 乗算・除算・剰余算 (mul, div, rem)
 - 'A': atomicメモリアクセス (read-modify-write)
 - 'F': 単精度浮動小数点演算
 - 'D': 倍精度浮動小数点演算
 - その他: 'C' (16-bit命令長), 'B' (ビット操作演算), 'V' (ベクトル演算)
- **命令長**: 32ビット ('C' は16ビット命令長)
- **データ長**: 32ビット (RV32)、64ビット (RV64)

RV32-I (32ビット整数演算) プロセッサ構成

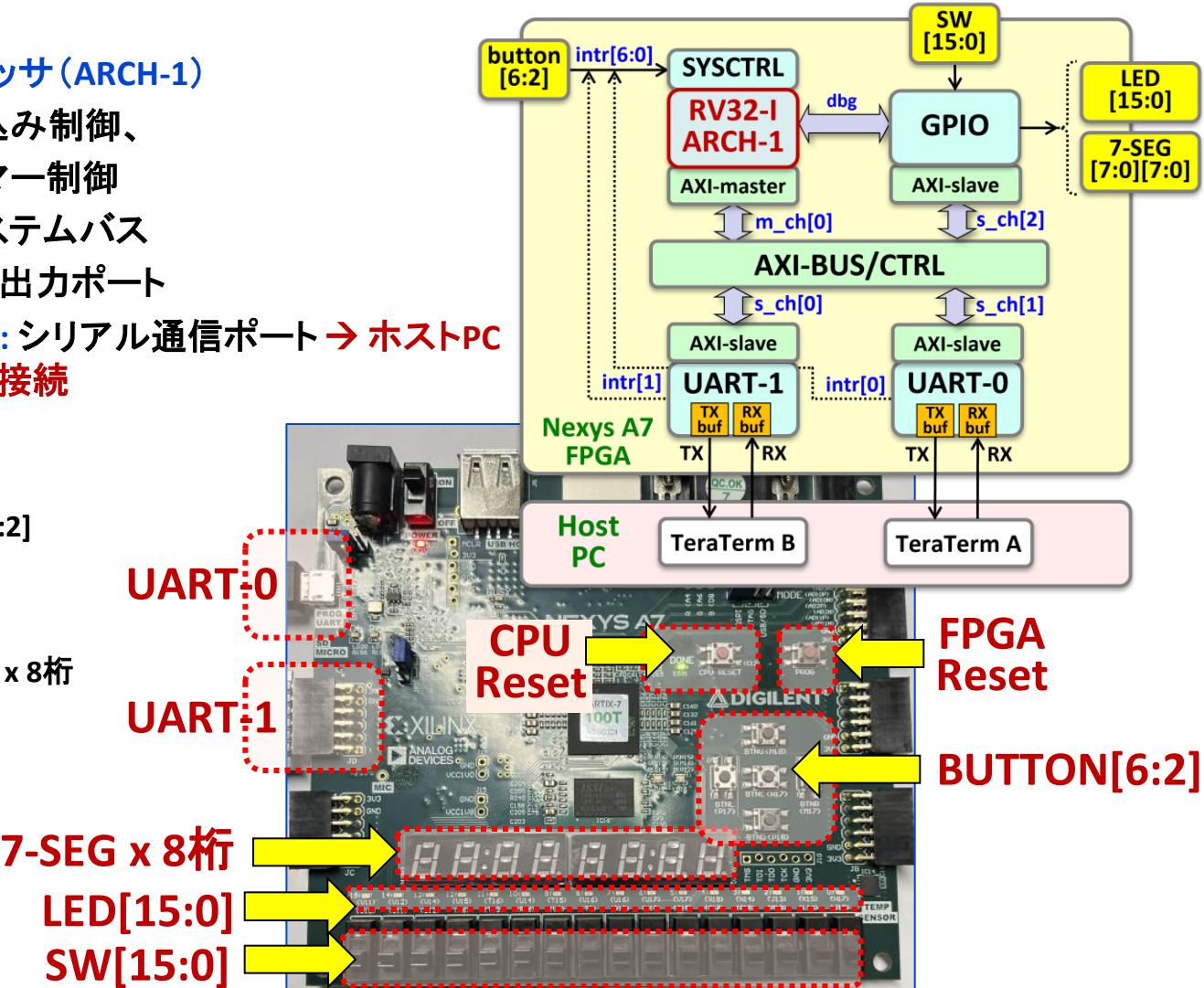
- プロセッサ構成要素 : RV32の場合データ幅は32ビット
 - メモリ : プログラムとデータを格納(32ビットアドレス、32ビットデータ)
 - PC : プログラムカウンタ(実行する命令アドレスを格納)
 - レジスタ(32個) : $x[0] \sim x[31]$ (ただし、 $x[0] = 0$ に固定)



レジスタ	レジスタ別名	説明 ABI mnemonic(後述)
$x[0]$	zero	0に固定
$x[1]$	ra	return address
$x[2]$	sp	stack pointer
$x[3]$	gp	global pointer
$x[4]$	tp	thread pointer
$x[5] \sim x[7]$	t0 \sim t2	temporary registers
$x[8] \sim x[9]$	s0 \sim s1	callee-saved registers
$x[10] \sim x[17]$	a0 \sim a7	argument registers
$x[18] \sim x[27]$	s2 \sim s11	callee-saved registers
$x[28] \sim x[31]$	t3 \sim t6	temporary registers

プロセッサシステム構成 (ex3_rv32)

- RV32-I プロセッサ (ARCH-1)
- SYSCTRL : 割込み制御、
タイマー制御
- AXI4 BUS : システムバス
- GPIO : 汎用入出力ポート
- UART x 2系統 : シリアル通信ポート → ホストPC
のTeraTermと接続
- 入力装置:
 - SW[15:0]
 - BUTTON[6:2]
- 出力装置:
 - LED[15:0]
 - 7-SEG[7:0] x 8桁



資料概要

1. RISC-Vプロセッサとシステム概要
2. RV32-I命令セット仕様
3. ABI : エンディアン、アラインメント、関数呼出し規則
4. 入出力装置と割込み処理機構
5. RV32プログラミング手順 : WSL(Ubuntu)
6. FPGA使用手順 : Windows-11
7. RV32プログラミング課題

高級プログラミング言語と アセンブリ言語・機械語の違い

```
int d[6] = { 2, 3, 5, 7, 11, 13 };
int main() {
    int sum = 0, i;
    for (i = 0; i < 6; i++) {
        sum += d[i];
    }
    return sum;
}
```

コンパイラ

```
00001010 <main>:
    1010: 007ff797    auipc   a5 , 0x7ff
    1014: ff078793    addi    a5 , a5 , -16 # 800000 <d>
    1018: 007ff697    auipc   a3 , 0x7ff
    101c: 00068693    mv      a3 , a3
    1020: 00000513    li      a0 , 0
    1024: 0007a703    lw      a4 , 0 (a5)
    1028: 00478793    addi   a5 , a5 , 4
    102c: 00e50533    add    a0 , a0 , a4
    1030: fed79ae3    bne    a5 , a3 , 1024
    1034: 00008067    ret
```

```
00800000 <d>:
    800000: 00000002
    800004: 00000003
    800008: 00000005
    80000c: 00000007
    800010: 0000000b
    800014: 0000000d
```

C言語プログラム

- 命令セットに非依存 → 異種CPU/命令セットで共通化可能
- 記述の抽象化により、より複雑な処理が簡素に表現できる
- 多様なデータ型のプログラミング変数が幾らでも使える

アドレス 機械語
(32ビット)

アセンブリ言語・機械語プログラム

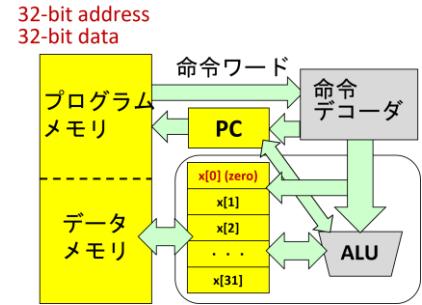
- 計算機ハードウェアが直接解釈する機械語と等価
- 「プログラミング変数」は、レジスタとメモリ番地のみ
- 実行できる演算種類も命令セットで限られる

16進表記: 4ビットを{ 0, 1, ... 9, a (10), b (11), c (12), d (13), e (14), f (15) }で表し、先頭に「0x」をつける

(例) 0x7ff → 7 * 16² + 15 * 16¹ + 15 * 16⁰ = 2047

- アセンブリプログラムでは、アドレス値と機械語はデフォルトで16進表記(先頭の「0x」は省略)

RV32-I命令セット仕様



- **オペランド: 演算対象データ、格納データ**
 - レジスタオペランド: 5ビットで $x[k]$ ($k = 0, 1, \dots, 31$)を指定 ($x[0] = 0$ に固定)
 - 格納レジスタが $x[0]$ の場合は、演算結果は格納されない
 - メモリオペランド: レジスタへのロード命令、ストア命令のみ
 - 即値オペランド: 命令ワードに埋め込まれた定数
- **即値(定数)オペランド: 様々なビット長と符号なし・付きの組合せ**
 - $\text{simm}[11:0], \text{simm}[12:1], \text{simm}[20:1]$: 符号付き即値(32ビットへ符号拡張)
 - $\text{uimm}[4:0], \text{uimm}[31:12]$: 符号なし即値(32ビットへ0拡張)
- **整数データ型: 32-bit (Word), 16-bit (Half), 8-bit (Byte)**
 - 符号なし(unsigned)、符号付き(signed)
- **演算命令: レジスタ格納、レジスタ・即値オペランド**
 - 算術演算(加減算)、シフト演算(右シフト、左シフト)、論理演算(AND/OR/EXOR)、比較演算
- **プログラム制御命令: サブルーチン呼び出し、条件分岐**
- **メモリアクセス命令:**
 - ロード(読み出し): **Byte** (signed/unsigned), **Half** (signed/unsigned), **Word**
 - ストア(書き込み): **Byte**, **Half**, **Word** (signed/unsignedの区別はない)
- **その他の命令(本実験では未使用):**
 - FENCE命令: 前後のメモリアクセス命令の実行順序制御
 - ECALL, EBREAK命令: システム制御

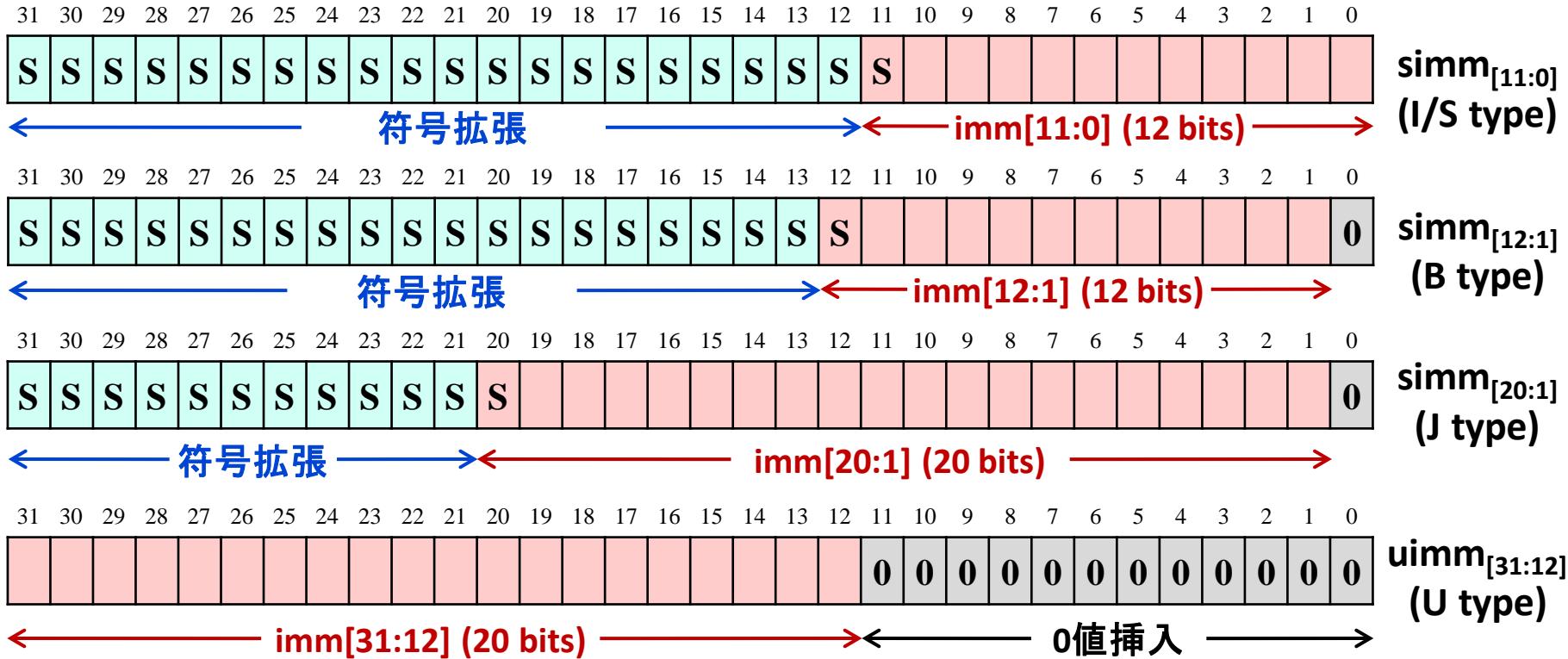
RV32-I命令形式(フォーマット)

命令
種別

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																																																																																								
funct7					rs2					rs1					funct3					rd					opcode																																																																																														
11					10					9					8					7					6					5																																																																																									
imm[11:0]					rs1					funct3					rd					opcode					R-type																																																																																														
11					10					9					8					7					6					5																																																																																									
imm[11:0]					rs1					funct3					4					3					2					1					0					opcode																																																																															
S-type					imm[11:0]					rs2					rs1					funct3					4					3					2					1					0					opcode																																																																					
11					10					9					8					7					6					5					rs2					rs1					funct3					imm[4:0]					opcode																																																																
B-type					imm[11:0]					12					10					9					8					7					6					5					rs2					rs1					funct3					4					3					2					1					11					opcode																																		
12					10					9					8					7					6					5					rs2					rs1					funct3					imm[4:1], imm[11]					opcode																																																																
U-type					imm[12:1]					31					30					29					28					27					26					25					24					23					22					21					20					19					18					17					16					15					14					13					12					rd					opcode				
J-type					imm[31:12]					20					10					9					8					7					6					5					4					3					2					1					11					19					18					17					16					15					14					13					12					rd					opcode				
imm[20:1], imm[10:1], imm[11], imm[19:12]					imm[20]					imm[10:1]					imm[11]					imm[19:12]					uimm[31:12]					J-type					simmm[20:1]					opcode																																																																															

- rs1, rs2, rd : レジスタID (5ビット)
- opcode, funct3, funct7 : 命令判別コード
- imm[n:m] : 即値データ(n:最上位ビット、m:最下位ビット)

即値データの32ビット拡張



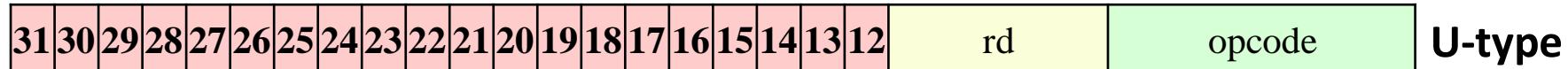
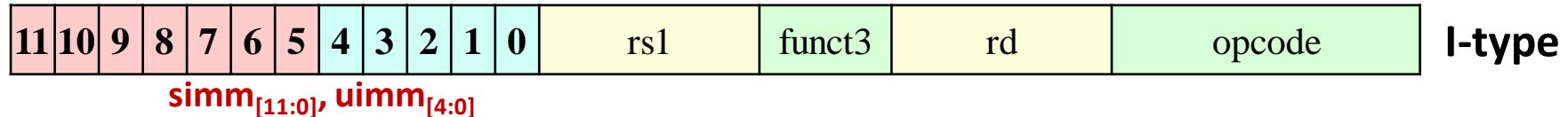
RV32-I命令セット：演算命令(レジスタオペランド)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
funct7				rs2		rs1		funct3		rd		opcode						R-type													

種別	アセンブリ命令	レジスタ転送式	funct7	funct3	opcode
R-type	ADD rd, rs1, rs2	$rd \leftarrow rs1 + rs2$	0000000	000	0110011
	SUB rd, rs1, rs2	$rd \leftarrow rs1 - rs2$	0100000	000	0110011
	SLT rd, rs1, rs2	$rd \leftarrow rs1 < rs2$ (signed比較)	0000000	010	0110011
	SLTU rd, rs1, rs2	$rd \leftarrow rs1 < rs2$ (unsigned比較)	0000000	011	0110011
	AND rd, rs1, rs2	$rd \leftarrow rs1 \& rs2$ (ビット毎の論理積)	0000000	111	0110011
	OR rd, rs1, rs2	$rd \leftarrow rs1 rs2$ (ビット毎の論理和)	0000000	110	0110011
	XOR rd, rs1, rs2	$rd \leftarrow rs1 ^ rs2$ (ビット毎の排他的論理和)	0000000	100	0110011
	SLL rd, rs1, rs2	$rd \leftarrow rs1 \ll rs2[4:0]$ (論理左シフト)	0000000	001	0110011
	SRL rd, rs1, rs2	$rd \leftarrow rs1 \gg rs2[4:0]$ (論理右シフト)	0000000	101	0110011
	SRA rd, rs1, rs2	$rd \leftarrow rs1 \gg rs2[4:0]$ (算術右シフト)	0100000	101	0110011

シフト演算(SLL, SRL, SRA) : rs2の下位5ビットでシフト量(0~31)が決まり、上位27ビットは無視される

RV32-I命令セット：演算命令(即値オペランド)



種別	アセンブリ命令	レジスタ転送式	imm[11:5]	funct3	opcode
I-type	ADDI rd, rs1, simm	$rd \leftarrow rs1 + simm[11:0]$	000	0010011	
	SLTI rd, rs1, simm	$rd \leftarrow rs1 < simm[11:0]$ (signed比較)	010	0010011	
	SLTIU rd, rs1, simm	$rd \leftarrow rs1 < simm[11:0]$ (unsigned比較)	011	0010011	
	ANDI rd, rs1, simm	$rd \leftarrow rs1 \& simm[11:0]$ (ビット毎の論理積)	111	0010011	
	ORI rd, rs1, simm	$rd \leftarrow rs1 simm[11:0]$ (ビット毎の論理和)	110	0010011	
	XORI rd, rs1, simm	$rd \leftarrow rs1 ^ simm[11:0]$ (ビット毎の排他的論理和)	100	0010011	
	SLLI rd, rs1, uimm	$rd \leftarrow rs1 \ll uimm[4:0]$ (論理左シフト)	0000000	001	0010011
	SRLI rd, rs1, uimm	$rd \leftarrow rs1 \gg uimm[4:0]$ (論理右シフト)	0000000	101	0010011
	SRAI rd, rs1, uimm	$rd \leftarrow rs1 \gg uimm[4:0]$ (算術右シフト)	0100000	101	0010011
U-type	LUI rd, uimm	$rd \leftarrow uimm[31:12]$			0110111
	AUIPC rd, uimm	$rd \leftarrow PC + uimm[31:12]$			0010111

RV32-I命令セット：メモリアクセス命令

11	10	9	8	7	6	5	4	3	2	1	0	rs1	funct3	rd	opcode	I-type	
11	10	9	8	7	6	5	rs2					rs1	funct3	4	3	2	S-type
種別	アセンブリ命令	レジスタ転送式					funct3	opcode									
I-type	LB rd, simm(rs1)	$rd \leftarrow \text{SignExt}(M_8[rs1 + simm[11:0]])$					000	0000011									
	LBU rd, simm(rs1)	$rd \leftarrow \text{ZeroExt}(M_8[rs1 + simm[11:0]])$					100	0000011									
	LH rd, simm(rs1)	$rd \leftarrow \text{SignExt}(M_{16}[rs1 + simm[11:0]])$					001	0000011									
	LHU rd, simm(rs1)	$rd \leftarrow \text{ZeroExt}(M_{16}[rs1 + simm[11:0]])$					101	0000011									
	LW rd, simm(rs1)	$rd \leftarrow M_{32}[rs1 + simm[11:0]]$					010	0000011									
S-type	SB rs2, simm(rs1)	$M_8[rs1 + simm[11:0]] \leftarrow rs2[7:0]$					000	0100011									
	SH rs2, simm(rs1)	$M_{16}[rs1 + simm[11:0]] \leftarrow rs2[15:0]$					001	0100011									
	SW rs2, simm(rs1)	$M_{32}[rs1 + simm[11:0]] \leftarrow rs2[31:0]$					010	0100011									

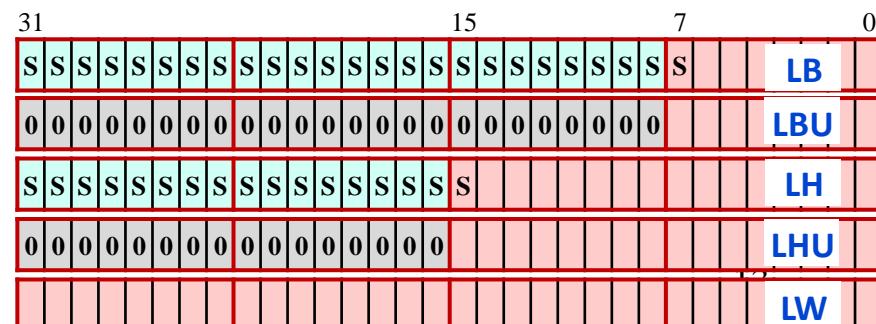
$M_8[addr]$: メモリ上のaddr番地の8ビットデータ

$M_{16}[addr]$: メモリ上のaddr番地の16ビットデータ

$M_{32}[addr]$: メモリ上のaddr番地の32ビットデータ

$\text{SignExt}(X)$: Xの符号拡張(signedビット長拡張)

$\text{ZeroExt}(X)$: Xの0拡張(unsignedビット長拡張)



RV32-I命令セット：プログラム制御命令

	J-type
	I-type
	B-type

種別	アセンブリ命令	レジスタ転送式	funct3	opcode
J-type	JAL rd, <i>paddr</i>	$rd \leftarrow PC + 4, PC \leftarrow PC + simm[20:1]$	1101111	
I-type	JALR rd, simm(rs1)	$rd \leftarrow PC + 4, PC \leftarrow ((rs1 + simm[11:0]) \& (\sim 1))$	000	1100111
B-type	BEQ rs1, rs2, <i>paddr</i>	$\text{if}(rs1 == rs2) PC \leftarrow PC + simm[12:1]$	000	1100011
	BNE rs1, rs2, <i>paddr</i>	$\text{if}(rs1 != rs2) PC \leftarrow PC + simm[12:1]$	001	1100011
	BLT rs1, rs2, <i>paddr</i>	$\text{if}(rs1 < rs2) PC \leftarrow PC + simm[12:1]$ (signed比較)	100	1100011
	BLTU rs1, rs2, <i>paddr</i>	$\text{if}(rs1 < rs2) PC \leftarrow PC + simm[12:1]$ (unsigned比較)	110	1100011
	BGE rs1, rs2, <i>paddr</i>	$\text{if}(rs1 \geq rs2) PC \leftarrow PC + simm[12:1]$ (signed比較)	101	1100011
	BGEU rs1, rs2, <i>paddr</i>	$\text{if}(rs1 \geq rs2) PC \leftarrow PC + simm[12:1]$ (unsigned比較)	111	1100011

paddr : ジャンプ先のPC → 現PCの相対オフセット値 $simm[20:1], simm[12:1]$ で指定

- $((rs1 + simm[11:0]) \& (\sim 1))$: 最下位ビットを0にクリア ($(\sim 1) = 0xffffffff$)
- PCの値(命令アドレス)は2の倍数: 命令長が4バイト、2バイト(c拡張の場合)

アセンブリ言語と機械語

1008: 008000ef	jal	1010 <main> ## サブルーチン
100c: 10500073	wfi	## 割込み待ち(wait for interrupt)
00001010 <main>:		
1010: 007ff797	auipc	a5,0x7ff
1014: ff078793	addi	a5,a5,-16 # 800000 <a>
1018: 007ff697	auipc	a3,0x7ff
101c: 00068693	mv	a3,a3
1020: 00000513	li	a0,0
1024: 0007a703	lw	a4,0(a5)
1028: 00478793	addi	a5,a5,4
102c: 00e50533	add	a0,a0,a4
1030: fed79ae3	bne	a5,a3,1024 <main+0x14>
1034: 00008067	ret	

レジスタ	レジスタ別名
x[0]	zero
x[1]	ra
x[2]	sp
x[3]	gp
x[4]	tp
x[5] - x[7]	t0 - t2
x[8] - x[9]	s0 - s1
x[10] - x[17]	a0 - a7
x[18] - x[27]	s2 - s11
x[28] - x[31]	t3 - t6

type	opcode
R-type	01100 11
I-type	00100 11
U-type	01101 11
U-type	00101 11
I-type	00000 11
S-type	01000 11
J-type	11011 11
I-type	11001 11
B-type	11000 11

type	アセンブリ命令	機械語	命令フィールド		
			1	2	
J	jal 1010 <main>	008000ef	0000 0000 1000 0000 0000 0000 1110 1111 imm[20:1] = 0000 0000 0000 0100 (8)		$x[1] \leftarrow PC + 4,$ $PC \leftarrow PC + (0x0004 \ll 1)$
U	auipc a5,0x7ff	007ff797	0000 0000 0111 1111 1111 0111 1001 0111		$x[15] \leftarrow (0x007ff \ll 12) + 0x1010$
I	addi a5,a5,-16	ff078793	1111 1111 0000 0111 1000 0111 1001 0011		$x[15] \leftarrow x[15] + 0xfffffe0$
U	auipc a3,0x7ff	007ff697	0000 0000 0111 1111 1111 0110 1001 0111		$x[13] \leftarrow (0x007ff \ll 12) + 0x1018$
I	mv a3,a3	00068693	0000 0000 0000 0110 1000 0110 1001 0011		$x[13] \leftarrow x[13] + 0x000$
I	li a0,0	00000513	0000 0000 0000 0000 0000 0101 0001 0011		$x[10] \leftarrow x[0] + 0x000$
I	lw a4,0(a5)	0007a703	0000 0000 0000 0111 1010 0111 0000 0011		$x[14] \leftarrow M_{32}[x[15] + 0x000]$
I	addi a5,a5,4	00478793	0000 0000 0100 0111 1000 0111 1001 0011		$x[15] \leftarrow x[15] + 0x004$
R	add a0,a0,a4	00e50533	0000 0000 1110 0101 0000 0101 0011 0011		$x[10] \leftarrow x[10] + x[14]$
B	bne a5,a3,1024	fed79ae3	1111 1110 1101 0111 1001 1010 1110 0011 imm[12:1] = 1111 1111 1010 ((- 6) << 1)		$\text{if}(x[15] != x[13]) PC \leftarrow PC - 12$
I	ret	00008067	0000 0000 0000 0000 1000 0000 0110 0111		$x[0] \leftarrow PC + 4, (\text{実行されない})$ $PC \leftarrow (x[1] + 0x000) \& (~1)$

演算命令と疑似命令

1010: 007ff797	auipc	a5, 0x7ff
1014: ff078793	addi	a5, a5, -16 # 800000 <a>
1018: 007ff697	auipc	a3, 0x7ff
101c: 00068693	mv	a3, a3
1020: 00000513	li	a0, 0
1024: 0007a703	lw	a4, 0 (a5)
1028: 00478793	addi	a5, a5, 4
102c: 00e50533	add	a0, a0, a4
1030: fed79ae3	bne	a5, a3, 1024 <main+0x14>

レジスタ	レジスタ別名
x[0]	zero
x[1]	ra
x[2]	sp
x[3]	gp
x[4]	tp
x[5] – x[7]	t0 – t2
x[8] – x[9]	s0 – s1
x[10] – x[17]	a0 – a7
x[18] – x[27]	s2 – s11
x[28] – x[31]	t3 – t6

type	opcode
R-type	0110011
I-type	0010011
U-type	0110111
U-type	0010111
I-type	0000011
S-type	0100011
J-type	1101111
I-type	1100111
B-type	1100011

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	rd	opcode
11	10	9	8	7	6	5	4	3	2	1	0		rs1	funct3		rd		opcode			
funct7				rs2			rs1			funct3			rd			opcode					
12	10	9	8	7	6	5		rs2		rs1	funct3	4	3	2	1	11		opcode			

mv a3, a3 (**move**) → addi a3, a3, 0 の疑似命令
 li a0, 0 (**load immediate**) → addi a0, zero, 0 の疑似命令

type	アセンブリ命令	機械語	命令フィールド	
U	auipc a5, 0x7ff	007ff797	0000 0000 0111 1111 1111 0111 1001 0111	$x[15] \leftarrow (0x007ff \ll 12) + 0x1010$
I	addi a5, a5, -16	ff078793	1111 1111 0000 0111 1000 0111 1001 0011	$x[15] \leftarrow x[15] + 0xffffffe0$
U	auipc a3, 0x7ff	007ff697	0000 0000 0111 1111 1111 0110 1001 0111	$x[13] \leftarrow (0x007ff \ll 12) + 0x1018$
I	mv a3, a3	00068693	0000 0000 0000 0110 1000 0110 1001 0011	$x[13] \leftarrow x[13] + 0x000$
I	li a0, 0	00000513	0000 0000 0000 0000 0000 0101 0001 0011	$x[10] \leftarrow x[0] + 0x000$
I	lw a4, 0 (a5)	0007a703	0000 0000 0000 0111 1010 0111 0000 0011	$x[14] \leftarrow M_{32}[x[15] + 0x000]$
I	addi a5, a5, 4	00478793	0000 0000 0100 0111 1000 0111 1001 0011	$x[15] \leftarrow x[15] + 0x004$
R	add a0, a0, a4	00e50533	0000 0000 1110 0101 0000 0101 0011 0011	$x[10] \leftarrow x[10] + x[14]$
B	bne a5, a3, 1024	fed79ae3	1111 1110 1101 0111 1001 1010 1110 0011 imm[12:1] = 1111 1111 1010 ((- 6) << 1)	if($x[15] \neq x[13]$) $PC \leftarrow PC - 12$

サブルーチン呼出し・復帰(リターン)

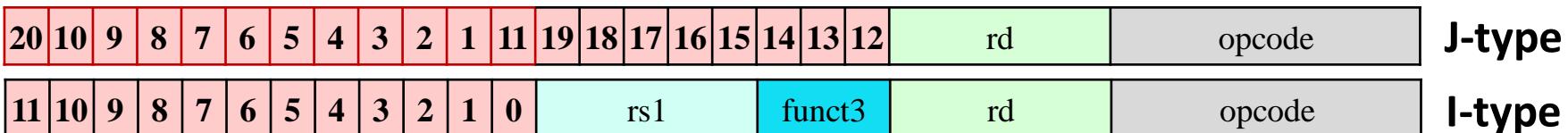
```

1008: 008000ef    jal      1010 <main> ## サブルーチン
100c: 10500073    wfi      ## 割込み待ち(wait for interrupt)
00001010 <main>:
1010: 007ff797    auipc   a5,0x7ff
1014: ff078793    addi    a5,a5,-16 # 800000 <a>
1018: 007ff697    auipc   a3,0x7ff
101c: 00068693    mv      a3,a3
1020: 00000513    li      a0,0
1024: 0007a703    lw      a4,0(a5)
1028: 00478793    addi    a5,a5,4
102c: 00e50533    add     a0,a0,a4
1030: fed79ae3    bne    a5,a3,1024 <main+0x14>
1030: 00008067    ret

```

レジスタ	レジスタ別名
x[0]	zero
x[1]	ra
x[2]	sp
x[3]	gp
x[4]	tp
x[5] - x[7]	t0 - t2
x[8] - x[9]	s0 - s1
x[10] - x[17]	a0 - a7
x[18] - x[27]	s2 - s11
x[28] - x[31]	t3 - t6

type	opcode
R-type	0110011
I-type	0010011
U-type	0110111
U-type	0010111
I-type	0000011
S-type	0100011
J-type	1101111
I-type	1100111
B-type	1100011



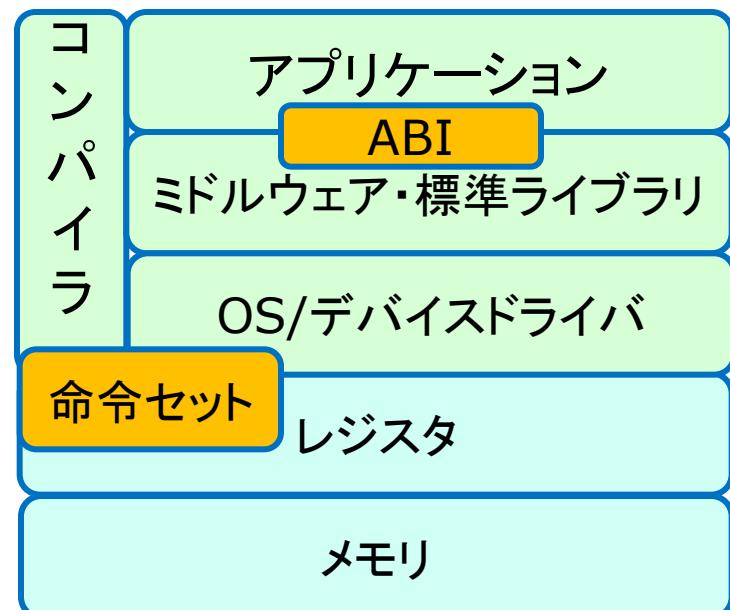
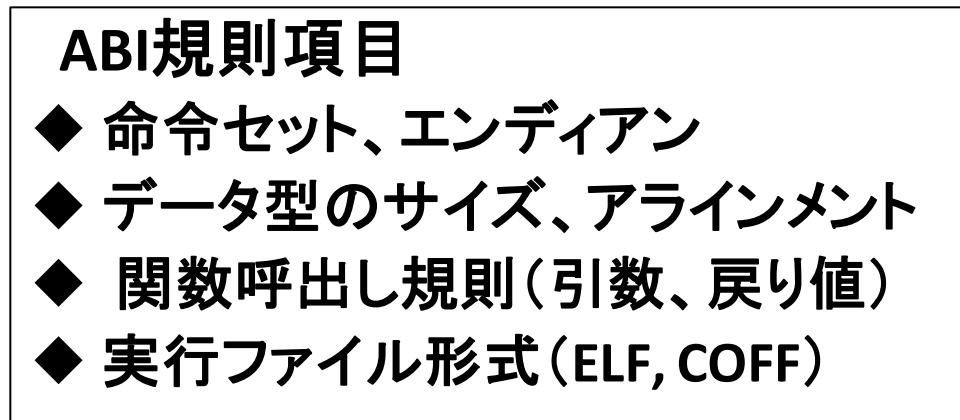
type	アセンブリ命令	機械語	命令フィールド	
J	jal 1010 <main>	008000ef	0000 0000 1000 0000 0000 0000 1110 1111 imm[20:1] = 0000 0000 0000 0100 (8)	$x[1] \leftarrow PC + 4$, $PC \leftarrow PC + (0x0004 \ll 1)$
				$x[1] (ra) : \text{復帰アドレス}(0x1)を格納$ $\text{分岐先アドレス} : 0x1008 + 8 = 0x1010$
I	ret	00008067	0000 0000 0000 0000 1000 0000 0110 0111	$x[0] \leftarrow PC + 4, (\text{実行されない})$ $PC \leftarrow (x[1] + 0x000) \& (\sim 1)$
				$PC \leftarrow ra : \text{サブルーチン呼出し命令の}$ $\text{次の命令}(0x100c: wfi)に復帰する$

資料概要

1. RISC-Vプロセッサとシステム概要
2. RV32-I命令セット仕様
3. ABI : エンディアン、アラインメント、関数呼出し規則
4. 入出力装置と割込み処理機構
5. RV32プログラミング手順 : WSL(Ubuntu)
6. FPGA使用手順 : Windows-11
7. RV32プログラミング課題

ABI (Application Binary Interface)

- ABI : アプリケーションプログラムとシステムソフトウェア(os/ライブラリ)のバイナリ(機械語)レベルの互換性を担保するための規則
- API (Application Programming Interface) : アプリケーションのソースコード上のライブラリとのインターフェース → 関数名、引数、戻り値の定義



エンディアン(Endian)

■ バイト単位データの順番

```
int a = 0x7f010203;  
char *p = (char*)&a;  
printf("p = %02x %02x %02x %02x\n",  
      p[0], p[1], p[2], p[3]);
```

出力(little endian) : p = 03 02 01 7f

出力(big endian) : p = 7f 01 02 03

```
int x = 1;  
bool isLittleEndian = (* (char*) &x) == 1;
```

byte address 3 2 1 0

x : little endian	00	00	00	01
x : big endian	01	00	00	00

byte address	3	2	1	0
byte data (little endian)	7f	01	02	03
byte data (big endian)	03	02	01	7f

x86, ARM, RISC-Vなど主流命令セットはlittle endian

Internet Protocol (IP)はbig endianで統一されている:
network byte order

例) IP address : 127.1.2.3
→ 0x7f010203

データ型のサイズとアライメント

■ C/C++言語のデータ型とアライメント (RV32)

データ型	サイズ (bytes)	アライメント (bytes)
bool	1	1
char	1	1
short	2	2
int	4	4
long	4	4
long long	8	8
void *(ポインタ型)	4	4
float	4	4
double	8	8
long double	16	16

```
struct SB {
    char a;
    short b;
    int c;
    long d;
} b = {1,2,3,4};
```

padding

00800018 :
byte pos: 3 2 1 0
800018: 00020001
80001c: 00000003
800020: 00000004

アライメント: データアドレスはアライメントの倍数となる必要がある → b(2バイト)のアドレスは、a(1バイト: 0x80018)の直後(0x80019)ではなく、0x8001aとなる

```
struct SC {
    char a;
    long long b;
    short c;
} c = {5,6,7};
```

padding

00800028 <c>:
byte pos: 3 2 1 0
800028: 00000005
80002c: 00000000
800030: 00000006
800034: 00000000
800038: 00000007
80003c: 00000000

- 基本データ型では、サイズ=アライメント
- RV64では、ポインタ型は8バイト(サイズ、アライメント)

データ型のサイズとアライメント

■ 構造体データのサイズとアライメント

構造体メンバー 変数	アドレス	サイズ (bytes)	アライメン ト(bytes)
---------------	------	----------------	-------------------

b.a (char)	800018	1	1
b.b (short)	80001a	2	2
b.c (int)	80001c	4	4
b.d (long)	800020	4	4
b (struct sb)	800018	12	4
c.a (char)	800028	1	1
c.b (long long)	800030	8	8
c.c (short)	800040	2	2
c (struct SC)	800028	24	8

```
struct SB {  
    char a;  
    short b;  
    int c;  
    long d;  
} b = {1,2,3,4};
```

padding

00800018 :
byte pos: 3 2 1 0
800018: 00020001
80001c: 00000003
800020: 00000004

アライメント: データアドレスはアライメントの倍数となる必要がある → b(2バイト)のアドレスは、a(1バイト: 0x80018)の直後(0x80019)ではなく、0x8001aとなる

```
struct SC {  
    char a;  
    long long b;  
    short c;  
} c = {5,6,7};
```

padding

00800028 <c>:
byte pos: 3 2 1 0
800028: 00000005
80002c: 00000000
800030: 00000006
800034: 00000000
800038: 00000007
80003c: 00000000

- 構造体データのアライメントは、メンバー変数のアライメントの最大値
- 構造体データのサイズは、アライメントの倍数となる必要がある

関数呼出し規則

- 関数呼出しの引数と戻り値 : a0 – a7レジスタで引き渡す
- コンパイラで割当できないレジスタ : zero, gp, tp
- 関数呼出し前後で値が保持されるレジスタ : sp, s0 – s12

レジスタ	レジスタ別名	説明 ABI mnemonic	コンパイラで 割当可能	関数呼出し前後 で値が保持
x[0]	zero	0に固定	No	---
x[1]	ra	return address	Yes	No
x[2]	sp	stack pointer	Yes	Yes
x[3]	gp	global pointer	No	---
x[4]	tp	thread pointer	No	---
x[5] – x[7]	t0 – t2	temporary registers	Yes	No
x[8] – x[9]	s0 – s1	callee-saved registers	Yes	Yes
x[10] – x[17]	a0 – a7	argument registers	Yes	No
x[18] – x[27]	s2 – s11	callee-saved registers	Yes	Yes
x[28] – x[31]	t3 – t6	temporary registers	Yes	No

サブルーチンとスタックの役割

```

int d[6] = { 2, 3, 5, 7, 11, 13 };
int main() {
    int sum = 0, i;
    for (i = 0; i < 6; i++) {
        sum += d[i] * d[i];
    }
    return sum;
}

```

00001000 <_start>:	
1000: auipc	sp, 0x802
1004: mv	sp, sp
1008: jal	1034 <main>
100c: wfi	

00001010 <__mulsi3>:	
1010: mv	a2, a0
1014: li	a0, 0
1018: andi	a3, a1, 1
101c: beqz	a3, 1024
1020: add	a0, a0, a2
1024: srl	a1, a1, 0x1
1028: slli	a2, a2, 0x1
102c: bnez	a1, 1018
1030: ret	

00001034 <main>:

1034: addi	sp, sp, -16
1038: sw	s0, 8(sp)
103c: sw	s1, 4(sp)
1040: sw	s2, 0(sp)
1044: sw	ra, 12(sp)
1048: auipc	s0, 0x7ff
104c: addi	s0, s0, -72 # 800000 <d>
1050: auipc	s2, 0x7ff
1054: addi	s2, s2, -56 # 800018
1058: li	s1, 0
105c: lw	a1, 0(s0)
1060: addi	s0, s0, 4
1064: mv	a0, a1
1068: jal	1010 <__mulsi3>
106c: add	s1, s1, a0
1070: bne	s0, s2, 105c
1074: lw	ra, 12(sp)
1078: lw	s0, 8(sp)
107c: lw	s2, 0(sp)
1080: mv	a0, s1
1084: lw	s1, 4(sp)
1088: addi	sp, sp, 16
108c: ret	

00800000 <d>:	
800000: 00000002	
800004: 00000003	
800008: 00000005	
80000c: 00000007	
800010: 0000000b	
800014: 0000000d	

sum += d[i] * d[i];

RV32-I : 乗算命令がない
ので、乗算サブルーチン
で実現

サブルーチンとスタックの役割

```
int d[6] = { 2, 3, 5, 7, 11, 13 };
int main() {
    int sum = 0, i;
    for (i = 0; i < 6; i++) {
        sum += d[i] * d[i];
    }
    return sum;
}
```

```
int main() {
    s0 = d; // 0x80000
    s2 = d + 0x18;
    s1 = 0;
    do {
        a1 = M32[s0 + 1];
        s0 += 4;
        a0 = a1;
        a0 = _mulsi3(a0, a1);
        s1 += a0;
    } while (s0 != s2);
    a0 = s1;
    return a0;
}
```

- a0 = _mulsi3(a0, a1) → 引数(a0, a1), 戻り値(a0)
- s0, s1, s2 : _mulsi3(a0, a1)呼出し前後で値が保持される

<pre>00001034 <main>: 1034: addi sp,sp,-16 1038: sw s0,8(sp) 103c: sw s1,4(sp) 1040: sw s2,0(sp) 1044: sw ra,12(sp) 1048: auipc s0,0x7ff 104c: addi s0,s0,-72 # 800000 1050: auipc s2,0x7ff 1054: addi s2,s2,-56 # 800018 1058: li s1,0 105c: lw a1,0(s0) 1060: addi s0,s0,4 1064: mv a0,a1 1068: jal 1010 < _mulsi3> 106c: add s1,s1,a0 1070: bne s0,s2,105c 1074: lw ra,12(sp) 1078: lw s0,8(sp) 107c: lw s2,0(sp) 1080: mv a0,s1 1084: lw s1,4(sp) 1088: addi sp,sp,16 108c: ret</pre>	<pre>00800000 <d>: 800000: 00000002 800004: 00000003 800008: 00000005 80000c: 00000007 800010: 0000000b 800014: 0000000d</pre>
<p>s0 = d; (0x80000)</p>	<p>s2 = d + 0x18; (0x80018)</p>
<p>s1 = 0; $a1 = M_{32}[s0 + 1];$ $s0 += 4;$ $a0 = a1;$ $a0 = \text{_mulsi3}(a0, a1);$ $s1 += a0;$</p>	<p>if ($s0 \neq s2$) goto 105c</p>
<p>a0 = s1;</p>	<p>return a0;</p>

サブルーチンとスタックの役割

00001000 <_start>:

```

1000: auipc    sp, 0x802   sp ← 0x803000
1004: mv       sp, sp
1008: jal      1034 <main> (spの初期化)
100c: wfi

```

00001034 <main>:

```

1034: addi    sp, sp, -16   sp ← 0x802ff0
1038: sw       s0, 8(sp)
103c: sw       s1, 4(sp)
1040: sw       s2, 0(sp)
1044: sw       ra, 12(sp)
1048: auipc   s0, 0x7ff
104c: addi   s0, s0, -72 # 800000 <d>
1050: auipc   s2, 0x7ff
1054: addi   s2, s2, -56 # 800018
1058: li       s1, 0
105c: lw       a1, 0(s0)
1060: addi   s0, s0, 4
1064: mv       a0, a1
1068: jal     1010 <__mulsr3>
106c: add     s1, s1, a0
1070: bne     s0, s2, 105c
1074: lw       ra, 12(sp)
1078: lw       s0, 8(sp)
107c: lw       s2, 0(sp)
1080: mv     a0, s1
1084: lw       s1, 4(sp)
1088: addi   sp, sp, 16   sp ← 0x803000
108c: ret

```

00800000:<d> #グローバル変数領域

```

800000: 00000002
800004: 00000003
800008: 00000005
80000c: 00000007
800010: 0000000b
800014: 0000000d

```

- データメモリ配置(グローバル変数領域、スタック領域):コンパイル時にリンクスクリプトファイル等で指定

データアドレス領域	割当てられる領域
800000 – 800ffff	グローバル変数領域
801000 – 802ffff	スタック領域

- スタック領域:サブルーチン処理中で使用するデータメモリ領域
- スタックポインタ(sp):使用中のスタック領域の先頭アドレスを指す

サブルーチンとスタックの役割

00001000 <_start>:

```

1000: auipc    sp, 0x802
1004: mv       sp, sp
1008: jal   1034 <main>
100c: wfi

```

$sp \leftarrow 0x803000$
(spの初期化)

- $ra, s0, s1, s2 : main$ 内で更新されるため、
スタック領域に退避・復元する必要がある

00001034 <main>:

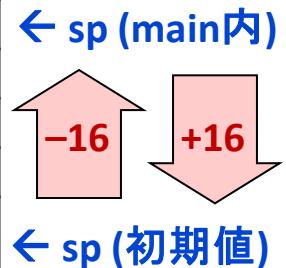
```

1034: addi   sp, sp, -16    $sp \leftarrow 0x802ff0$ 
1038: sw     s0, 8(sp)
103c: sw     s1, 4(sp)
1040: sw     s2, 0(sp)
1044: sw   ra, 12(sp)   stackへ退避
1048: auipc  s0, 0x7ff
104c: addi  s0, s0, -72 # 800000 <d>
1050: auipc  s2, 0x7ff
1054: addi  s2, s2, -56 # 800018
1058: li    s1, 0
105c: lw    a1, 0(s0)
1060: addi  s0, s0, 4
1064: mv    a0, a1
1068: jal 1010 <_mulsi3>    $ra \leftarrow 0x106c$ 
106c: add   s1, s1, a0    $pc \leftarrow 0x1010$ 
1070: bne
1074: lw    ra, 12(sp)
1078: lw    s0, 8(sp)
107c: lw   s2, 0(sp)   stackから復元
1080: mv    a0, s1
1084: lw    s1, 4(sp)
1088: addi  sp, sp, 16    $sp \leftarrow 0x803000$ 
108c: ret

```

main内の
スタック領域確保

アドレス	main 内スタック領域
802ff0	s2 退避 : 0(sp)
802ff4	s1 退避 : 4(sp)
802ff8	s0 退避 : 8(sp)
802ffc	ra 退避 : 12(sp)
803000	Bottom of stack (未使用領域)

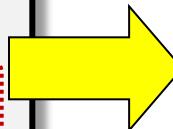


- mainに突入後、スタック領域を確
保し($sp -= 16$)、mainから戻る時につ
タック領域を解放する($sp += 16$)
- sp : 関数呼出し前後で値は不変

乗算サブルーチン(_mulsi3)

- $a0 = \text{mulsi3}(a0, a1) \rightarrow \text{引数}(a0, a1), \text{戻り値}(a0)$
- 乗算サブルーチン : RV32-I命令セットは乗算命令を実装していないため、コンパイラ内蔵のライブラリ関数としてsw実装される

```
00001010 <_mulsi3>:
    1010: mv      a2,a0
    1014: li      a0,0
    1018: andi   a3,a1,1
    101c: beqz   a3,1024
    1020: add    a0,a0,a2
    1024: srli   a1,a1,0x1
    1028: slli   a2,a2,0x1
    102c: bnez   a1,1018
    1030: ret
```



```
int _mulsi3(int a0, int a1) {
    a2 = a0;
    a0 = 0;
    do {
        a3 = a1 & 1;
        if (a3 != 0)
            a0 = a0 + a2;
        a1 = a1 >> 1;
        a2 = a2 << 1;
    } while (a1 != 0);
    return a0;
}
```

$a0 = 13, a1 = 13$ の計算例

0000000000001101	(13)
x 0000000000001101	(13)
0000000000001101	
0000000000000000	
000000000000110100	
0000000000001101000	
00000000000010101001	(169)

a1	a3	a2	a0
00001101	1	00001101	00001101
00000110	0	00011010	00001101
00000011	1	00110100	01000001
00000001	1	01101000	10101001
00000000		do-while(a1!=0) 終了	

$$a0 = a0 + a2$$

$$\begin{array}{r} 00000000 \\ +00001101 \\ \hline 00001101 \end{array} \quad (13)$$

$$\begin{array}{r} 00001101 \\ +01101000 \\ \hline 01000001 \end{array} \quad (52)$$

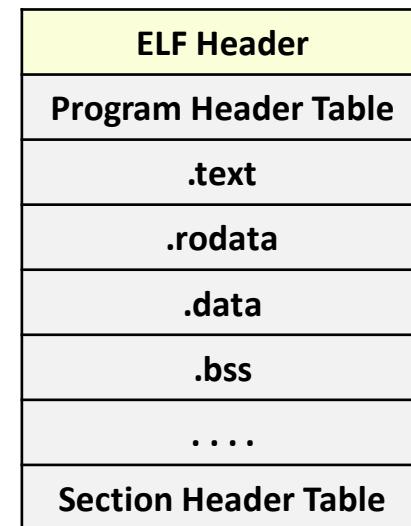
$$\begin{array}{r} 01000001 \\ +01101000 \\ \hline 10101001 \end{array} \quad (169)$$

実行ファイル形式(ELF, COFF)

- オブジェクトファイル：プログラムソースファイル単位でコンパイラが出力する機械語命令列を含む実行プログラム情報
- 実行ファイル：アプリケーション単位でコンパイラが出力する実行プログラム情報（複数のオブジェクトファイルをマージし、シンボルをメモリアドレスに配置する）
- シンボル：メモリアドレスの配置対象となる関数やglobal変数
- COFF (Common Object File Format) : Windowsの実行ファイル形式
- ELF (Executable and Linkable Format) : Linuxの実行ファイル形式

ELF形式概要：

- ELF Header : アーキテクチャ(命令セット)情報、Entry Point Address (最初に実行する命令アドレス)、他のテーブル情報など
- Program Header Table : ELFファイル内の各セグメント(セクションの集まり)のメモリ上への配置方法やアクセス属性(Executable, readable, writeableなど)の情報
- Section Header Table : メモリ上へ配置するセクションの情報
- セクション種類：
 - .text : 機械語命令列(関数シンボルの集まり)
 - .rodata : read-only グローバル変数
 - .data : グローバル変数(初期値データ列)
 - .bss : 0初期化global/static変数



資料概要

1. RISC-Vプロセッサとシステム概要
2. RV32-I命令セット仕様
3. ABI : エンディアン、アラインメント、関数呼出し規則
4. 入出力装置と割込み処理機構
5. RV32プログラミング手順 : WSL(Ubuntu)
6. FPGA使用手順 : Windows-11
7. RV32プログラミング課題

入出力装置 (Memory-Mapped IO)

Memory-mapped IO : 入出力装置(IO)をプログラムからアクセスするために、各IOに固有のアドレス値を割り当て、IOのアドレスへロード命令(読み出し)、ストア命令(書き込み)で行う

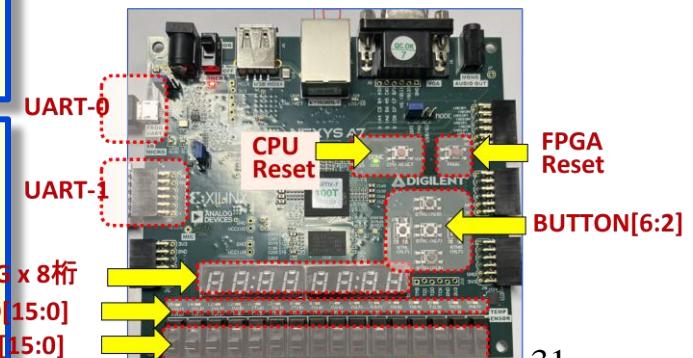
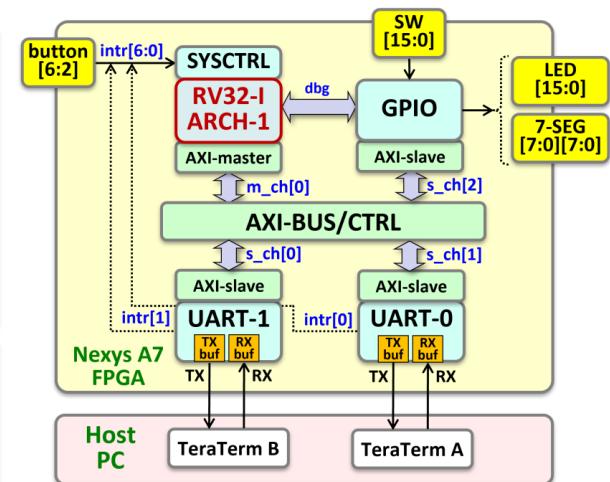
```
#define GPIO_ADDR      0x10000080 // memory-mapped GPIO

typedef volatile struct ST_GPIO {
    unsigned int data;      // read SW[15:0], write 7SEG (hex-mode)
    unsigned int dout7SEG[2]; // write 7SEG (raw-mode)
    unsigned short doutLED; // write LED[15:0]
} GPIO;
GPIO* gpio = (GPIO*) GPIO_ADDR;
```

```
#define UART_0_ADDR     0x10000000 // memory-mapped UART-0
#define UART_1_ADDR     0x10000040 // memory-mapped UART-1

typedef volatile struct ST_UART {
    char data;        // read RX-buf, write TX-buf
    unsigned ctrl;   // read RX/TX status, write RX/TX settings
} UART;
UART* _uart[2] = { (UART*)_UART_0_ADDR, (UART*)_UART_1_ADDR };
```

```
#define SYSCTRL_ADDR   0x10001000 // memory-mapped SYSCTRL
typedef volatile struct ST_SYSCTRL {
    unsigned int irq_ctrl; // 割込み検知状態制御
    ...
    unsigned int irq_enable; // 割込み検知許可設定
    ... // タイマー制御レジスタ
} SYSCTRL;
extern SYSCTRL* sysctrl = (SYSCTRL*) SYSCTRL_ADDR;
```



GPIO : 汎用入出力装置

- 入力 : SW[15:0]
- 出力 : 7-SEG[7:0] x 8行、LED[15:0]

```
#define GPIO_ADDR      0x10000080 // memory-mapped GPIO

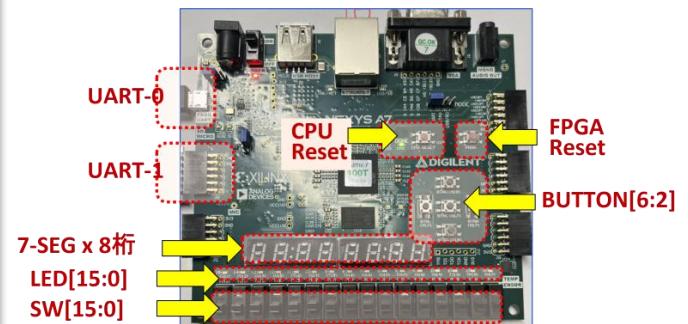
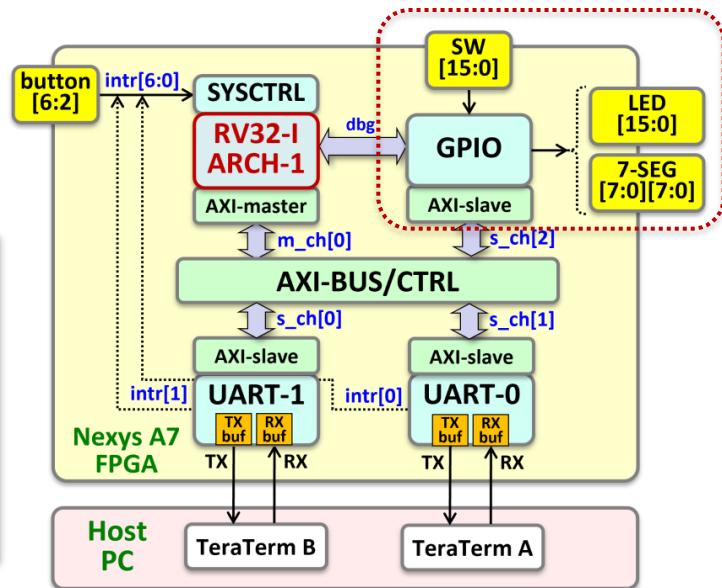
typedef volatile struct ST_GPIO {
    unsigned int data;          /// read SW[15:0], write 7SEG (hex-mode)
    unsigned int dout7SEG[2];   /// write 7SEG (raw-mode)
    unsigned short doutLED;    /// write LED[15:0]
} GPIO;
GPIO* gpio = (GPIO*) GPIO_ADDR;
```

```
///////// read SW[15:0] //////////
unsigned short sw = (unsigned short) gpio->data;

///////// write 7SEG (hex-mode) 4-bits x 8-digit 7SEG display //////////
gpio->data = 0x1234abcd;

///////// write 7SEG (raw-mode) 8-bits x 8-digit 7SEG display //////////
/// '1' '2' '3' '4' 'a' 'b' 'c' 'd'
gpio->dout7SEG[0] = 0x777c395e; // 'a' 'b' 'c' 'd' (7SEG patterns)
gpio->dout7SEG[1] = 0x065b4f66; // '1' '2' '3' '4' (7SEG patterns)

///////// write LED[15:0] //////////
gpio->doutLED = (unsigned short)0x1234; // LED : 0001 0010 0011 0100
```

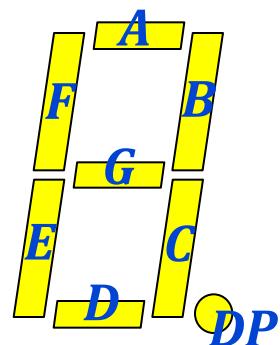


7-SEGMENT表示装置 (8 bits x 8 digits)

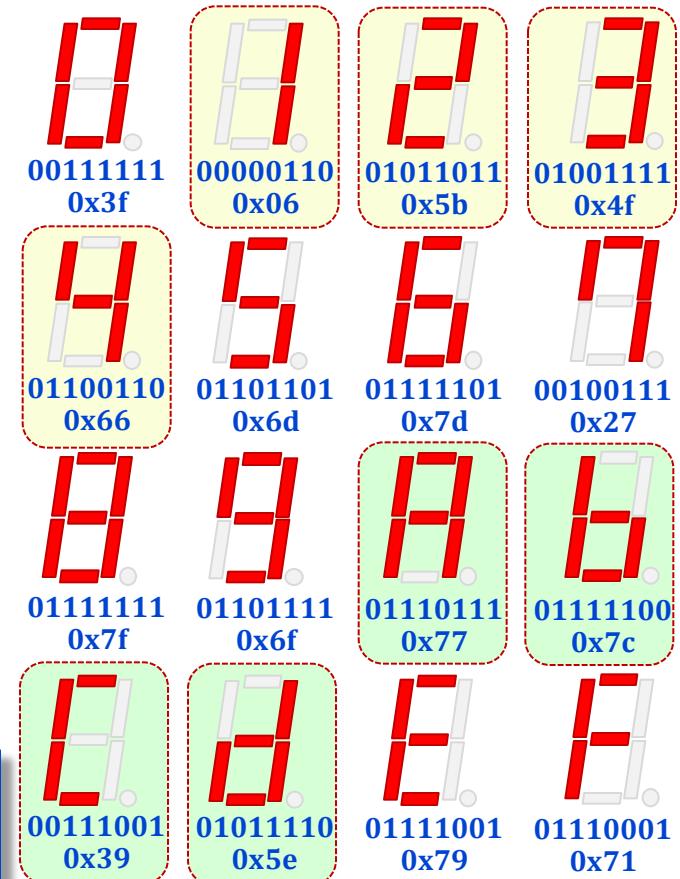


{ DP, G, F, E, D, C, B, A }

```
/// { DP, G, F, E, D, C, B, A }
S7_0 = 0b00111111,    /// 0 : 0x3f
S7_1 = 0b00000010,    /// 1 : 0x06
S7_2 = 0b01011011,    /// 2 : 0x5b
S7_3 = 0b01001111,    /// 3 : 0x4f
S7_4 = 0b01100110,    /// 4 : 0x66
S7_5 = 0b01101101,    /// 5 : 0x6d
S7_6 = 0b01111101,    /// 6 : 0x7d
S7_7 = 0b00100011,    /// 7 : 0x27
S7_8 = 0b01111111,    /// 8 : 0x7f
S7_9 = 0b01101111,    /// 9 : 0x6f
S7_a = 0b01110111,    /// A : 0x77
S7_b = 0b01111100,    /// b : 0x7c
S7_c = 0b00111001,    /// C : 0x39
S7_d = 0b01011110,    /// d : 0x5e
S7_e = 0b01111001,    /// E : 0x79
S7_f = 0b01110001,    /// F : 0x71
```



```
//////// write 7SEG (raw-mode) 8-bits x 8-digit 7SEG display //////////
/// '1' '2' '3' '4' 'a' 'b' 'c' 'd'
gpio->dout7SEG[0] = 0x777c395e; // 'a' 'b' 'c' 'd' (7SEG patterns)
gpio->dout7SEG[1] = 0x065b4f66; // '1' '2' '3' '4' (7SEG patterns)
```



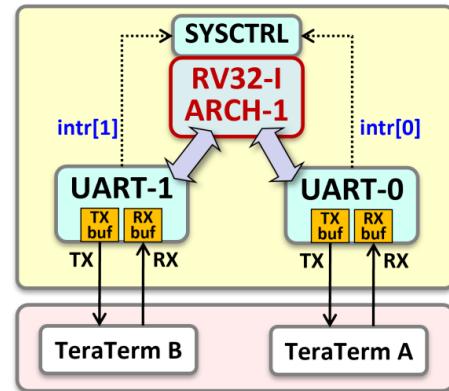
UARTポート（シリアル通信）

(Universal Asynchronous Receiver Transmitter)

- シリアル通信: 1ビット毎の受信・送信(非同期通信)
- Baud rate : 1秒当たりの転送ビット数 → 実験用FPGAボードでは38,400 bits/秒にデフォルトで設定
- 1ビット当たりのクロック数(プロセッサの動作周波数が50MHzの場合) : $50M / 38.4K = 1,302 \text{ clks}$

```
#define UART_0_ADDR 0x10000000 // memory-mapped UART-0
#define UART_1_ADDR 0x10000040 // memory-mapped UART-1

typedef volatile struct ST_UART {
    char data; // read RX-buf, write TX-buf
    unsigned ctrl; // read RX/TX status, write RX/TX settings
} UART;
UART* _uart[2] = { (UART*)_UART_0_ADDR, (UART*)_UART_1_ADDR };
```



UART::dataポート	説明
Read (8 bit入力)	data[7:0] ← 受信バッファ
Write (8 bit出力)	data[7:0] → 送信バッファ

RX_empty	受信バッファが EMPTY
RX_full	受信バッファが FULL
TX_empty	送信バッファが EMPTY
TX_full	送信バッファが FULL

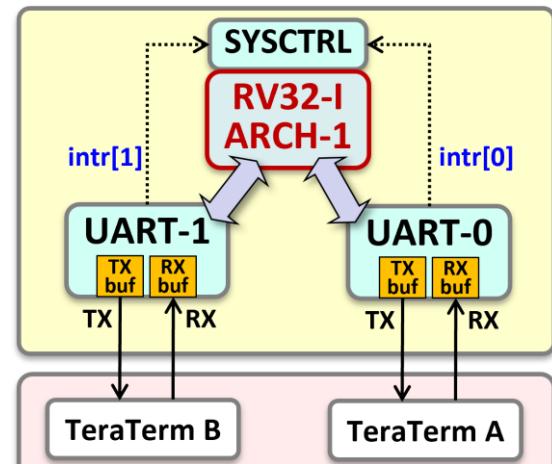
UART::ctrl ポート	説明
Read (4 bit入力)	ctrl[3:0] ← { RX_empty, RX_full, TX_empty, TX_full } (送受信バッファ状態)
Write (11 bit出力)	ctrl[10:0] → { parity[2:0], stop_bit, length[1:0], baud[3:0], activate } (UARTポート設定 : Baud rate, ビット長, その他パラメータ)

UARTのプログラム実行同期（ポーリング方式）

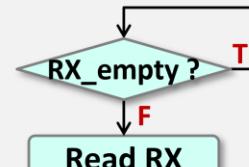
- データ受信：受信バッファがEMPTY状態でないことを確認して、受信データを読み出す
- データ送信：送信バッファがFULL状態でないことを確認して、送信データを書き込む
- ポーリング方式の欠点：シリアル通信はプログラム実行速度に比べ大幅に遅いため、長期間のポーリング同期中は計算処理ができない。また、送信と受信が同時に起こる場合は、受信バッファがFULL状態になった後の受信データが失われる。

ctrl[3:0] ← { RX_empty, RX_full, TX_empty, TX_full }

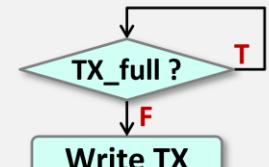
```
#define UART_TX_FULL(uart)      (((uart->ctrl >> 0) & 1)
#define UART_TX_EMPTY(uart)     (((uart->ctrl >> 1) & 1)
#define UART_RX_FULL(uart)      (((uart->ctrl >> 2) & 1)
#define UART_RX_EMPTY(uart)     (((uart->ctrl >> 3) & 1)
UART*_uart[2] = { (UART*)_UART_0_ADDR, (UART*)_UART_1_ADDR };
int UART_CHAN_ID = 0; // UARTチャネルID
////////////////// データ受信(ポーリング方式) ///////////////////
char _io_getch() {
    UART* uart = _uart[UART_CHAN_ID];
    while (UART_RX_EMPTY(uart)) {} // 受信バッファがEMPTYの間待機
    return uart->data; // 受信データを読み出す
}
////////////////// データ送信(ポーリング方式) ///////////////////
void _io_putch(char ch) {
    UART* uart = _uart[UART_CHAN_ID];
    while (UART_TX_FULL(uart)) {} // 送信バッファがFULLの間待機
    uart->data = ch; // 送信データを書き込む
}
```



_io_getch()



_io_putch()



割込み処理機構

- 外部装置(ボタン入力、UARTポート)の状態変化を「割込み信号」としてプロセッサに通知
 - UART受信割込み : 1 byte のデータ受信が完了したときに発生 → (`RX_empty = 0`) が保証される
 - UART送信割込み : 1 byte のデータ送信が開始したときに発生 → (`TX_full = 0`) が保証される
- 割込み処理 : 割込み信号を検知したときに、実行プログラムを中断し、UARTデータ送受信処理を割込みルーチン(送受信データをメモリ上の送受信バッファに格納する処理)で行い、再び元のプログラムに復帰する
- 7ビット割込み信号 : UART2系統 (`intr[1:0]`)、BUTTON[6:2] (`intr[6:2]`)
- 割込み検知制御論理 (`SYSCTRL`) : memory-mapped IOとしてアクセス
 - `irq_ctrl[6:0]` : (Read) 割込み信号 `intr[6:0]` の各ビットの検知状態、(Write) 割込み検知状態クリア(1のビット位置の割込み検知状態ビットをクリア)
 - `ext_irq_enable[6:0]` : 割込み信号 `intr[6:0]` の各ビットの検知許可(1のビット位置を検知許可)
- CSR (Control and Status Registers) : 割込み処理やOSのための専用レジスタ

```
#define SYSCTRL_ADDR 0x10001000 // memory-mapped SYSCTRL
typedef volatile struct ST_SYSCTRL {
    unsigned int irq_ctrl; // 割込み検知状態制御
    unsigned int pad0;
    unsigned int irq_enable; // 割込み検知許可設定
    unsigned int pad1;
    unsigned long long mtime; // 64-bit タイマー
    unsigned long long mtimecmp; // 64-bit タイマー比較値
} SYSCTRL;
extern SYSCTRL* sysctrl = (SYSCTRL*) SYSCTRL_ADDR;
```

SYSCTRL割込み関連レジスタ

レジスタ	説明
<code>irq_pending[6:0]</code>	<code>intr[6:0]</code> の (0->1) 遷移を検知 (プログラムからアクセス不可)
<code>irq_enable[6:0]</code>	割込み許可ビット
	<code>irq_ctrl</code> の read/write 動作
<code>read</code>	<code>irq_ctrl = irq_pending & irq_enable;</code>
<code>write</code>	<code>irq_pending &= ~irq_ctrl;</code>

CSR (Control and Status Registers)

- 外部割込み検知条件 : mstatus[3] & mie[11] & (sysctrl->irq_ctrl != 0)
- タイマー割込み検知条件 : mstatus[3] & mie[7] & (タイマー割込み条件)

割込み処理に関連したCSR		
レジスタ	アドレス	説明
mstatus[31:0]	0x300	mstatus[3] : グローバル割込み許可ビット
mie[31:0]	0x304	mie[11] : 外部割込み許可ビット mie[7] : タイマー割込み許可ビット mie[3] : SW割込み許可ビット
mtvec[31:0]	0x305	割込みハンドラー地址
mepc[31:0]	0x341	割込みハンドラー復帰アドレス
mcause[31:0]	0x342	割込み・例外の発生起因情報
mip[31:0]	0x344	mip[11] : 外部割込み検知ビット mip[7] : タイマー割込み検知ビット mi[3] : SW割込み検知ビット

CSR address (12 bits)	rs1 / uimm (5 bits)	funct3 (3 bits)	rd (5 bits)	opcode 1110011
CSR操作命令				
アセンブリ	funct3	動作		
csrrw rd, csr, rs1	001	rd ← csr, csr ← rs1		
csrrs rd, csr, rs1	010	rd ← csr, csr ← csr rs1		
csrrc rd, csr, rs1	011	rd ← csr, csr ← csr & ~rs1		
csrrwi, rd, csr, uimm	101	rd ← csr, csr ← uimm		
csrrsi rd, csr, uimm	110	rd ← csr, csr ← csr uimm		
csrrci rd, csr, uimm	111	rd ← csr, csr ← csr & ~uimm		

タイマー・カウンタ系CSR		
レジスタ	アドレス	説明
cycle[31:0]	0xc00	mcycle[31:0]と同一
time[31:0]	0xc01	sysctrl->mtime[31:0]と同一
instret[31:0]	0xc02	minstret[31:0]と同一
cycleh[31:0]	0xc80	mcycleh[31:0]と同一
timeh[31:0]	0xc81	sysctrl->mtime[63:32]と同一
instreth[31:0]	0xc82	minstreth[31:0]と同一
mcycle[31:0]	0xb00	実行サイクル数(下位32ビット)
minstret[31:0]	0xb02	実行命令数(下位32ビット)
mcycleh[31:0]	0xb80	実行サイクル数(上位32ビット)
minstreth[31:0]	0xb82	実行命令数(上位32ビット)

SYSCTRLタイマー関連レジスタ

レジスタ	説明
mtime[63:0]	16サイクルでインクリメント
mtimecmp[63:0]	タイマー比較値
タイマー割込み条件	
mtime >= mtimecmp;	

UARTポート割込み有効化処理 (enable_UART_interrupt)

- 割込み検知許可:
 - `sysctrl->ext_irq_enable |= 3;` (`intr[1:0]`: UART2系統の割込み検知許可ビットを1に設定)
 - `csrs mie, 0x800` (`mie[11]`: 外部割込み許可ビット)を1に設定
 - `csrs mstatus 0x008` (`mstatus[3]`: グローバル割込み許可ビット)を1に設定
- 割込みハンドラーアドレス設定:
 - `csrw mtvec _mhandler;` (_mhandler関数を割込みハンドラーアドレス`mtvec`に登録)

```
/// UART受信関数ポインタ : io_getch
void (*io_getch)(char) = _io_getch; // デフォルト設定 : UART受信関数(ポーリング方式)

/// UART送信関数ポインタ : io_putch
char (*io_putch)(void) = _io_putch; // デフォルト設定 : UART受信関数(ポーリング方式)

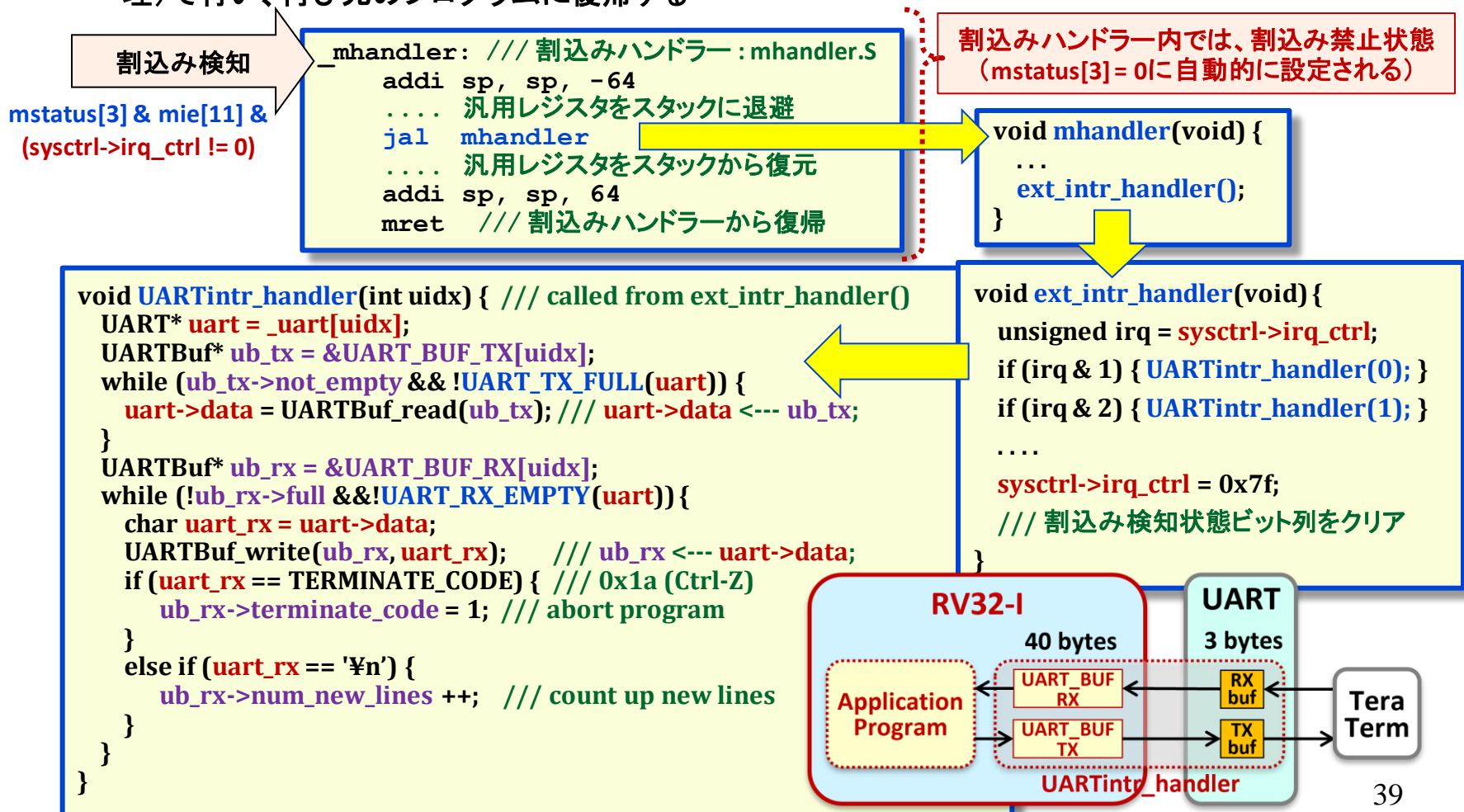
void enable_UART_interrupt() { // UARTポート割込み許可処理
    unsigned csr_val;
    UARTBuf_reset_all(); // メモリのUART送受信バッファを初期化
    io_getch = _UARTBuf_getch; // UART受信関数(割込み方式)に変更
    io_putch = _UARTBuf_putch; // UART送信関数(割込み方式)に変更
    sysctrl->ext_irq_enable |= 3; // enable uart1 uart0 interrupts

    asm volatile("csrs mie, %[new] :: [new] "r" (0x800)); // set mie[11]
    asm volatile("csrs mstatus, 0x008 ::); // set mstatus[3]
    asm volatile("csrw mtvec, %[new] :: [new] "r" (_mhandler)); // 割込みハンドラーアドレス設定
}
```

cプログラムにRISC-Vアセンブリ語命令を挿入

UARTポート割込みハンドラー

- **割込み処理:** 割込み信号を検知したときに、実行プログラムを中断し、UARTデータ送受信処理を割込みルーチン(送受信データをメモリ上の送受信バッファ(**UART_BUF_RX, UART_BUF_TX**)に格納する処理)で行い、再び元のプログラムに復帰する

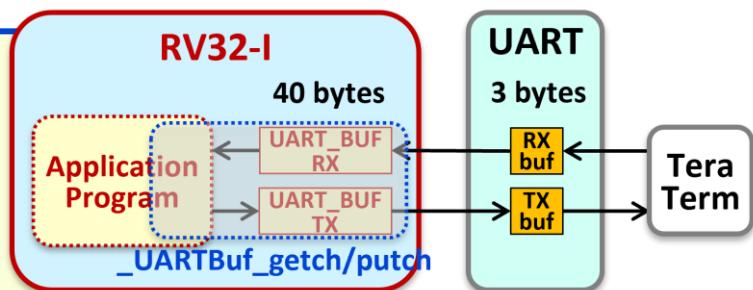


UART送受信関数（割込み方式）

- データ受信：受信データをメモリ上の受信バッファから読み出す
- データ送信：送信データをメモリ上の送信バッファに書き込む

```
////////// データ受信(割込み方式) ///////////
char _UARTBuf_getch() {
    UARTBuf* ub_rx = &UART_BUF_RX[UART_CHAN_ID];
    while (!ub_rx->not_empty) {} // wait until (ub_rx->not_empty == 1)
    enter_critical_section();
    char ch = UARTBuf_read(ub_rx);
    leave_critical_section();
    return ch;
}

////////// データ送信(割込み方式) ///////////
void _UARTBuf_putch(char ch) {
    UARTBuf* ub_tx = &UART_BUF_TX[UART_CHAN_ID];
    while (ub_tx->full) {} // wait until (ub_tx->full == 0)
    UART* uart = _uart[UART_CHAN_ID];
    enter_critical_section();
    UARTBuf_write(ub_tx, ch);
    if (!UART_TX_FULL(uart)) { // if (tx_full == 0);
        uart->data = UARTBuf_read(ub_tx);
    }
    leave_critical_section();
}
```



```
static unsigned prv_mstatus = 0, critical_section_depth = 0;

void enter_critical_section() {
    if ((critical_section_depth++) == 0) {
        asm volatile("csrr %[old], mstatus" : [old] "=r" (prv_mstatus));
        asm volatile("csrw mstatus, 0x0008");
    } else { set_gpio_error(0); }
}

void leave_critical_section() {
    if ((--critical_section_depth) == 0) {
        asm volatile("csrw mstatus, %[new]" : [new] "r" (prv_mstatus));
    } else { set_gpio_error(1); }
}
```

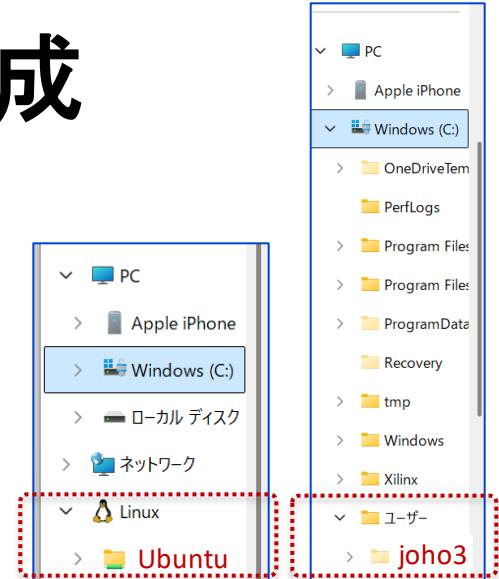
Critical Section : 割込み禁止領域 → メモリ上の送受信バッファ (UART_BUF_RX, UART_BUF_TX) の更新中は割込み処理が発生しないための排他制御

資料概要

1. RISC-Vプロセッサとシステム概要
2. RV32-I命令セット仕様
3. ABI : エンディアン、アラインメント、関数呼出し規則
4. 入出力装置と割込み処理機構
5. RV32プログラミング手順 : WSL(Ubuntu)
6. FPGA使用手順 : Windows-11
7. RV32プログラミング課題

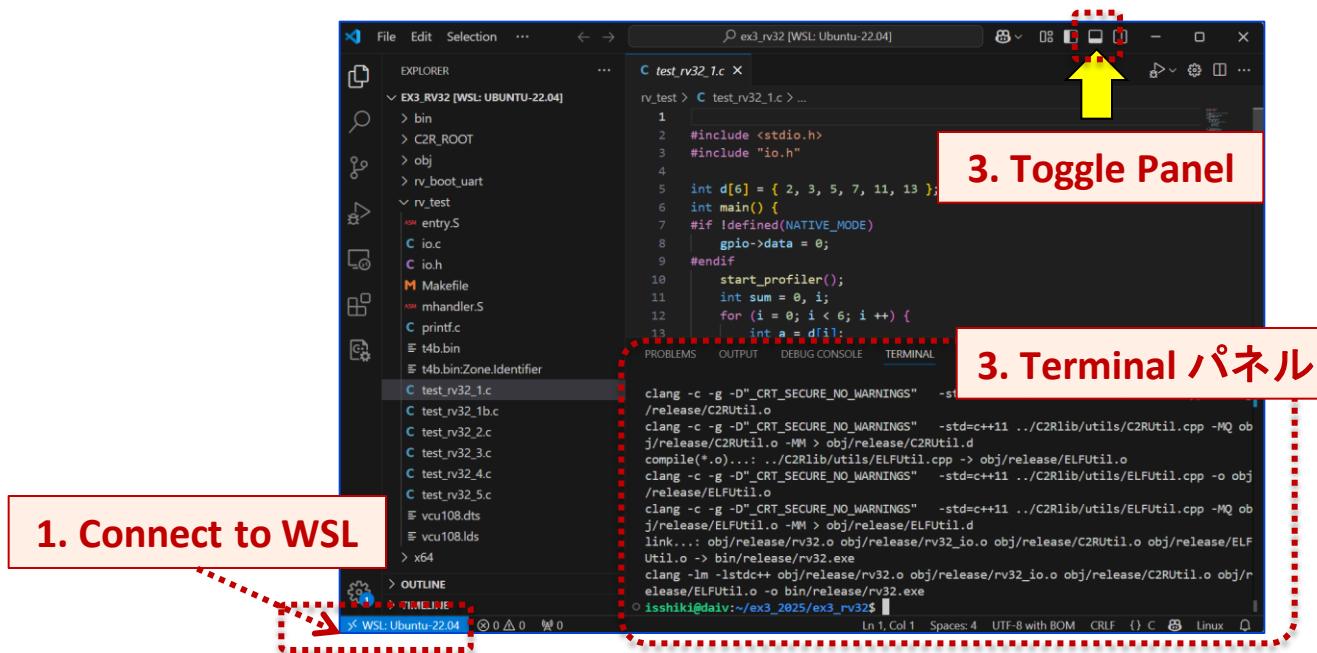
ex3_2025 フォルダ構成

- ex3_2025 フォルダのコピー場所:
 - Windows : C:/Users/joho3/ex3_2025
 - Linux/Ubuntu : /home/joho3/ex3_2025
- ex3_2025/documents/ : 実験説明書関連
 - ex3_1.pptx : RISC-Vプログラミング編(本書) : 課題1.A
 - ex3_2.pptx : RISC-Vプロセッサ基本設計編
 - ex3_3.pptx : RISC-V Verilog設計編(準備中) : 課題1.B
- ex3_2025/ex3_rv32/ : RV32シミュレータ、cプログラムビルド・実行環境
 - rv_test/ : RV32用cプログラムサンプル、ビルド・実行環境
- ex3_2025/C2Rlib/ : RV32シミュレータ用ライブラリ
- ex3_2025/FPGA/ : FPGA実行環境
 - FPGA/ex3_rv32_2025/ : FPGAボード用データ
 - FPGA/TERATERM.INI : Teraterm設定ファイル



VS-Codeプログラミング環境

1. VS-Codeを起動し、左下のボタンをクリックし"Connect to WSL"を選択 → 左下に"WSL Ubuntu"が表示される
2. File → Open Folderで/home/joho3/ex3_2025/ex3_rv32を開く
3. Terminal パネルを開き(上の"Toggle Panel"ボタンをクリック)、makeを実行 → ex3_rv32/bin/release/rv32.exe(RV32シミュレータ)がビルドされる



サンプルCプログラム実行手順 (ex3_rv32/rv_test/)

- **Makefile** : Cプログラムコンパイル手順や実行手順を記述したファイル
 - SRC = xxx : 第1ソースファイル名(拡張子なし) → 出力ファイル名でも使用
 - SRC_C = \$(SRC).c : Cソースファイル名 → 複数のCソースファイルがある場合は、SRC_Cにファイル名(拡張子付き)を追加する
- RV32コンパイラ : riscv32-unknown-elf-gcc
- ホストPC用コンパイラ : gcc (プログラム動作確認用)
- RV32シミュレータ : ex3_rv32/bin/release/rv32.exe
- 使用手順 : ex3_rv32/rv_test/に移動、makeを実行 → コンパイルとシミュレーションが実行される

(SRC = test_rv32_1 の場合)

ソースファイル名	説明
test_rv32_1.c	第1ソースファイル
entry.S (アセンブリプログラム)	初期化処理 → main呼出し → 終了処理
printf.c	printf関数(簡易的な実装)
io.h, io.c	IOデバイス定義、IO処理、割込み処理
mhandler.S (アセンブリプログラム)	割込みハンドラー
出力ファイル名	説明
test_rv32_1.out (バイナリ)	RV32 ELF形式実行ファイル
test_rv32_1.bin (バイナリ)	FPGA用プログラムファイル
test_rv32_1.dump (テキスト)	アセンブリ形式ダンプファイル
test_rv32_1.log (テキスト)	RV32シミュレーションログ

Makefile (ex3_rv32/rv_test/)

```
SRC = test_rv32_1
SRC_C = $(SRC).c

DIR_CC=~/tools/rv32ia/bin
CC=$(DIR_CC)/riscv32-unknown-elf-gcc
LD=$(DIR_CC)/riscv32-unknown-elf-ld
DUMP=$(DIR_CC)/riscv32-unknown-elf-objdump
CFLAGS=-O1

CFLAGS_RV=-march=rv32iazicsr
#CFLAGS_RV+= -DTEST_WFI_BREAK
OUT= temp

OUT2= $(SRC)

MEM_START=0x00001000
DATA_START=0x00020000

LDFLAGS= -I. -nostartfiles -static -Wl,--defsym=MEM_START=$(MEM_START),--defsym=DATA_START=$(DATA_START),-T,vcu108.lds

RV_SRC= entry.S printf.c io.c io.h mhandler.S ../rv32_io_setting.h

%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<
%.o: %.cpp
    $(CC) $(CFLAGS) -c -o $@ $<

all: $(OUT).out $(OUT).dump a.out
    cp $(OUT).out $(OUT2).out
    cp $(OUT).dump $(OUT2).dump
    ./bin/release/rv32.exe $(OUT2).out
    ./a.out
    rm $(OUT).out
    rm $(OUT).dump

$(OUT).out: $(SRC_C) $(RV_SRC)
    $(CC) -o $@ $^ $(LDFLAGS) $(CFLAGS) $(CFLAGS_RV)

$(OUT).dump: $(OUT).out
    $(DUMP) -D $^ > $@

a.out: $(SRC_C)
    gcc $(CFLAGS) -DNATIVE_MODE $(SRC_C)

clean:
    rm -rf $(OUT).out *.o $(OUT).dump
```

SRC : 第1ソースファイル名(拡張子なし)
SRC_C : Cソースファイル名 → 複数のCソースファイルがある場合は、SRC_Cにファイル名(拡張子付き)を追加する
例) SRC_C = \$(SRC).c test2.c test3.c

"#"を外すと、#define TEST_WFI_BREAKが追加される

MEM_START : プログラムメモリの最小アドレス(0x1000)
DATA_START : データメモリの最小アドレス(0x20000)
メモリサイズ : $2^{18} = 256\text{K Byte}$ (0x40000)

RV_SRC = entry.S printf.c io.c io.h mhandler.S ../rv32_io_setting.h

他のRV32用ソースファイル

- 出力ファイルのコピー
- RV32シミュレーション(.../rv32.exe)
- ホストPCでプログラム実行(a.out)

RV32実行プログラムのビルド

ホストPC実行プログラムのビルド
(-DNATIVE_MODE → #define NATIVE_MODE
マクロを追加してビルド)

サンプルCプログラム概要 (ex3_rv32/rv_test/)

ソースファイル名	説明
test_rv32_1.c	配列の二乗和計算(GPIO出力、printf出力)
test_rv32_2.c	1系統Teraterm/UART送受信(ポーリング方式)
test_rv32_3.c	2系統Teraterm/UART送受信(ポーリング方式)
test_rv32_4.c	2系統Teraterm/UART送受信(割込み方式)
test_rv32_5.c	タイマー割込み処理

SRC = test_rv32_1.c

```
#include <stdio.h>
#include "io.h"

int d[6] = { 2, 3, 5, 7, 11, 13 };
int main(){
    #if !defined(NATIVE_MODE)
        gpio->data = 0;
    #endif
    start_profiler();
    int sum = 0, i;
    for (i = 0; i < 6; i++) {
        int a = d[i];
        WFI_BREAK; // デバッグ用break命令
        sum += a * a;
    }
    // end_profiler();
    #if !defined(NATIVE_MODE)
        gpio->data = sum;
    #endif
    printf("sum = %d (0x%08X)\n", sum, sum);
    end_profiler();
    return 0;
}
```

NATIVE_MODE : ホストPC用マクロ
→ GPIOへの書き込みはRV32専用

WFI_BREAK : FPGAデバッグ用break命令(後述)

UART初期化・起動

GPIO出力

UART出力

make実行結果

```
>> make
..... <中略>.....
./bin/release/rv32.exe test_rv32_1.out <-- RV32シミュレーション実行
..... <中略>.....
[ 0] <UART_EXT-0> (activate) : baud_period(1302), bit_count(8), sto
[ 0] <UART_EXT-1> (activate) : baud_period(1302), bit_count(8), stop_count(1), parity_type(0)
[10289] <UART_AXI-0> (activate) : baud_period(1302), bit_count(8), stop_count(1), parity_type(0)
[10299] <UART_AXI-1> (activate) : baud_period(1302), bit_count(8), stop_count(1), parity_type(0)
[10377] gpio.dout[0] = (00000000)
[10954] gpio.dout[0] = (00000179) <-- GPIO出力
[244197] <UART_X-0.RX> sum = 377 (0x179) <-- UART出力
dc.error = 0, mem.error = 0, wfi = 1
minstret = 180242, mcycle = 870535
[870536] <UART_X-0.RX> PROFILE: 183152 cycles, 37283 insts, 3662 usecs
timer(RV32I_ARCH1): 0.725 sec(elapsed), 0.725 sec(total), 1,201,068.706 cycles/sec, 870,537 cycles
```

entry.S (ex3_rv32/rv_test/)

```
.section .text.init
.globl _start
_start: /// アプリケーションプログラムの最初の命令アドレス
    la sp, stack_bot /// initialize sp
    la t0, __bss_start
    la t1, __end
    beq t0, t1, 2f /// Label "2:" (forward location)
1:
    sw zero, 0(t0)
    addi t0, t0, 4
    bne t0, t1, 1b /// Label "1:" (backward location)
2:
    jal disable_UART_interrupt /// UART割込み無効化
    jal __io_init             /// UARTポート初期化・起動
    jal enable_BUTTON_interrupt /// ボタン割込み有効化
    jal main                  /// main関数呼出し
    jal __io_fini              /// UARTポート終了
    wfi           /// WaitForInterrupt : 割込み検知待機
    j _start      /// _start へ戻る
.section .stack,"aw",@nobits
stack_top:
    .skip 0x2000
stack_bot:
```

データメモリの0初期化
セクションの初期化処理

entry.Sの実行の流れ

1. `_start`: 最初の命令アドレス
2. `sp`(スタックポインタ)の初期化
3. データメモリの0初期化セクションの初期化処理
4. UART割込み無効化
5. UARTポート初期化・起動
6. ボタン割込み有効化
7. `main`関数呼出し
8. UARTポート終了処理(送信バッファが空になるまで待機)
9. WFI : 割込み検知待機 → ボタン割込みで解除
10. `_start`へ戻る

実行プログラムのアセンブリ表示

test_rv32_1.dump

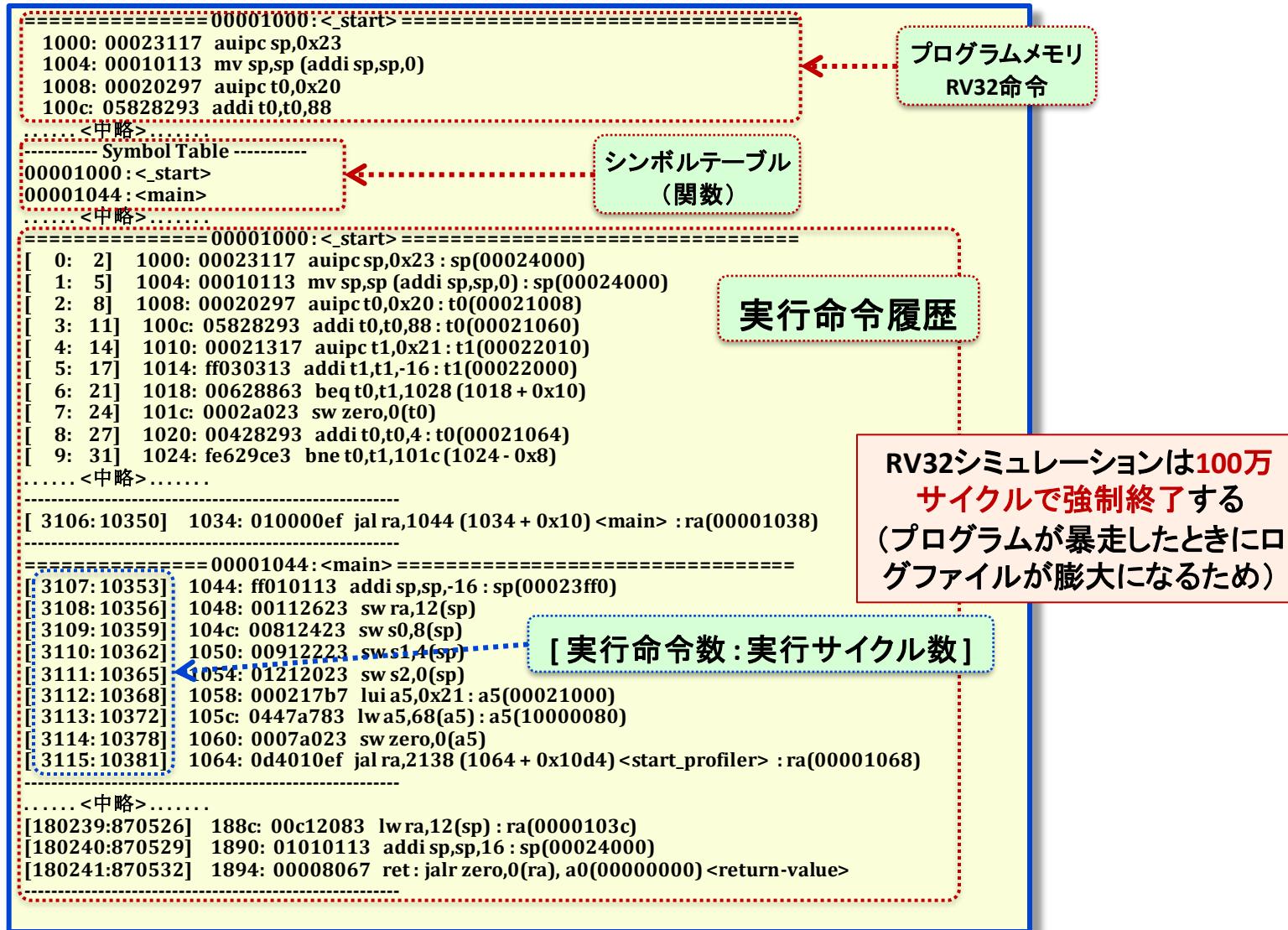
```
00001000 <_start>:
1000: 00023117 auipc    sp,0x23
1004: 00010113 mv       sp,sp
1008: 00020297 auipc    t0,0x20
100c: 05828293 addi     t0,t0,88 # 21060 <UART_BUF_TX>
1010: 00021317 auipc    t1,0x21
1014: ff030313 addi     t1,t1,-16 # 22000 <_end>
1018: 00628863 beq      t0,t1,1028 <_start+0x28>
101c: 0002a023 sw       zero,0(t0)
1020: 00428293 addi     t0,t0,4
1024: fe629ce3 bne      t0,t1,101c <_start+0x1c>
1028: 151000ef jal      1978 <disable_UART_interrupt>
102c: 6d0000ef jal      16fc <_io_init>
1030: 0ad000ef jal      18dc <enable_BUTTON_interrupt>
1034: 010000ef jal      1044 <main>
1038: 049000ef jal      1880 <_io_fini>
103c: 10500073 wfi
1040: fc1ff06f j        1000 <_start>

00001044 <main>:
1044: ff010113 addi     sp,sp,-16 # 23ff0 <_end+0x1ff0>
1048: 00112623 sw       ra,12(sp)
104c: 00812423 sw       s0,8(sp)
1050: 00912223 sw       s1,4(sp)
1054: 01212023 sw       s2,0(sp)
1058: 000217b7 lui      a5,0x21
105c: 0447a783 lw       a5,68(a5) # 21044 <gpio>
1060: 0007a023 sw       zero,0(a5)
1064: 0d4010ef jal      2138 <start_profiler>
.....<中略>.....
```

entry.S

```
.section .text.init
.globl _start
_start:
    la sp, stack_bot
    la t0, __bss_start
    la t1, __end
    beq t0, t1, 2f
1:
    sw zero, 0(t0)
    addi t0, t0, 4
    bne t0, t1, 1b
2:
    jal disable_UART_interrupt
    jal _io_init
    jal enable_BUTTON_interrupt
    jal main
    jal _io_fini
    wfi
    j _start
.section .stack,"aw",@nobits
stack_top:
    .skip 0x2000
stack_bot:
```

RV32シミュレーションログ (test_rv32_1.1.log)



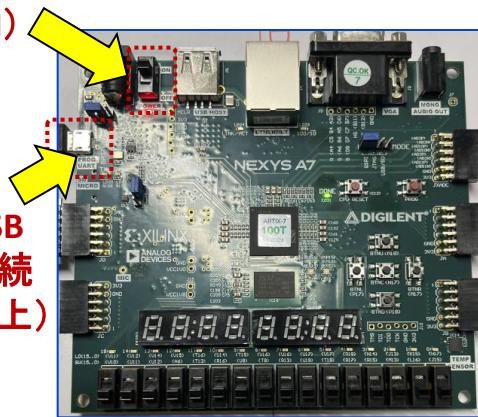
資料概要

1. RISC-Vプロセッサとシステム概要
2. RV32-I命令セット仕様
3. ABI : エンディアン、アラインメント、関数呼出し規則
4. 入出力装置と割込み処理機構
5. RV32プログラミング手順 : WSL(Ubuntu)
6. FPGA使用手順 : Windows-11
7. RV32プログラミング課題

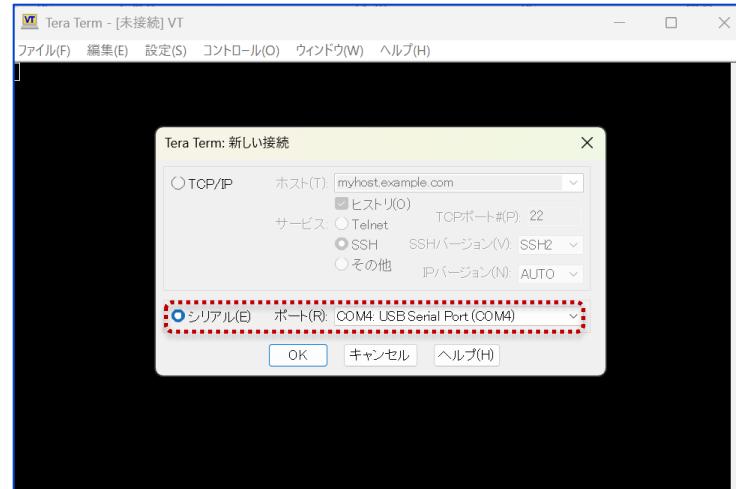
FPGA使用手順(1)：準備

1. **FPGAボード・ホストPC(Windows11)の接続** → Nexys FPGA付属のUSB-Micro USBケーブル(平らの面が上)をボードに挿入し、**電源スイッチをONにする**
2. **Teraterm起動(Version 5.3であることを確認)**:
 - a. 「新しい接続」画面で、「シリアル」をクリック、ポート「**COM4 USB Serial Port**」を確認
 - b. 「設定」→「設定の読み込み」→エクスプローラー画面で、**ex3_2025/FPGA/TERATERM.INI**を選択

1. 電源スイッチ
(上側がON)



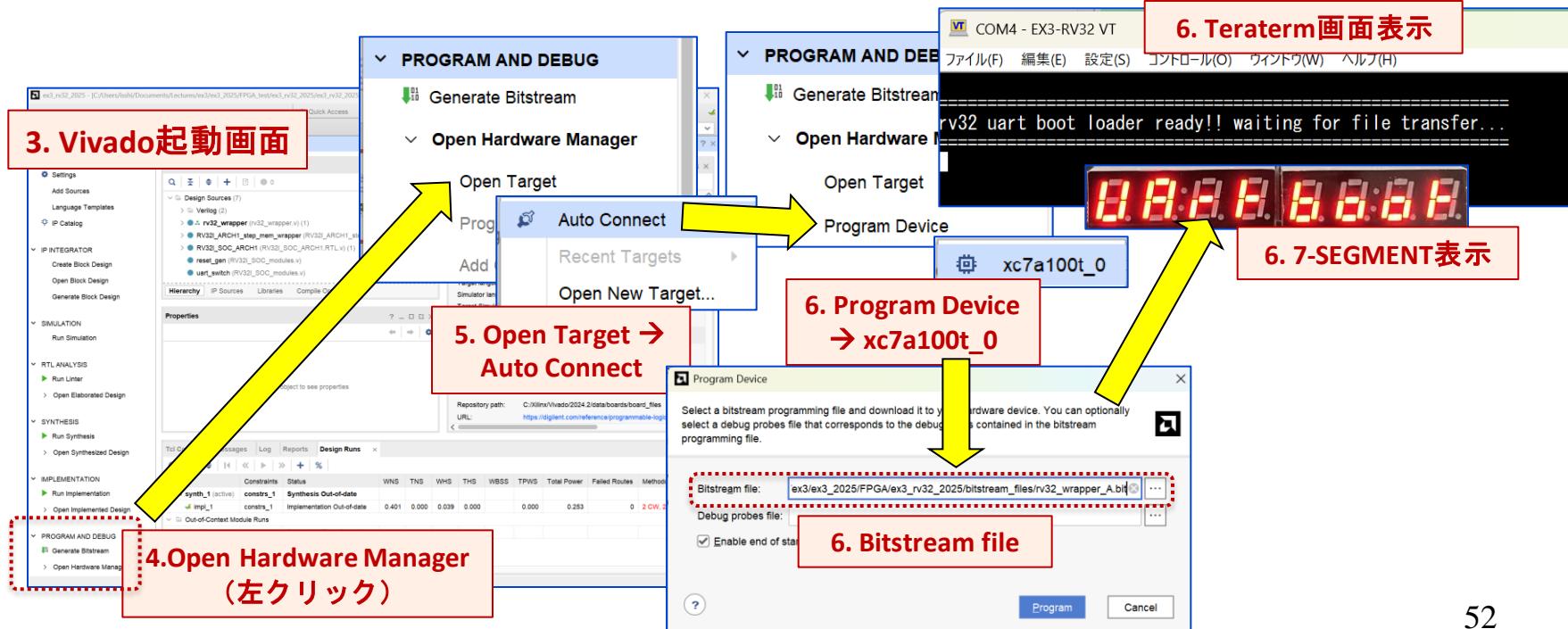
1. Micro-USB
ケーブル接続
(平らの面が上)



2. Teraterm起動画面

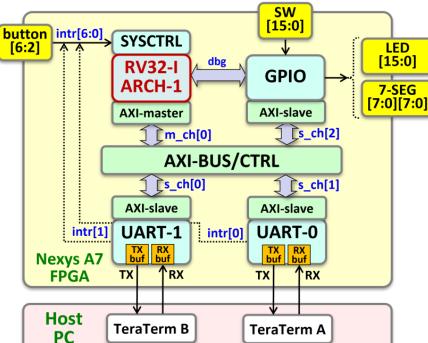
FPGA使用手順(2) : FPGA bitstream転送

3. `ex3_2025/FPGA/ex3_rv32_2025/ex3_rv32_2025.xpr`をダブルクリック → Vivadoが起動
 4. Open Hardware Manager (左下)を左クリック
 5. Open Target (左クリック) → Auto Connect (左クリック)
 6. Program Device (左クリック) → xc7a100t_0 (左クリック) → Bitstream file : `ex3_2025/FPGA/bitstream_files/rv32_wrapper_A.bit` を指定
- 転送終了後、Teratermに下図の表示され、FPGAボードの7 SEGMENTに”UART BOOT”が表示される

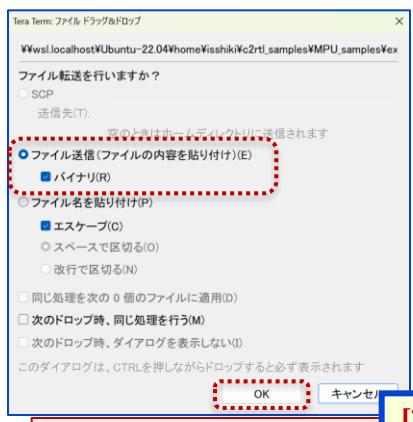


FPGA使用手順(3) : RV32プログラム転送

- FPGAボードには、下図のRV32-Iシステムがダウンロードされ、Teratermのファイル転送機能でRV32プログラムを書き込む「boot loaderプログラム」が動いている
7. Linuxサーバー上のex3_rv32/rv_test/で作成したRV32用バイナリファイル(例: test_rv32_1.bin)をWindows PC上にコピーする。
 8. RV32バイナリのファイル転送: Windowsエクスプローラーで、RV32バイナリファイルをTeratermにドラッグ & ドロップ → 「ファイル送信」、「バイナリ」が選択されていることを確認し、OKをクリックする。
 9. RV32プログラム起動: ファイル転送終了後、下図のファイル転送終了画面で、RV32プログラムの最初の64命令のHEX表示に続いて、「press any key to run your program...」が表示される。enterキーなどを入力するとファイル転送したRV32プログラムが実行する。



6. RV32-Iシステム



8. Teratermファイル
ドラッグ & ドロップ画面



9. ファイル転送終了画面

```
uart boot loader transfer succeeded!! press any key to run your program...
sum = 377 (0x179)
PROFILE : 183152 cycles, 37283 insts, 3662 usecs
```

RV32プログラムの
HEX表示

9. プログラム実行結果

[244197] <UART_X-0.RX> sum = 377 (0x179)
[870536] <UART_X-0.RX> PROFILE : 183152 cycles, 37283 insts, 3662 usecs

RV32シミュレーション結果 (P45参照)

FPGA使用手順(4) : WFIデバッグ手順

- RV32プログラムのmain関数実行終了後、0x103c番地のWFI(Wait For Interrupt)命令で停止する。
- WFI命令停止状態では、SW[5:0]で指定したRV32の内部レジスタの値が7 SEGMENTに表示される。
- WFI命令停止状態は、ボタンを押すことで解除され、SW[15:12]の設定でデバッグが出来る。

test_rv32_1.dump			
00001000 <_start>:	1000: 00023117	auipc sp,0x23	
.....<中略>.....	1034: 010000ef	jal 1044 <main>	
	1038: 049000ef	jal 1880 <_io_fini>	
	103c: 10500073	wfi	
	1040: fc1ff06f	j 1000 <_start>	
.....<中略>.....			

WFI停止中のデバッグモード	
SW[15:12]	ボタン動作
0000	RUN(WFI停止解除)
0001	STEP CYCLE(1サイクル後停止)
0010	STEP IN** (1命令後停止)
0011	STEP OVER** (関数呼出し後停止)
0100	STEP OUT** (現関数リターン後停止)
1000	IGNORE WFI(WFI停止を無効化)

** STEP IN/OVER/OUT : gdbの操作に準拠

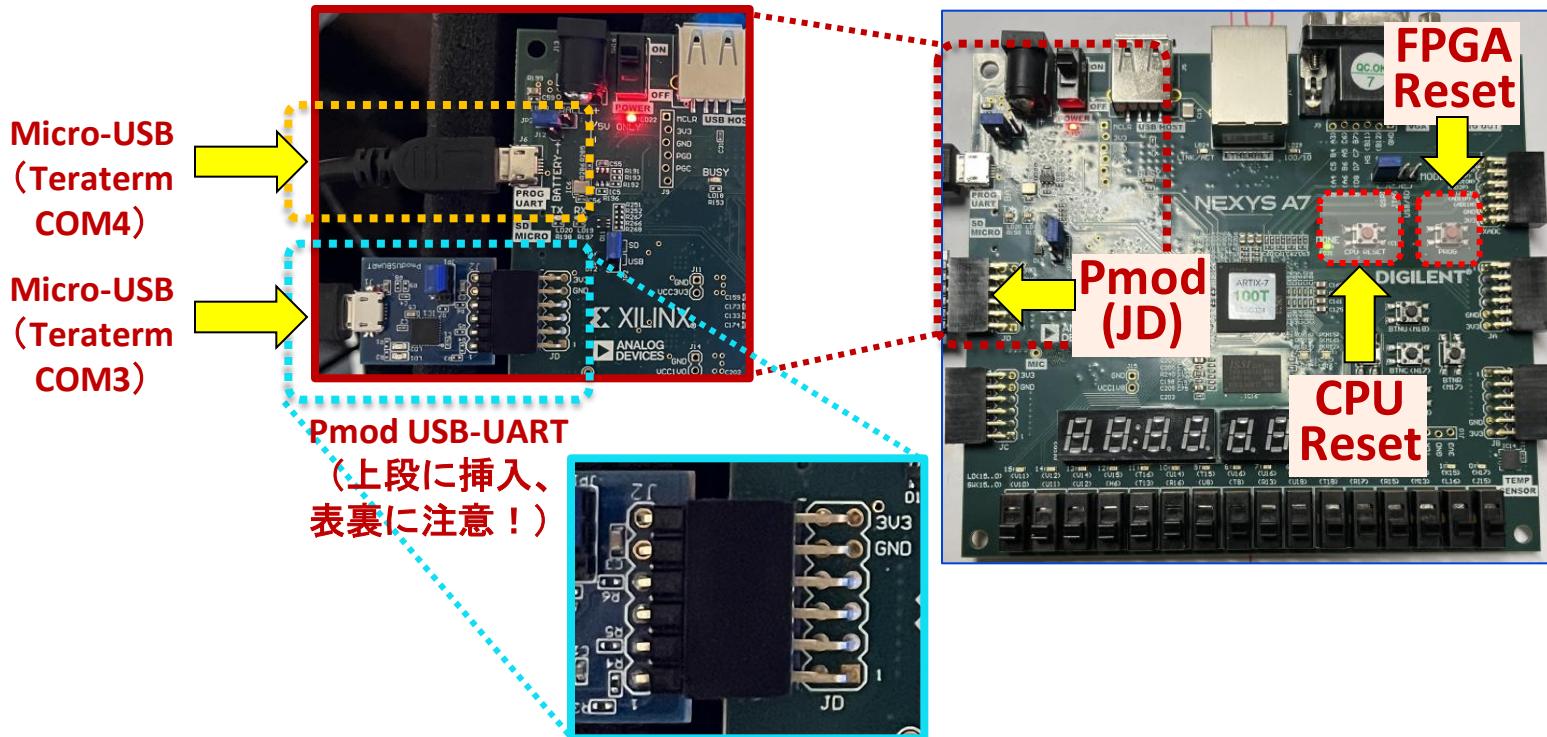
WFI停止中の表示モード			
SW[5:0]	7 SEGMENT出力	SW[5:0]	7 SEGMENT出力
000000	PC*(プログラムカウンタ)	100111	mcycle
000001～ 011111	xreg[1]～xreg[31]	101000	mcycleh
100000	IR(命令レジスタ)	101001	minstret
100001	mstatus	101010	minstreth
100010	mie	110000	sysctrl->ext_irq_enable
100011	mtvec	110001	sysctrl->mtime[31:0]
100100	mepc	110010	sysctrl->mtime[63:32]
100101	mcause	110011	sysctrl->mtimecmp[31:0]
100110	mip	110100	sysctrl->mtimecmp[63:32]

*PCの値は、命令実行状態によっては、次命令アドレス値を示す
命令実行状態(0, 1, 2, 3) : LED[3:0]で表示されている

WFI停止中のデバッグモードで動作の不具合を見つけた場合は、TAに報告してください。
今後の実験の改善にご協力ください。

FPGA使用手順(5)：その他

10. **2個目のMicro-USB接続**: Micro-USBをPmod USB-UARTモジュールに接続し、Pmod USB-UARTモジュールをPmodコネクタ(JD)の上段に挿入する。
11. **2個目のTeraterm起動**: ポート「COM3 USB Serial Port」を選択
12. **CPU Resetボタン**: boot loaderプログラムが再起動し、RV32プログラム転送待機状態に戻る
13. **FPGA Resetボタン**: FPGAボードが電源投入後の初期状態に戻る



資料概要

1. RISC-Vプロセッサとシステム概要
2. RV32-I命令セット仕様
3. ABI : エンディアン、アラインメント、関数呼出し規則
4. 入出力装置と割込み処理機構
5. RV32プログラミング手順 : WSL(Ubuntu)
6. FPGA使用手順 : Windows-11
7. RV32プログラミング課題

第1回 レポート課題

課題プログラム	説明
1. 10進数 → 16進数変換	Teratermの数字キー入力の10進数を16進数で7 SEGMENTに表示する。
2. 割り算・剰余算	アセンブリ語出力 (*.dump)を解析し、内部関数呼出しや条件分岐、ループ処理などをフローチャートで示し、計算アルゴリズムを分り易く説明する。入力データはTeratermの数字キー入力の10進数で設定する。
3. 電卓プログラム	Teratermのキー入力に対して電卓機能を実装する。キー入力の状態や、計算結果は7 SEGMENT表示に10進数で出力する。実装する電卓機能仕様は各自に任せる。
4. 素数計算	整数Nについて、N以下の最大の素数を計算(Nをどこまで大きくできるか工夫すること)
5. 7 SEGMENTタイマー制御プログラム	test_rv32_5.cのタイマー割込みプログラムを参考に 、7 SEGMENT表示を周期的に変化させる(文字列や任意のパターン)。FPGA電源投入後の7 SEGMENTのパターン表示動作も参考に。
6. チャットプログラム	test_rv32_4.cを参考に 、一方のTeratermのキー入力に対し、そのecho応答を他方のTeratermに出力するプログラムを作成する。2つのFPGAボードをpmodコネクタケーブルで接続し、チャットプログラムの動作を確認する。 補足：接続する2つのFPGAボードのbitstreamは、一つは rv32_wrapper_A.bit を使用し、もう一つは rv32_wrapper_B.bit を使用する(Pmod信号ピンの入出力を交換するため)