Ver. 2024-10-21

コンピュータアーキテクチャ 演習 (3)
Computer Architecture Exercise (3)

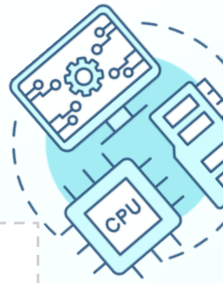
情報工学系　Berjab Nesrine

Computer Architecture support page :

https://www.arch.cs.titech.ac.jp/lecture/CA/

2024年度（令和6年）版

コンピュータアーキテクチャ 演習の注意点 (1)



❑ 連絡について

- 連絡は Slack を使用する。登録がまだの場合は速やかに行うこと。招待メールが来ていない場合は、教員あるいはTAにmアドレスを伝え再送要求すること。

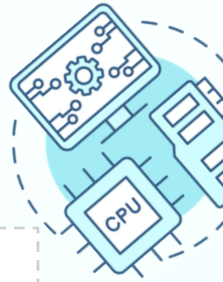
❑ 演習について

- 演習は 15:25～17:05 の時間で行う。15:20 までに学術国際情報センター 3階、情報工学系計算機室に集合すること。15:45 までに到着しない場合、欠席扱いになる。
- 最初の15分は課題の説明、その後は課題の進行とチェックポイントの確認を行う。演習ではACRi ルームを利用する。

❑ グループ作業

- 3人のグループを作成し、グループ内で情報を共有しながら演習を進める。問題が発生した場合、まずグループ内で相談し、それでも解決しない場合は TA や教員に質問すること。

コンピュータアーキテクチャ 演習の注意点 (2)

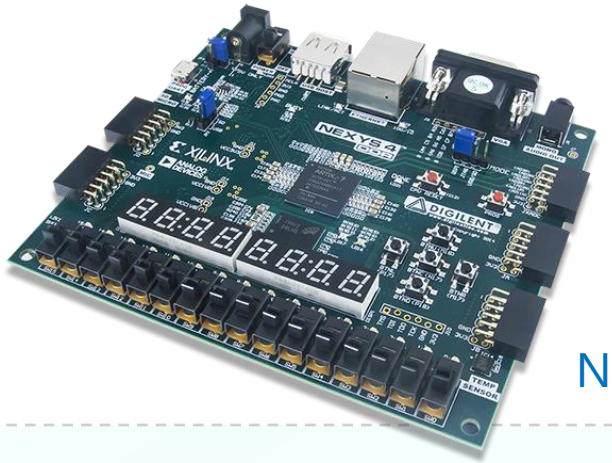


❑ 出席について

- 演習には出席点があるため、全ての授業に休まず参加すること。チェックポイントの図が演習スライドに示されている箇所で、作業の確認を受ける。全てのチェックポイントをクリアすることを目指す。

❑ 演習時間外について

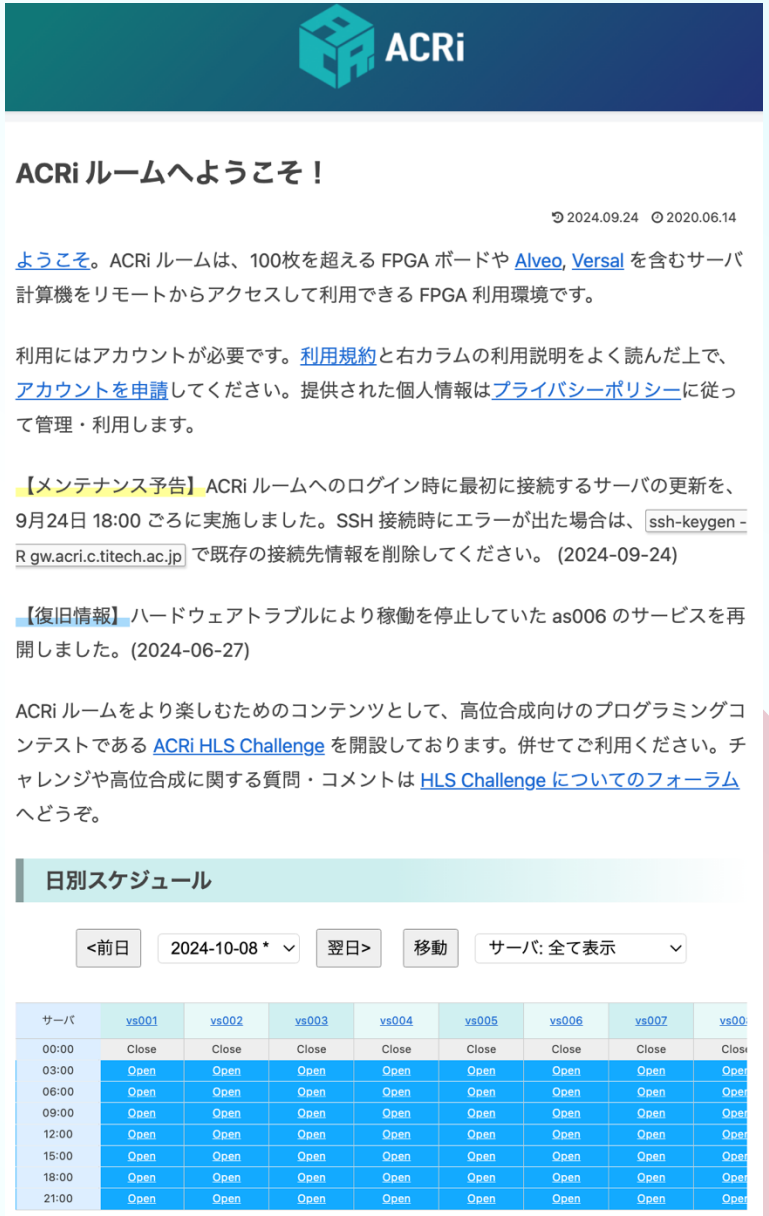
- 演習時間外にも手元の FPGA ボードや ACRi ルームを利用できる。
- 手元のFPGA ボードの貸出も可能なので、独自のハードウェア設計に挑戦してみよう！

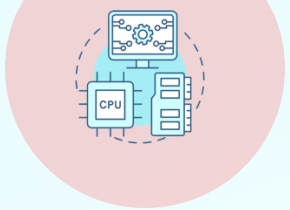


Nexys4 DDR Artix-7 FPGAボード

【重要】 ACRiルームのサーバの予約

- ACRi ルームのアカウントを使って，次の URL からログインする。
  https://gw.acri.c.titech.ac.jp/wp
- アカウントがなければ、今、次の URL のページを参考にアカウントを申請すること。
  https://gw.acri.c.titech.ac.jp/wp/manual/apply-for-account
  - ACRi ルームでは、次の2種類のアカウントを使用している。
    1. Web 予約システム用アカウント
    2. Linux サーバログイン用アカウント
  - 2つのアカウントで同じ「ユーザアカウント名」を使っているが、別々の「パスワード」が設定されているので注意すること。
- 「予約ページトップ」から，vs0xx~vs7xxで始まるサーバで演習の日の15:00～18:00の枠を予約すること。

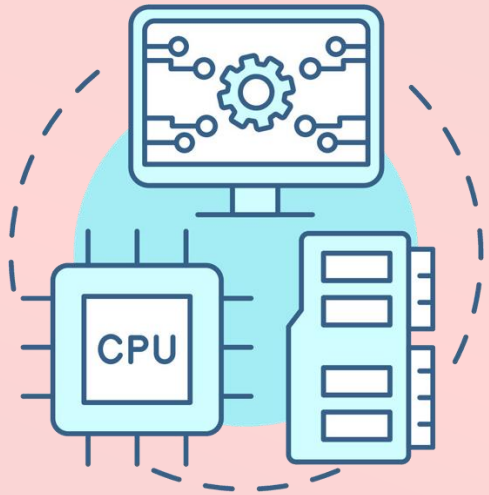




演習第四回の内容 (Project 4)

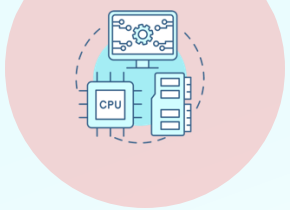
❑ 目的:

- この演習の目的は、シングルサイクルのRISC-Vプロセッサを使って、文字「a」を出力するアセンブリコードを書き、その動作をシミュレーションとFPGAの両方で確認すること。
- この演習は、2つの部分に分かれている。
  - **Part 1**: アセンブリ言語プログラミング + シミュレーション。
    1. 「a」という文字を出力するアセンブリコードを記述する。
    2. Venus シミュレータを使って、アセンブリコードを機械語に変換する。
    3. 機械語をシミュレーションで実行し、「a」という文字が出力されるか確認する。
  - **Part 2**: FPGA
    1. 機械語に変換されたコードを FPGA 上で実行する。
    2. 「a」 という文字が正しく出力されるかを確認する。



Project 4

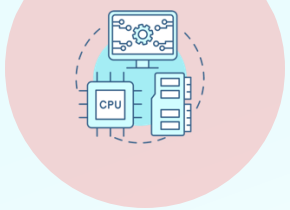
(Part 1)



演習第四回の内容 (Project 4-Part 1)

❑ 目的:

- この演習の目的は、シングルサイクルのRISC-Vプロセッサを使って、文字「a」を出力するアセンブリコードを書き、その動作をシミュレーションで確認すること。
- Steps:
    1. 「a」という文字を出力するアセンブリコードを記述する。
    2. Venus シミュレータを使って、アセンブリコードを機械語に変換する。
    3. 機械語をメモリに書き込む。
    4. 機械語をシミュレーションで実行し、 「a」という文字が出力されるか確認する。

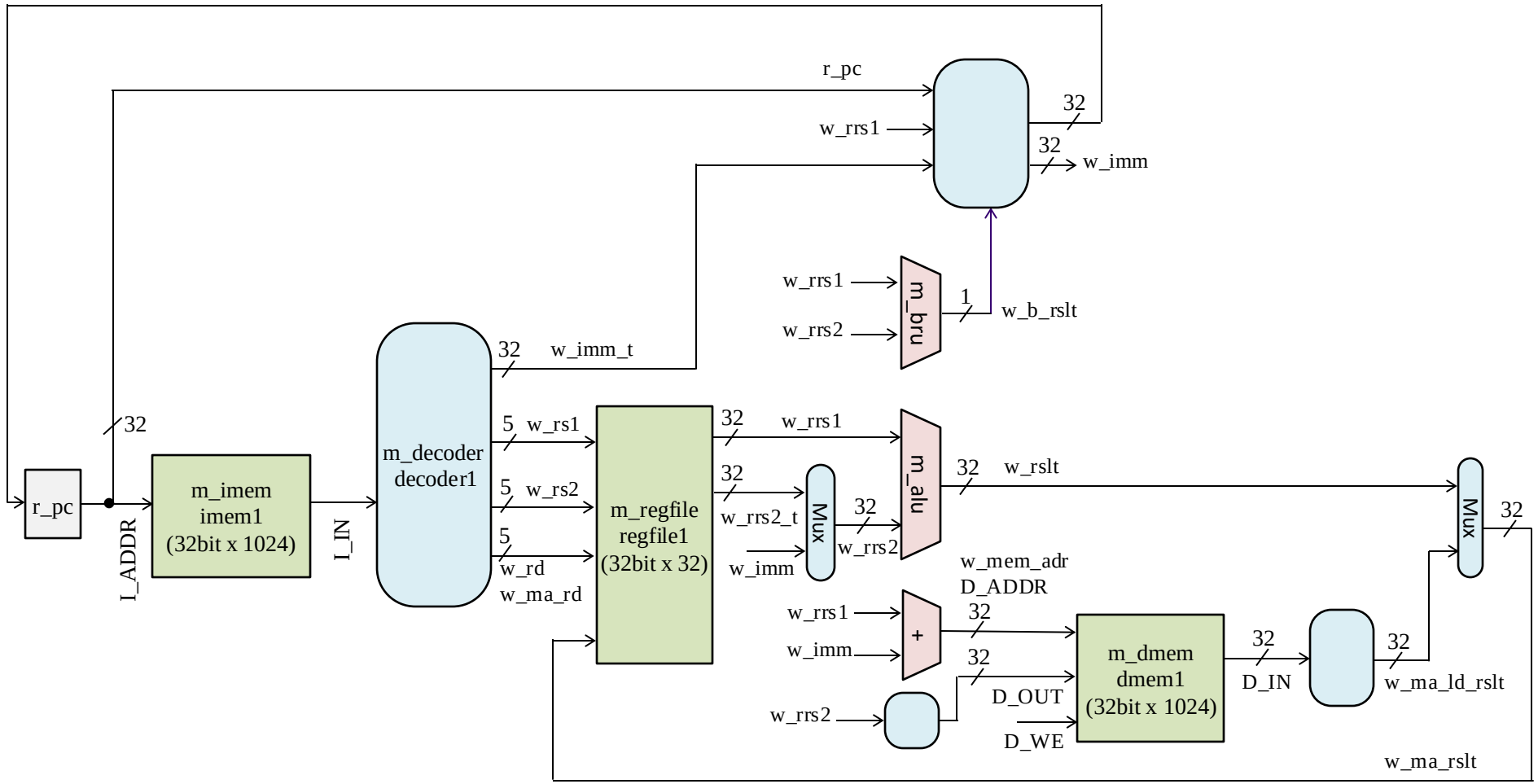


RVCore シングルサイクルのRISC-Vプロセッサ

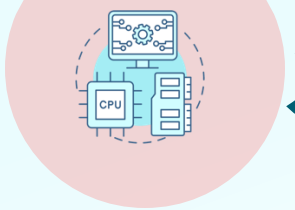
❑ この演習で使用する RISC-V シングルサイクルプロセッサの主な特徴

- 単一クロックサイクルでの命令実行。
- 32ビットの RISC-V 基本整数命令セットである RV32I をサポート。
- 4KiB の命令/データメモリ。
- UART を使って文字を出力するためのメモリマップド I/O。
- ALU (Arithmetic Logic Unit): 加減算、ビット演算 (AND, OR, XOR)、シフト演算をサポート。
- BRU (Branch Resolution Unit): 条件分岐 (branch) 命令、無条件分岐 (jump) 命令をサポート。
- Load/Store: バイト (8-bit)、ハーフワード (16-bit)、ワード (32-bit) 単位でのメモリのロード/ストア命令をサポート。
- レジスタファイル: 32個の32ビットレジスタを搭載し、立ち上がりエッジで書き込み。
- 即値生成: I, S, B, U, J 形式の命令の即値を生成。

module m_rvcore in proc1.v (RV32I, single-cycle processor)



(The source code of proc1.v is available in /home/u_nesrine/ca2024/src)



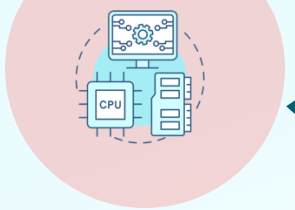
ステップ1: 文字「a」を出力するアセンブリコード (1/2)

❑ 目的:

- シングルサイクルのRISC-Vプロセッサを使って、文字「a」を出力するアセンブリコードを書いてもらう。プロセッサはメモリマップされたI/Oデバイス（UART）とやり取りし、「a」を表示する。
- ヒント:
  - プロセッサは UART デバイスがメモリマップドされているアドレス (0x40008000) にデータを書き込むことで、文字を出力したり、シミュレーションを終了したりすることができる。
  - こちらの図に示されているように、 UART デバイスに書き込むデータのフォーマットは「コマンド (CMD)」と「文字データ (CHAR)」から成る。

```
tohost_data format
 31             17 16          7       0
+-------------+-----+--------+--------+
|             | CMD |        |  CHAR  |
+-------------+-----+--------+--------+
```

  - CMD==1 は「文字出力コマンド」であり、CHAR にセットされている ASCII 値を出力する。
  - CMD==2 は「終了コマンド」であり、シミュレーションを終了する。 (CHAR は無視される)
  - よって、文字「a」 を出力するには、UART アドレス (0x40008000) に「文字出力コマンド」と文字「a」の ASCII 値 0x61 を組み合わせた 0x00010061 をストアすればよい。
  - 文字を出力後は「終了コマンド」(0x00020000) を UART アドレスにストアしてから、それ以上命令を実行しないように無限ループに入るようにする。



ステップ1: 文字「a」を出力するアセンブリコード (2/2)

❑ Pseudo code:

1. UART アドレスをレジスタにセットする。
   - reg0 = 0x40008000 (UART デバイスのアドレス)
2. UART アドレスに送る「文字出力コマンド」と「文字データ」を別のレジスタにセットする。
   - reg1 = 0x00010061
3. UART アドレスにデータをストアする。
   - mem[reg0] = reg1
4. UART アドレスに「終了コマンド」をレジスタにセットする。
   - reg2 = 0x00020000
5. UART アドレスにデータをストアする。
   - mem[reg0] = reg2
6. プログラムをそれ以上実行しないように無限ループに入る。

ステップ2: Venus シミュレーター を使用

❑ 目的:

- Venus シミュレーターを使って、 記述したアセンブリコードを機械語に変換する。

https://venus.cs61c.org

- 左側のタブ Editor を使って、RISC-V アセンブリコードを書く。
- 次はタブ Simulator を使って、 Assemble & Simulate ボタンをクリックして、シミュレーターでコードを実行する。
- Dumpタブを使って、コード中のすべての命令の16進数表現を生成する。
- その後、Download を押すと、出力された機械語をテキストファイルにコピーする。
  - ファイル名（例: sample1.txt）を入力し保存する。

ステップ3: 機械語をメモリに書き込む

❑ 目的:

- 機械語をプロセッサが実行できるよう、 メモリを初期化する。
- Venusシミュレーターからダウンロードした機械語を手動でメモリ情報に変換し、mem[ ] 配列に書き込む必要がある。
- 各命令を正しいメモリアドレスに割り当て、プロセッサがそれを実行できるように設定する。
- 結果: このプロセスにより、Venus から得た機械語が正しくメモリにロードされ、シミュレーションまたはFPGAで実行できるようになる。

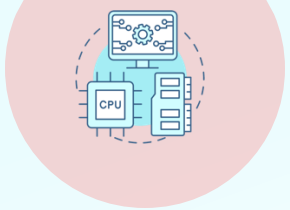
  - 例: 0X 00000013 という機械語を mem[0] = 32'h 00000013; に変換して、メモリのアドレス0に格納する。
    - 0X 00000013 はRISC-V命令の16進数表現。
    - mem[0] = 32'h 00000013; は、メモリの特定のアドレス(この場合は0番目)に機械語を格納するためのコード。

sample1.txt

```
1 0x00000013
2 0x00000033
3 0x000000b3
4 0x00000133
5 0x000001b3
```

sample1.txt

```
1 mem[   0] = 32'h00000013;
2 mem[   1] = 32'h00000033;
3 mem[   2] = 32'h000000b3;
4 mem[   3] = 32'h00000133;
5 mem[   4] = 32'h000001b3;
6 mem[   5] = 32'h00000233;
```



ステップ4: シミュレーション

❑ 目的:

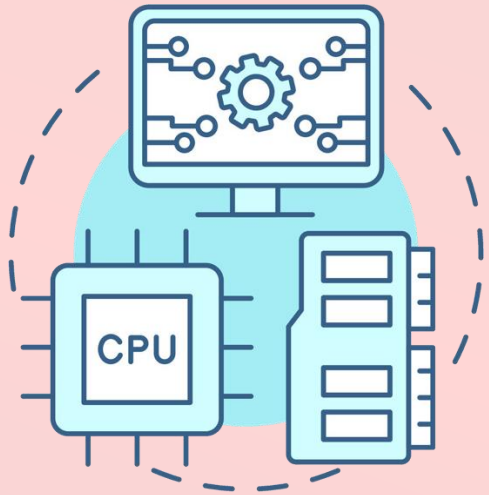
- ➢ 機械語をシミュレーションで実行し、正しく動作を確認する。

```
$ cd ~/ca2024/
$ cp /home/u_nesrine/ca2024/src/proc1.v .
$ /tools/cad/bin/verilator --binary -o simv proc1.v
$./obj_dir/simv
```

「a」 という文字が正しく出力される →

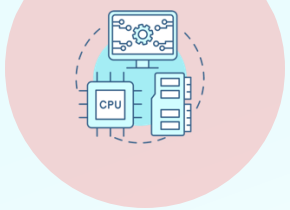
```
== VERIFY is defined and generate verify.txt
a
simulation finished.
- proc1.v:411: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.028 2024-08-21
- Verilator: $finish at 900ps; walltime 0.005 s; speed 973.741 ns/s
- Verilator: cpu 0.001 s on 1 threads; alloced 8 MB
```

Check Point 4



Project 4

(Part 2)



演習第四回の内容 (Project 4-Part ２)

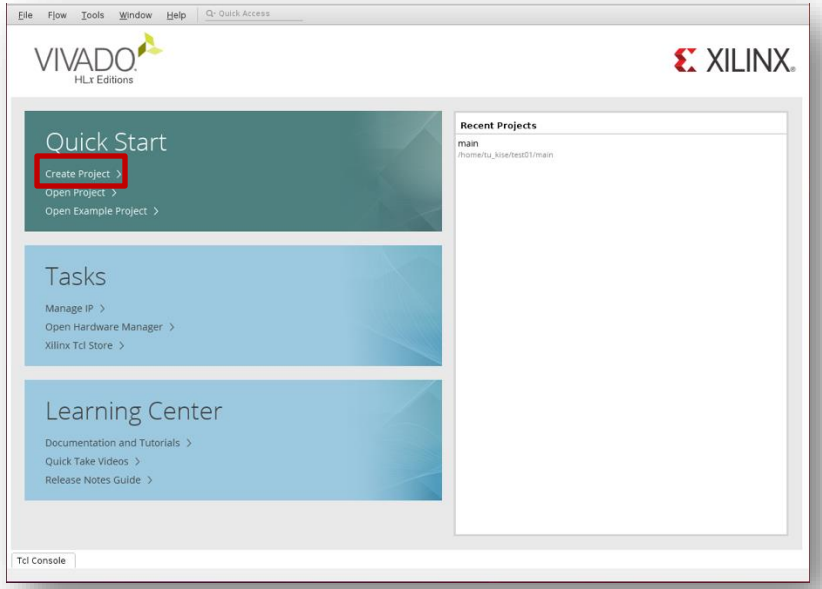
❑ 目的:

- この演習の目的は、シングルサイクルのRISC-Vプロセッサを使って、文字「a」を出力するアセンブリコードを書き、その動作を FPGAで確認すること。
- Steps:
    1. 新しいVivado Project 「project_4」 を作る。
    2. シングルサイクル (main4.v)プロセッサの修正する。
    3. Clocking Wizard でclock を変化させる。
    4. VIOを設定する。
    5. バイナリ形式でメモリの内容 (sample1.txt) が出力する。
    6. UARTによるシリアル通信。
    7. VIO で「a」という文字が出力されるか確認する。

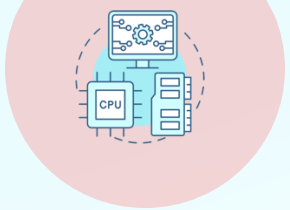
ステップ1:新しい Vivado Project を作る (1/2)

- 新しいVivado Project 「project_4」 を作る。
- ターミナルで次のコマンドを入力し，Vivado を起動する.
  - 「Vivado 2024.1」を利用する。

```
$ source /tools/Xilinx/Vivado/2024.1/settings64.sh
$ vivado &
```



- Select Create Project, Click Next
- Project name “project_4” and location “/home/your_username/ca2024” are selected.
  - Check “Create project subdirectory”.
- Click Next
- In Default Part window, select Parts, and write XC7A35TICSG324-1L.



ステップ1:新しい Vivado Project を作る (2/2)

```
$ cd ~/ca2024/project_4
$ cp /home/u_nesrine/ca2024/src/main4.v .
$ cp /home/u_nesrine/ca2024/src/main4.xdc .
```

- Source codeをコピーする
  - ターミナルで，ファイルをコピーする。
  - /home/u_nesrine/ca2024/src/ に保存されている main4.v と main4.xdc を，作成したプロジェクトのディレクトリ ~/ca2024/project_4 にコピーする。

- Click Add Sources, then select Add or create design sources and click Next.
  - In Add or Create Design Sources window, click Add Files, select main4.v in project_4 directory, and click OK.
  - Click Finish.
- Click Add Sources, then select Add or create constraints and click Next.
  - In Add or Create Constraints window, click Add Files, select main4.xdc in project_4 directory, and click OK.
  - Click Finish.

ステップ2:シングルサイクル (main4.v)プロセッサの修正

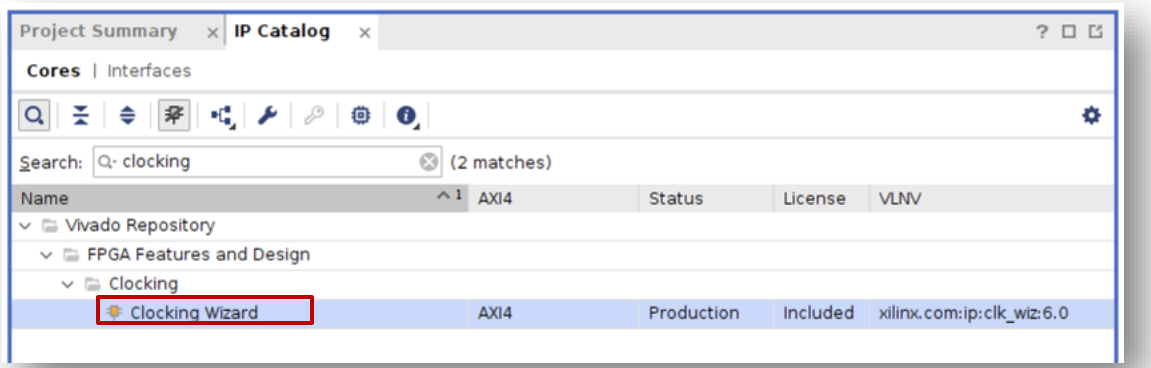
- m_main モジュールを完成させ、「a」の16進数値がVIOを通じて表示されるか確認する。

Main4.v (Source code available in /home/u_nesrine/ca2024/src)

```
module m_main (
    input  wire w_clk,      // 100MHz clock signal
    input  wire w_uart_rx,  // UART rx, data line from PC to FPGA
    output wire w_uart_tx   // UART tx, data line from FPGA to PC
);

    //Code

    // Checkpoint 5: Complete here
    reg [7:0] r_tohost_char;
    reg [1:0] r_vio_tohost_cmd;

    always @(posedge w_clk50m) begin
        //Complete here
    end

    vio_0 vio_inst (
        .clk      (),    // Complete here to connect to the 50 MHz clock
        .probe_in0(),    // Complere here to connect UART character to VIO input
    );

endmodule
```

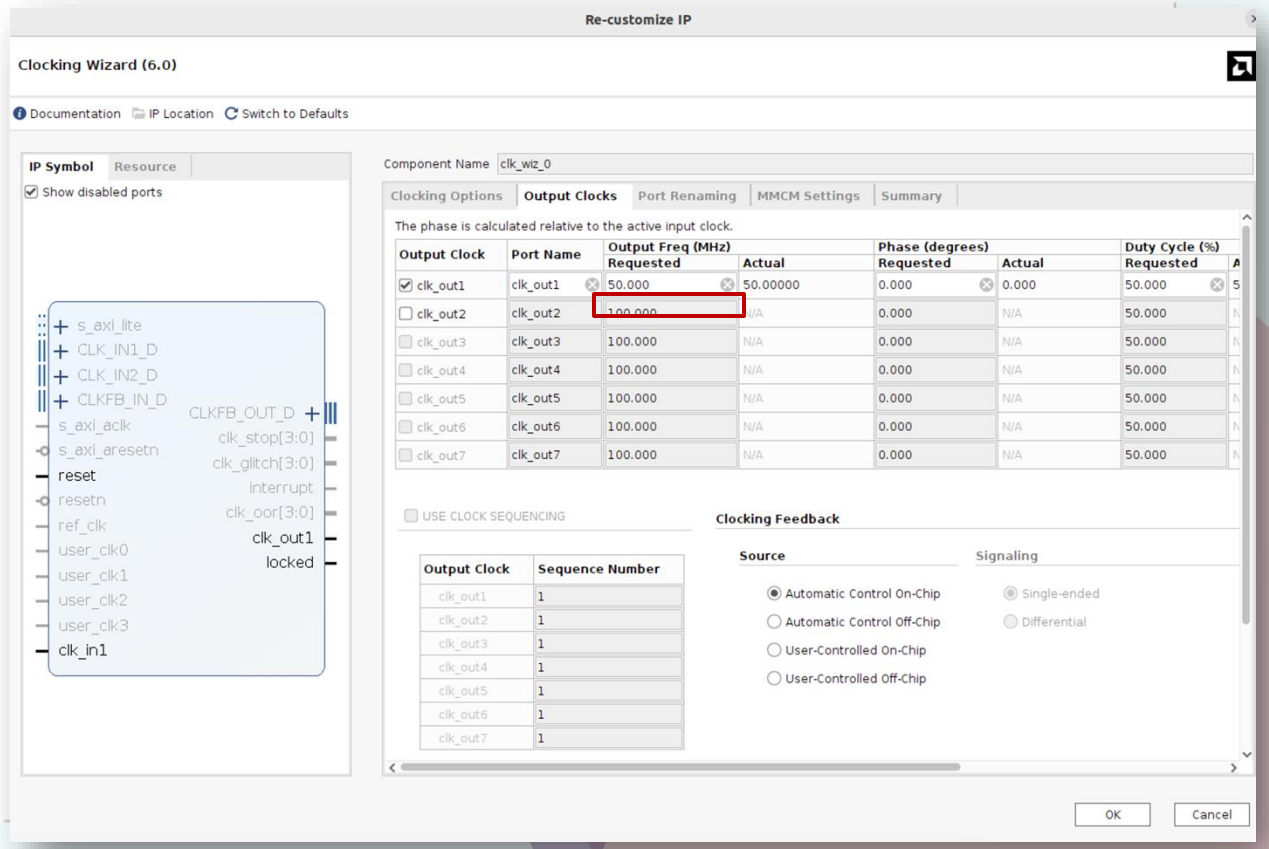
ステップ3: Clocking Wizard でclock を変化させる (1/2)

- ❑ Click IP Catalog
- ❑ Double click Clocking Wizard in IP Catalog window



ステップ3: Clocking Wizard でclock を変化させる (2/2)

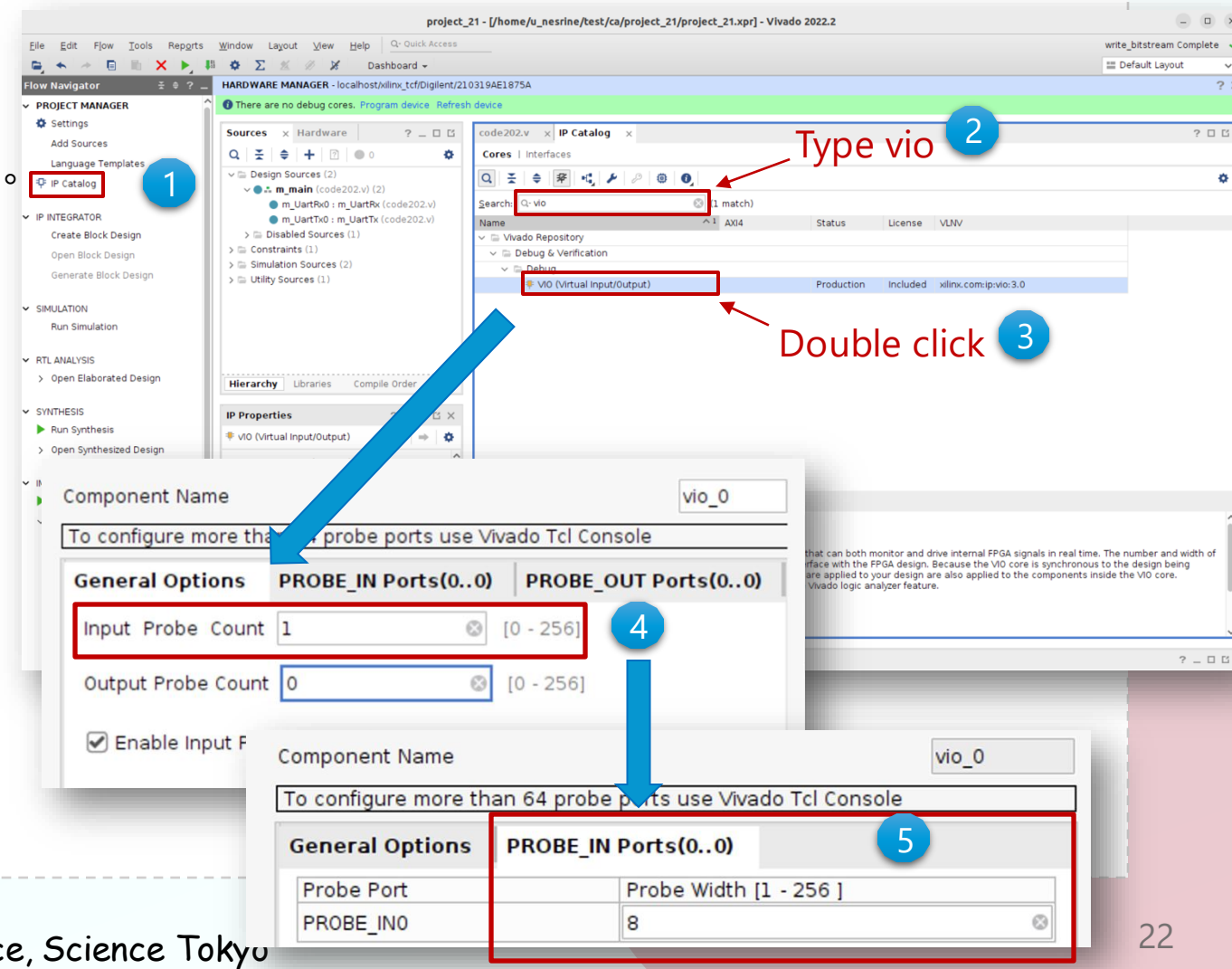
- 50MHzのクロックを出力するIP を生成する。
  - In Output Clocks, set the frequency to 50.000 for clk_out1 to generate 50MHz clock signal. Click OK.
  - In Generate Output Products window, click Generate.

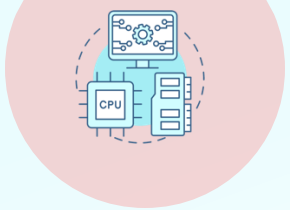


ステップ4: VIOを使った確認する

- 最後に、VIO (Virtual Input/Output)を使った確認する。

- VIO のコンフィギュレーション：
  1. Vivadoで IP Catalog を開き、vio を検索する。
  2. 以下のようにVIOを設定する：
     - Input Probe Count: 1
     - Probe Width: 8
  3. Click Generate and click OK if asked in Generate Output Products window.
- ビットストリームを生成し、FPGAにプログラムする。





ステップ5:バイナリ形式でメモリの内容が出力する

❑ 目的:

- Pythonファイル（generate_bin_file.py）を使ってメモリの内容をバイナリ形式に変換する。
  - generate_bin_file.py と同じディレクトリに sample1.txt も必要。

```
$ cd ~/ca2024/
$ cp /home/u_nesrine/ca2024/src/generate_bin_file.py
$ python3 generate_bin_file.py
```

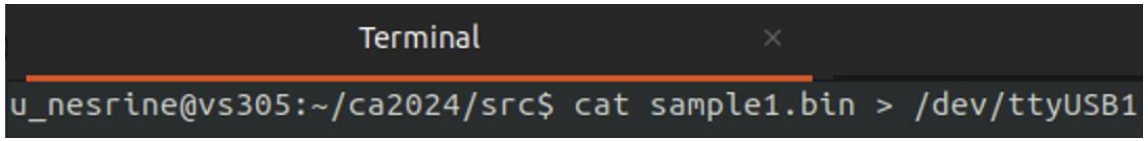
generate_bin_file.py

```python
import struct

# Read the machine code from the file 'sample1.txt'
with open("sample1.txt", "r") as file:
    # Read all lines from the file and strip any extra whitespace/newlines
    hex_instructions = [line.strip() for line in file.readlines()]

# Open a binary file to write the output
with open("sample1.bin", "wb") as bin_file:
    for hex_instruction in hex_instructions:
        # Convert each hex string to an integer
        instruction = int(hex_instruction, 16)
        # Convert the 32-bit integer to a 4-byte binary format (little-endian)
        bin_file.write(struct.pack('<I', instruction))

print("Binary file 'sample1.bin' generated successfully from 'sample1.txt'.")
```

(Source code available in /home/u_nesrine/ca2024/src)

ステップ6: UARTによるシリアル通信

- UART を用いてバイナリデータを FPGA に送信する。
  - ターミナルを開いて、次のコマンドを実行して sample1.bin をFPGAに送信する。
    - GtkTerm でボーレートなどを設定してからコマンドを実行すること。
    - cat sample1.bin > /dev/ttyUSB1



ステップ7: VIO で「a」という文字が出力されるか確認する

- 生成した sample1.bin を送信して，転送されたデータの値が 0x61 となることを VIO を用いて確認する。

hw_vio_1

Name	Value	Activ...	Direct...	VIO
r_tohost_char[7:0]	[H] 61		Input	hw_vio_1
r_tohost_cmd[1:0]	[H] 2		Input	hw_vio_1

Check Point 5