

# 計算機設計論 レポート課題:MIPS プロセッサの回路設計

1295149 森岡悠人

1295148 松本吏司

2025 年 8 月 23 日

## 1 モジュール仕様書

設計した各モジュールについて，上位階層から順にモジュールの仕様を示す．

### 1.1 Golden\_Top

このモジュールは最上位に位置し，コンパイル時に `IMem.txt` に書き込まれた命令をボタン押下に応じて順に実行する．現在のプログラムカウンタや内部モジュールの出力を 7 セグ LED で表示し，命令の実行結果が確認できるように構成している．

デフォルトで定義されているクロックや GPIO の定義に加え，以下のモジュールをインスタンス化する．モジュール名とインスタンス名を表 1 に示す．上記のモジュールの接続関係を図 1 に示す．

表 1 Golden\_Top で利用するモジュール

モジュール名	インスタンス名
BTN_IN	BTN0
SingleCycleClockMIPS	SCCM0
SELECTOR	SEL0
SEG7DEC	Res00
SEG7DEC	Res01
SEG7DEC	Res02
SEG7DEC	Res03
SEG7DEC	PC00
SEG7DEC	PC01

### 1.2 BTN\_IN

このモジュールは Golden\_Top 内で利用されており，DE10-lite の KEY 及び SW から得られる入力のチャタリングを排除し，KEY は立下がりを検出する．本モジュールの入力・出力信号を表 2, 3 に示す．

チャタリングを排除するために，40hz 毎に入力信号 KEY, SW の値を読み出す．このとき，入力を読み出

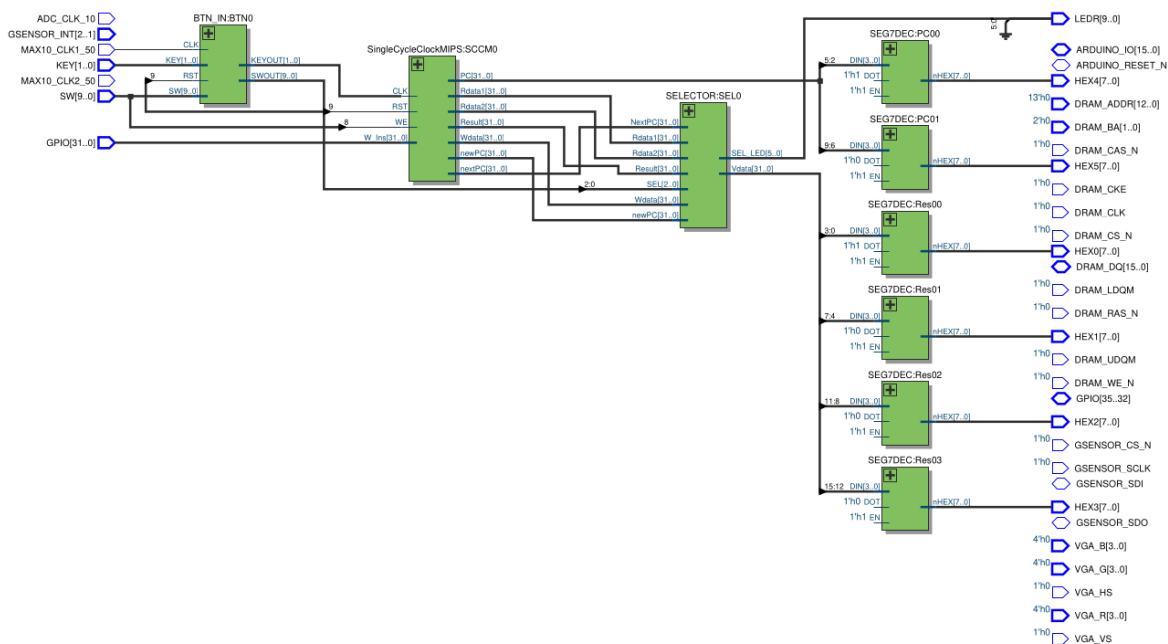


図 1 Golden.Top のサブモジュール間の接続関係

表 2 BTN.IN の入力信号

信号名	ビット幅	説明
CLK	[0:0]	DE10-lite で生成するクロック信号
RST	[0:0]	リセット用信号 (DE10-lite の SW9)
KEY	[1:0]	DE10-lite の KEY 入力信号
SW	[9:0]	DE10-lite の SW 入力信号

表 3 BTN.IN の出力信号

信号名	ビット幅	説明
KEYOUT	[1:0]	チャタリングを排除した KEY の立下がりレジスタ値
SWOUT	[9:0]	チャタリングを排除した SW のレジスタ値

すフリップフロップ (FF) と直前の値を読み出す FF2 個を用いる。入力ではなく直前の値を利用することで、チャタリング排除する。同期回路でチャタリングを排除した信号を出力する。

### 1.3 SingleCycleClockMIPS

このモジュールは Golden.Top 内で利用されており、コンパイル時に IMem.txt で指定された命令を実行するシングルサイクルクロックの MIPS である。動作クロックは、DE10-lite の KEY0 とする。本モジュールの入力・出力信号を表 4, 5 に示す。

本モジュールでは表 6 のサブモジュールを利用する。また、サブモジュール間の接続関係を図 2 に示す。

表 4 SingleCycleClockMIPS の入力

信号名	ビット幅	説明
CLK	[0:0]	SingleCycleClockMIPS 用のクロック信号
RST	[0:0]	リセット用信号
WE	[0:0]	コンパイル時に IF モジュールの最適化を防止するための信号
W_Ins	[31:0]	コンパイル時に IF モジュールの最適化を防止するための信号

表 5 SingleCycleClockMIPS の出力

信号名	ビット幅	説明
PC	[31:0]	今から実行する命令のアドレスを示す値
Result	[31:0]	EX モジュールで出力する ALU の結果を示す値
Rdata1	[31:0]	ID モジュールで出力, IF モジュールで入力される値
Rdata2	[31:0]	ID モジュールで出力, IF・MA モジュールで入力される値
Wdata	[31:0]	MA モジュールで出力, ID モジュールで入力される値
nextPC	[31:0]	PC に 4 を加えた値
newPC	[31:0]	EX モジュールで決定する次の命令アドレスを示す値

表 6 SingleCycleClockMIPS で用いるサブモジュール

モジュール名	インスタンス名
IF	IF0
ID	ID0
EX	EX0
MA	MA0

本モジュールは IF モジュールを利用し、命令メモリから命令を読み出し、ID, EX, MA モジュールを利用し、命令を実行する。

## 1.4 IF

このモジュールは SingleCycleClockMIPS 内で利用されており、入力として与えられたプログラムカウンタ (PC) に記載された命令を読み出し、それを `Ins` として出力する。また、PC に 4 を加えた値を `nextPC` として出力する。命令メモリの保持、読み出しには IM モジュールを用いる。コンパイル時に最適化され命令メモリのアドレスが変わることを防ぐため、SW8 及び GPIO の出力を入力とし、それぞれ IM の入力とする。

本モジュールの入力・出力信号を表 7,8 に示す。

本モジュールでは、IM モジュールを IM0 としてインスタンス化する。

## 1.5 IM

本モジュールは IF 内で利用されており、入力として与えられた PC に記載された命令を読み出し、`Ins` としてその値を出力する。命令はワード単位であるため、PC を 2 ビット右シフトした値の番地から命令を読み

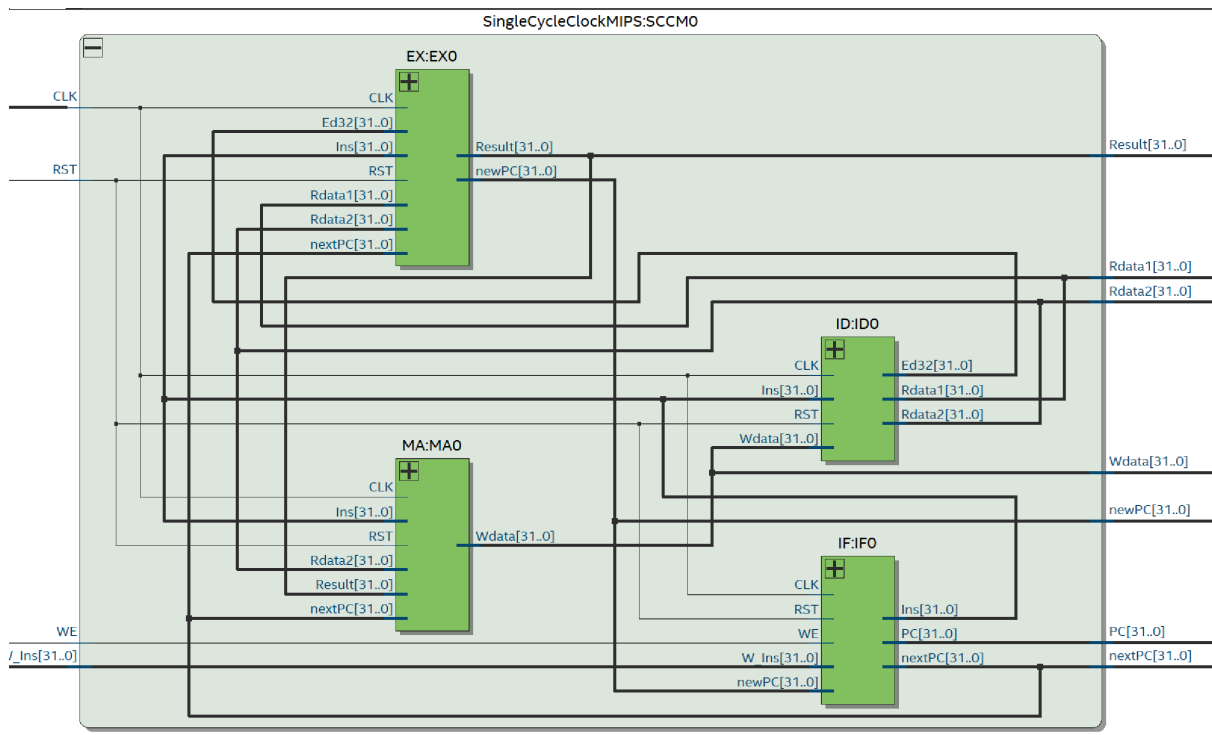


図 2 SingleCycleClockMIPS のサブモジュール接続関係

表 7 IF の入力

信号名	ビット幅	説明
CLK	[0:0]	SingleCycleClockMIPS 用のクロック信号
RST	[0:0]	リセット用信号
WE	[0:0]	コンパイル時に IF モジュールの最適化を防止するための信号
W_Ins	[31:0]	コンパイル時に IF モジュールの最適化を防止するための信号
newPC	[31:0]	次に実行する命令のアドレスを示す値

表 8 IF の出力

信号名	ビット幅	説明
PC	[31:0]	今から実行する命令のアドレスを示す値
nextPC	[31:0]	PC に 4 を加えた値
Ins	[31:0]	IM モジュールを用いて読み出した命令を示す値

出す。またコンパイル時に IMem.txt として命令を読み出し、最大 IMEM\_SIZE 個の命令をレジスタに格納する。その他に、コンパイル時の最適化によってアドレスが変わることを防ぐため、今回は動作することはないが、命令メモリの書き換え信号 WE が入力された場合に W\_Ins の値を命令メモリ (レジスタ) に書き込む処理を加えている。

本モジュールの入力・出力信号を表 9, 10 に示す。

表 9 IM の入力

信号名	ビット幅	説明
CLK	[0:0]	SingleCycleClockMIPS 用のクロック信号
RST	[0:0]	リセット用信号
WE	[0:0]	コンパイル時に IF モジュールの最適化を防止するための信号
W_Ins	[31:0]	コンパイル時に IF モジュールの最適化を防止するための信号
PC	[31:0]	命令を読み出すアドレス

表 10 IM の出力

信号名	ビット幅	説明
Ins	[31:0]	命令メモリから読み出した命令を示す値

## 1.6 ID

本モジュールは SingleClockCycleMIPS 内で利用されており、命令を示す値に従ってレジスタファイルから値を読み出し、それぞれ Rdata1, Rdata2 として出力する。また、命令に応じて符号拡張した値を Ed32 として出力する。また、レジスタファイルへの書き込み信号 WE が入力された場合に、Wdata で示された値をレジスタファイルの適切なアドレスへ書き込む。なお、WE は命令解釈によって生成する。

本モジュールの入力・出力信号を表 11, 12 に示す。

表 11 ID の入力

信号名	ビット幅	説明
CLK	[0:0]	SingleCycleClockMIPS 用のクロック信号
RST	[0:0]	リセット用信号
Ins	[31:0]	IF モジュールで読み出した命令を示す値
Wdata	[31:0]	MA モジュールで出力されるレジスタファイルへ書き込む値

表 12 ID の出力

信号名	ビット幅	説明
Rdata1	[31:0]	Ins[25:21] から読み出したレジスタファイルの値
Rdata2	[31:0]	Ins[20:16] から読み出したレジスタファイルの値
Ed32	[31:0]	Ins[15:0] を符号拡張した値

## 1.7 EX

本モジュールは SingleClockCycleMIPS 内で利用されており、加算や減算などの演算を行う ALU と、分岐命令に応じて newPC を決定する部分からなる。ALU は入力された Rdata1, と Rdata2, Ed32 から命令に応じて決定した片方の値を用いて演算を行う。演算の種類は Ins を用い、解釈することで決定する。また、分岐

命令とジャンプ命令が読み出された場合に、ALU の演算結果や Ed32 の値から newPC を決定し、出力する。  
本モジュールの入力・出力信号を表 13, 14 に示す。

表 13 EX の入力

信号名	ビット幅	説明
CLK	[0:0]	SingleCycleClockMIPS 用のクロック信号
RST	[0:0]	リセット用信号
Ins	[31:0]	IF モジュールで読み出した命令を示す値
Rdata1	[31:0]	ID モジュールで読み出した値
Rdata2	[31:0]	ID モジュールで読み出した値
Ed32	[31:0]	ID モジュールで生成した符号拡張した値
nextPC	[31:0]	IF モジュールで生成した PC に 4 を加えた値

表 14 EX の出力

信号名	ビット幅	説明
Result	[31:0]	ALU の演算結果の値
newPC	[31:0]	次の命令アドレスを示す値

## 1.8 MA

本モジュールは SingleCycleClockMIPS 内で利用されており、データメモリ、すなわちスタック領域を扱う。命令に応じてレジスタファイルへ書き込む値を出力するか、入力される Rdata2 で指定された値をアドレスの位置に書き込む。命令がジャンプである場合は nextPC を出力し、I 形式算術命令または R 形式命令の場合は Result を、ロード命令の場合に DM モジュールを用いてデータメモリから読み出した値を出力する。  
本モジュールの入力・出力信号を表 23, 24 に示す。

表 15 ma の入力

信号名	ビット幅	説明
CLK	[0:0]	SingleCycleClockMIPS 用のクロック信号
RST	[0:0]	リセット用信号
Ins	[31:0]	IF モジュールで読み出した命令を示す値
Result	[31:0]	EX モジュールの ALU の演算結果の値 (データメモリのアドレス)
Rdata2	[31:0]	ID モジュールで読み出した値 (書き込む値)
nextPC	[31:0]	IF モジュールで生成した PC に 4 を加えた値

表 16 MA の出力

信号名	ビット幅	説明
Wdata	[31:0]	レジスタファイルへ書き込む値

本モジュールでは、DM モジュールを DM0 としてインスタンス化する。

## 1.9 DM

本モジュールは、MA 内で利用されており、データメモリを扱う。今回はデータメモリのサイズが 128 であるため、入力されたアドレスの下位 7 ビットのみを利用してデータメモリを示すレジスタから値を読み出す。データメモリへの書き込みを示す WE が入力された場合は、入力されたアドレスへ値を書き込む。

本モジュールの入力・出力信号を表??, ??に示す。

表 17 ma の入力

信号名	ビット幅	説明
CLK	[0:0]	SingleCycleClockMIPS 用のクロック信号
RST	[0:0]	リセット用信号
Adr	[31:0]	読み出し、書き込み先アドレス ([6:0] のみ利用)
WDATA	[31:0]	書き込みデータ

表 18 MA の出力

信号名	ビット幅	説明
Rdata	[31:0]	データメモリから読み出した値

## 1.10 SELECTOR

本モジュールは Golden\_Top で利用されており、SingleCycleClockMIPS の動作を確認するために用いる 7SEG に表示するデータを出力する。DE10-lite の SW2-0 を利用し、6 種類のデータを選択する。表 19 に入力パターンと選択する信号を示す。

本モジュールの入力・出力信号を表??, ??に示す。

表 19 SELECTOR の入力パターンと出力信号

SW2, SW1, SW0	出力信号
0, 0, 0	Rdata1
0, 0, 1	Rdata2
0, 1, 0	Result
0, 1, 1	Wdata
1, 0, 0	nextPC
1, 0, 1	newPC

表 20 MA の入力

信号名	ビット幅	説明
Rdata1	[31:0]	ID モジュールで読み出した値
Rdata2	[31:0]	ID モジュールで読み出した値
Result	[31:0]	EX モジュールの ALU の演算結果
Wdata	[31:0]	ID モジュールで書き込む値
nextPC	[31:0]	PC に 4 を加えた値
newPC	[31:0]	次に命令を実行するアドレス

表 21 MA の出力

信号名	ビット幅	説明
SEL_LED	[5:0]	選択した信号を示す LED
Vdata	[31:0]	選択されたデータ

表 22 SEG7DEC に与える信号

DE10-lite の表示場所	インスタンス名	与える信号
HEX0	Res00	vdata[3:0]
HEX1	Res00	vdata[7:4]
HEX2	Res00	vdata[11:8]
HEX3	Res00	vdata[15:12]
HEX4	Res00	PC[5:2]
HEX5	Res00	PC[9:6]

## 1.11 SEG7DEC

本モジュールは Golden\_Top で利用されており、PC および、SELECTOR モジュールで選択された信号を表示するために用いられる。このモジュールは 1 桁のみ表示することができるため、表 22 に示すように 6 個の 7SEG にそれぞれ信号を振り分ける。なお、vdata は SELECTOR の出力信号である。

本モジュールの入力・出力信号を表??, ??に示す。

表 23 SEG7DEC の入力

信号名	ビット幅	説明
DIN	[3:0]	0-F の数値
EN	[0:0]	値を表示するかを選択する信号
DOT	[0:0]	7SEG のドットを表示するかを選択する信号



表 24 SEG7DEC の出力

信号名	ビット幅	説明
nHEX	[7:0]	7SEG の点灯状態を示す値

## 2 動作検証

作成した verilog コードが MIPS の命令セットを実行できるかどうかの検証を行った．テストプログラムとして，教科書 [1] に掲載されているアセンブラプログラム (load\_store, arithmetic, array, if\_then\_else, while, function, recursion, hanoi) を用いた．プログラムは CPUlator MIPS System Simulator [2] を用いてコンパイルした．その後 32bit の 16 進数で出力されたバイナリを IMem.txt に書き込んでおき，IM に読み込ませた状態でシミュレーションを実行した．動作の流れとデータメモリの中身を確認するため，modelsim20.1 を用いて動作のシミュレーションと検証を行った．シミュレーション結果は，display 命令を用いて，PC, Instruction, ALU\_result レジスタの順番が期待通りかどうかを確認した．シミュレーションの様子を図 3 に示す．

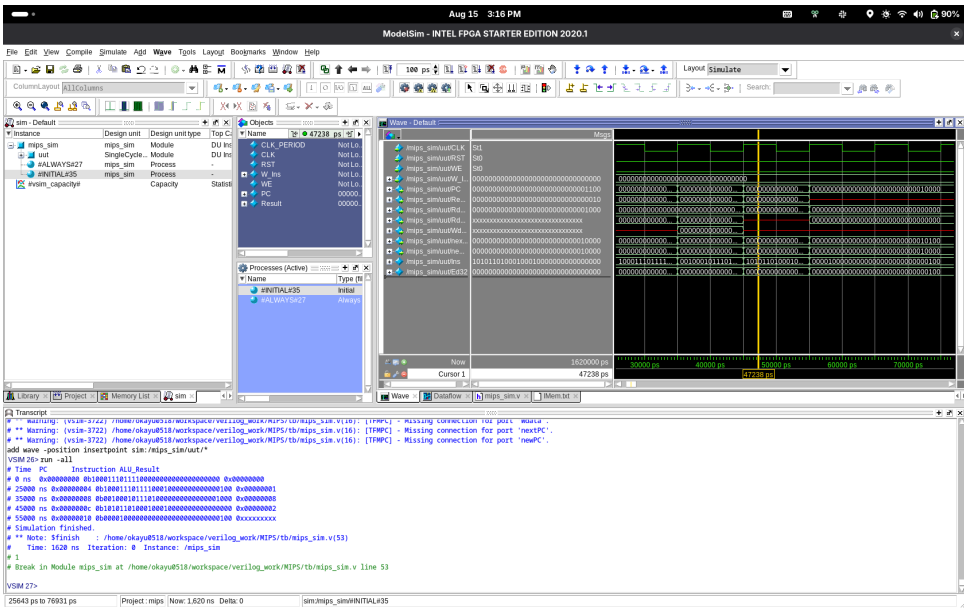


図 3 modelsim によるシミュレーションの様子

## 2.1 test: load\_store

基本的なロード・ストア命令の動作確認を行った。このプログラムではデータメモリから値をロードし、別のアドレスにストアする処理を行う。プログラムの動作は以下の通りである：

- lw \$s0, 0(\$s7): データメモリのアドレス\$s7+0 から値をロードし、\$s0 に格納
- lw \$s1, 4(\$s7): データメモリのアドレス\$s7+4 から値をロードし、\$s1 に格納

- `addi $t0, $s7, 8`:  $\$s7+8$  をアドレスとして  $\$t0$  に計算
- `sw $s1, 0($t0)`:  $\$s1$  の値を  $\$t0$  が示すアドレスにストア

あらかじめテストベンチで DMem の 0 番地からそれぞれ静的変数  $a=10$ ,  $b=20$  と、配列  $a[0]=0$ , 配列のオフセットを保持するレジスタ  $\$s7=0$  を初期値として設定してシミュレーションを実行した。その結果、期待通り  $\$s0=10$ ,  $\$s1=20$ ,  $\text{DMem}[0]=10$  であることを確認した。使用したアセンブリコードとそのテスト結果を付録 A.1 に示す。

## 2.2 test: arithmetic

基本的な算術演算命令の動作を確認するためのテストを行った。このプログラムは 3 つの値を読み込んで加算を行い、結果をレジスタに格納する。プログラムの動作は以下の通りである：

- 変数  $a, b, c$  の値 (1, 2, 3) をそれぞれ  $\$s0, \$s1, \$s2$  にロード
- `add $t0, $s0, $s1`:  $a + b$  の結果を  $\$t0$  に格納
- `add $s3, $t0, $s2`:  $(a + b) + c$  の結果を  $\$s3$  に格納 (期待値: 6)

あらかじめテストベンチで DMem の 0 番地からそれぞれ静的変数  $a=1, b=2, c=3, d=0$  と変数のオフセットを保持するレジスタ  $\$s7=0$  を初期値として設定してシミュレーションを実行した。その結果、期待通り  $a, b, c$  の値が `add` 命令で足し合わされ、 $\$s3=6$  であることを確認した。使用したアセンブリコードとそのテスト結果を付録 A.2 に示す。

## 2.3 test: array

配列のアクセスとインデックス計算の動作を確認するためのテストを行った。プログラムの動作は以下の通りである：

- `sll $t0, $s0, 2`: インデックス  $\$s0$  を 4 倍 (左シフト 2 ビット) してワードアドレスに変換
- `add $t0, $s7, $t0`: 配列の基底アドレス  $\$s7$  にオフセットを加算
- `lw $s1, 0($t0)`: 計算されたアドレスから配列要素をロード
- `lw $s2, 20($s7)`: 配列の 5 番目の要素 (20 バイトオフセット) をロード

あらかじめテストベンチで DMem の 0 番地からそれぞれ配列  $a[0]=0, a[1]=1, \dots, a[9]=9$  と配列のオフセットを保持するレジスタ  $\$s7=0$  と  $\$s2=2$  を初期値として設定してシミュレーションを実行した。その結果、期待通り  $\$s1=2, \$s2=5$  であることを確認した。使用したアセンブリコードとそのテスト結果を付録 A.3 に示す。

## 2.4 test: if\_then\_else

条件分岐命令の動作を確認するためのテストを行った。プログラムの動作は以下の通りである：

- `beq $s0, $s1, L1`:  $\$s0$  と  $\$s1$  が等しい場合は L1 にジャンプ
- 等しくない場合:  $\$s2 = \$s0$  を実行して N1 にジャンプ
- 等しい場合 (L1):  $\$s2 = \$s1$  を実行

まず、if 文が真になる場合のテストを行った。あらかじめテストベンチでレジスタ \$s0=0xa, \$s1=0xa, \$s2=0x0 を初期値として設定してシミュレーションを実行した。その結果、\$s0 と \$s1 が等しいため、\$s2 に \$s1 の値が代入され、\$s2=0xa であることを確認した。次に、if 文が偽になる場合のテストを行った。あらかじめテストベンチでレジスタ \$s0=0xa, \$s1=0xb, \$s2=0x0 を初期値として設定してシミュレーションを実行した。その結果、\$s0 と \$s1 が等しくないため、\$s2 に \$s0 の値が代入され、\$s2=0xa であることを確認した。真の場合と偽の場合の PC の遷移を比べると、偽の場合は else 節のラベルを飛び越えるために J 命令が実行されているためことがわかる。そのため偽の場合は 1 命令多い。使用したアセンブリコードとそのテスト結果を付録 A.4 に示す。

## 2.5 test: while

ループ処理と条件判定の動作を確認するためのテストを行った。プログラムの動作は以下の通りである：

- `slti $t0, $s0, 10`: \$s0 < 10 の条件判定結果を \$t0 に格納
- 条件が偽 (`$s0 >= 10`) の場合はループを終了して N1 にジャンプ
- 条件が真の場合：配列に \$s0 の値を格納し、\$s0 をインクリメントしてループを継続

あらかじめテストベンチで配列 a[10] を 0 で初期化し、配列のオフセットを保持するレジスタ \$s7=0, \$s0=0 を初期値として設定してシミュレーションを実行した。その結果、\$s0 が 10 になるまで while 文が繰り返され、配列 a[0] から a[9] にそれぞれ 0 から 9 までの値が代入されていることを確認した。使用したアセンブリコードとそのテスト結果を付録 A.5 に示す。

## 2.6 test: function

関数呼び出しとスタック操作の動作を確認するためのテストを行った。このプログラムは 1 から n までの和を計算する関数を実装している。プログラムの動作は以下の通りである：

- メイン部分：引数を \$a0 に設定して sum 関数を呼び出し、戻り値を \$s1 に格納
- sum 関数：スタックにレジスタを退避し、1 から n までの和を計算して \$v0 に結果を返す
- 関数終了時にはスタックからレジスタを復元し、呼び出し元に戻る

あらかじめテストベンチでレジスタ \$s0=10 を初期値として設定して 1 から n までの和を求めるプログラムのシミュレーションを実行した。その結果、\$s1=0x37=0d55 となることを確認した。使用したアセンブリコードとそのテスト結果を付録 A.6 に示す。

## 2.7 test: recursion

再帰関数呼び出しの動作を確認するためのテストを行った。このプログラムは再帰を用いて 1 から n までの和を計算する。プログラムの動作は以下の通りである：

- 引数が 1 未満の場合は 0 を返してベースケースとする
- 引数が 1 以上の場合は、引数を 1 減らして再帰呼び出しを行い、その結果に現在の引数値を加算
- 各再帰レベルでスタックに引数と戻りアドレスを保存・復元

2.6 のテストと同様に、あらかじめテストベンチでレジスタ\$s0=10 を初期値として設定してシミュレーションを実行した。その結果、\$s1=0x37=0d55 となることを確認した。使用したアセンブリコードとそのテスト結果を付録 A.7 に示す。

## 2.8 test: hanoi

ハノイの塔を解く再帰アルゴリズムを実装し、複雑な再帰処理の動作を確認するためのテストを行った。プログラムの動作は以下の通りである：

- 3 枚の円盤のハノイの塔問題を解く
- \$a0: 円盤の枚数、\$a1: 移動元、\$a2: 移動先、\$a3: 補助杆
- \$t1: 移動回数カウンタ（期待値：7 回）
- 再帰の深さに応じてスタックに多くの引数と戻りアドレスを保存
- ベースケース（円盤が 1 枚）では直接移動を実行

円盤が 3 枚のハノイの塔を解くプログラムを実行する。レジスタ\$t1 を移動回数カウント用としてシミュレーションを行った。結果、\$t1=7 となり、期待通りの動作を確認した。hanoi のテストは算術演算、関数呼び出しなど、上記の多くの処理を含むため、Quartus Prime を用いて DE10-Lite に書き込んで実行した。その結果、modelsim 同様に期待通りの動作をし、最後の無限ループの処理まで到達することを確認できた。使用したアセンブリコードとそのテスト結果を付録 A.8 に示す。

## 3 考察

### 3.1 森岡悠人

今回の MIPS 回路設計の課題は松本と協力して課題を分担して取り組んだ。定期的に対面で集まって作業し、問題点やバグを共有しながら進めた。どうしても理解できない部分や、詰まった部分は岩田研究室の M1 に聞きに行くことで解決した。github のリポジトリ上で共同編集を行ったが、回路のソースとテストベンチを分けて開発したため、変更が競合しづらく、効率よく開発を進めることができた。私は作成した回路のテストを担当した。まず、EX.v のテストベンチを書いて各命令を入力したときの ALU の result をチェックし、その後、MIPS 全体のテストベンチを書いて教科書にあるテストプログラムを使って命令実行時の挙動を確認した。テストベンチを作成する際、modelsim の wave だけでは確認するときの効率が悪かったため、display 命令を用いてテスト内容と結果はすべてコンソールに出力させた。また、テスト内容について、生成 AI を用いて効率よくテストを作成することを検討したが、生成 AI は verilog の知識はあまりない様子で、生成したテスト内容はところどころミスがあり、結局すべてのテストを手動で確認して修正した。しかし、テストベンチ全体の構造は生成 AI を用いて作成したため、ある程度効率的にテストベンチを作成できた。このとき、Instruction の値は、微妙な間隔でビットの区切りがあるため、バイナリで書いた。さらに、命令の形式に対応したビットの区切りとなる箇所にはアンダーバーを入れて記述することで読みやすくなるように工夫した。EX.v のテストベンチ全体として、test.instruction という task を作成し、そこに各命令のテストとなる入力と期待する出力を入れ、比較することでテストを行った。回路をデバッグする中で意識したことは、回路の出力が期待通りにならないとき、信号の流れを一つずつ追うことである。modelsim のシミュレーションのみたい箇所の信号が確認できる機能を使って、どの信号線まで正常に信号が流れているかを入力から順番に確認

することで、問題を特定できた。また、modelsim のデータメモリの中身を確認する機能も sw 命令のデバッグで非常に役に立った。ただし、Online MIPS Simulator では.data 領域に確保するメモリサイズや初期値を記述すればコンパイルするときに勝手にメモリ上に配置してくれたのに対し、作成した MIPS ではその機能がないため、テストベンチで手動でデータメモリの初期値を設定する必要があった。これは大変だと感じ、既存のコンパイラやアセンブラの自動でメモリに初期値を書き込んでくれる機能のありがたみを感じた。ただし、データメモリに初期値を書き込んだりするのは回路設計の果たして回路レベルでやるべき処理なのか疑問に思った。

今回作成した MIPS の改善点について述べる。まず、FPGA ボードの 7 セグに常に PC をワード単位で表示させたが、これはバイト単位の表示のほうが他シミュレータの結果との比較がしやすいのではないかと考える。また、今回は IMem.txt にバイナリを書き込んでおき、それを読み込む形で命令を実行したが、テストベンチから IM に読み込ませるテキストファイルを指定できるようにできれば、あらかじめテストプログラムをファイル単位で用意しておいて実行できて、テストが捗るのではないかと考えた。しかしそれはそれで、modelsim にも quartus にも IMem.txt を読み込む処理を別々に書かないといけないうえ、面倒だなと思った。また、教科書のテストプログラムについて、MULT や DIV など、実装したすべての命令が網羅できているとは言えず、common\_param.vh のすべての命令をテストしようと思うと別で大量のテストベンチを書く必要がある。そのため、もう少し網羅的かつ統一的にテストできるテストプログラムの雛形と正解となる出力データを先生側で用意してもらえるとテストも効率的に行えて、課題の確認もスムーズに行えて良いのではないかと考えた。また、講義内でテストベンチの作り方について説明が少なかったように感じられた。せめて、1 命令のテスト分、また、display 命令の使い方については説明してくれたほうが良いのではないかと考える。今回の MIPS は教科書に従って SE/UE や、MUX も全て assign 文で実装した。そのため、流れが追いやすく、デバッグしやすかった反面、他にも実装パターンは複数考えられるので、本当はさらに効率の良い実装方法があるのだろうなと思った。それとも、コンパイルして論理合成すると、結局同じ回路になるのかどうか興味を湧いた。

## 3.2 松本吏司

本授業での MIPS 回路設計の課題は、森岡君と協力して取り組んだ。基本的に自身が各モジュールの実装を行い、テストベンチの作成と動作のテストを森岡君に任せた。作業を分担して行うにあたり、それぞれが書いたコードの共有には github のリポジトリを活用した。今回はブランチを分けるようなことはしなかったが、モジュールの実装とテストベンチの作成でディレクトリを分けたため、競合などの問題は特に発生しなかった。これにより、スムーズに開発とテストを繰り返すことができる環境が整備できていたと思う。

自身の担当したモジュールの実装についての考察及び、そこから得られる改善点を述べる。基本的に教科書の回路図を参考にし、マルチプレクサやその他のモジュールについてもそれぞれ独立したものとして実装を行った。しかし、教科書には記述が不足しているジャンプ命令や分岐命令の実装方法について理解できないところがあった。また、初めて扱った言語 verilog の文法要素で何が問題か自身で解決できないところがあった。そのような箇所は、岩田研究室の修士の学生に授業時間外に質問に行くことで解決することができた。丁寧に順を追って動作を確認し、何が問題なのかの説明をしていただいて、非常に助かった。その際に、動作概念とプログラムの記述内容が一对一に対応している図が開発には必要であると考えた。動作概念は大きく分けて、IF, ID, EX, MA の 4 個のモジュールに分割できるが、それぞれのモジュールで繰り返し同じような処理である命令解読が必要であり、どのモジュールでも共通の記述でないことが大きな混乱を招き、デバッグの困

難を極めたと考える。しかし、森岡君が記述したテストベンチによって、どのモジュールでどのような信号のパターンのときに問題が起きるかを知ることができたため、実装のミスを修正することができた。終盤に非常に躓いた箇所として、データメモリ領域であるレジスタの配列にアクセスする際、スタックの挙動である通りに配列末尾からアクセスをする必要があったが、正常に値の書き込み及び読み込みができないことがあった。これはアクセスするインデックス、すなわち与えられるアドレスの大きさがデータメモリのサイズを大幅に超えていたことが原因であったため、末尾の必要なビットのみを取り出すことで解消できた。ここで、ModelSim を用いてテストを行ったところ、他の動作には影響を与えていないように見えたが、回路合成によって DE10-lite 上で修正前の挙動、すなわちレジスタの配列を超える場所へのアクセスを行った場合は何が起きるのだろうかという疑問に思った。c 言語などのように未定義の場所へアクセスを行い何らかの値を取得するのか、それともそもそもそのような領域が存在しないため回路として何らかの保護機能が働くのかが疑問である。このようなモジュールの実装にあたって、物理的に存在しない領域へのアクセスを禁止する保護機能のようなものを書く必要があるのではないかと考えた。その他に、データメモリやレジスタファイルの初期化をリセット時に行う必要があるのかは疑問である。メモリの初期値は命令を書くユーザまたはコンパイラが担保すべきだと考えるが、アーキテクチャとして初期化が組み込まれている方が安全に利用できるのではないかと考えた。しかし、これについては単一のプログラムを実行するだけであればリセットの必要回数は少なくなると考えるが、現代の CPU などは複数のプログラムを実行するため、スイッチ時の初期化が大幅なボトルネックになるのではないかと考えた。デバッグ目的のための 7 セグの利用について、コンソールのような標準出力のような気分で表示を行うことができれば、かなり有効に扱えるのではないかと考える。しかし、7 セグでは文字を表示することが非常に難しいため、数値として表現したデータの解釈を常に人が行わなければいけないのは不便であると感じた。

## 4 感想

### 4.1 森岡悠人

この講義で初めて Verilog や FPGA 開発を経験したが、これまでプログラミング言語しか触れてこなかったため、verilog の独自の命令や module の考え方、並列で回路の処理が実行される考え方は新鮮で面白かった。実際に FPGA が動作したときは達成感を感じた。一方、初めての HDL で、面食らったことも多々あった。modelsim および quartus prime の操作やプロジェクトファイルの構造が独特で、かつ頻繁に強制終了するため、実装に非常に時間がかかった。シミュレーションで、メモリの中身を見る機能や、回路の wave を確認する画面で、自由にソースコードの信号を追加して確認できる機能は非常に便利だった。だが、門屋に教えてもらうまではその存在に気が付かなかったため、もうちょっとわかりやすい UI にしてほしいと思った。ハードウェアを設計する技術者の使うツールだから、そのために用いるソフトウェアもあまり出来が良くないのではないかと考えた。また、私が普段利用する ChatGPT のような AI は、おそらく Verilog のコード生成に必要な学習データが不足していて、間違ったコードを生成することが多く、あまり役に立たなかった。また、1 時間計から MIPS に取り掛かるときにレベルが一気に 3 段階くらい上がるなという印象を受けた。

私は将来は低レベルのコンピュータの動作を理解したエンジニアを目指しているため、この講義を受講した。その結果、かつて苦しんだアセンブラの実験の内容をおさらいできたことに加え、FPGA 実機を用いた verilog の回路設計の経験を積むことができ、新たな知見を得られたため、受講して良かったと感じている。また、ペアの松本の実装が速く、常に私に先行して ALU や命令実行時の動作について教えてくれた。起こっ

た問題に対して二人でアイデアを出し合いながら原因を検討し、試行錯誤の中で回路の理解を深められたことが、今回の課題がうまく行った理由ではないかと感じている。

## 4.2 松本吏司

本授業で初めて実際に FPGA を動作させるためにハードウェア記述言語である verilog を用いて回路設計を行ったが、ブロッキングやノンブロッキングを考える必要があり、新鮮さを感じた。modelsim を用いて自身の書いたモジュールをテストし、期待した通りに動いたときや、それが実際に FPGA で動作したときには強い達成感を得た。しかし、これまでに馴染みのないツールを用いることによる開発で、問題のある箇所の特定や、便利なツールやデバッグのコツなどを知るまでに多くの時間を費やしたと思う。特に、wire 宣言を忘れたままでも assign 文のコンパイルが正常に行われ、1 ビットの信号と自動的にされてしまうことに、度々原因がわからず悩まされた。また、modelsim 上でテストベンチに記載していないレジスタの値を確認できることを知るまで、どこから問題の切り分けをしていけばよいのかが分からなく、難航した。情報学群実験の授業でアセンブリの記述について学び、これを復習し活かすことができる良い授業だったのではないと思う。ただ、verilog に関するドキュメントが自身にとって基本的に難解であり、実装に当たる基本的な考え方があやふやな部分が多かったと感じた。それでも、ペアで協力してくれた森岡君やその他に教えてくれた他の学生のおかげで、授業についていくことができたと感じる。

非常に内容がヘビーな授業であったと思うが、これまでに自身になかった新しい考え方を養うことができ、良い経験になったと思う。今後、この授業が直接活かされる場面に遭遇するかはわからないが、ペアでの協力や新しい物事への挑戦心、躓いたときの対処法などは活かすことができると考える。

modelsim の扱い方について、コンパイルしてシュミレーションができるというだけでなく、それぞれの時点での記述したレジスタがどのような値を持っているかを確認する方法や、それらを文字として表示する方法 (display) などについては一通り説明、または最初のほうに配布されたテストベンチに記述をしていただけると非常に助かると思います。

## 謝辞

本課題に取り組む過程で、忙しい中何度も有益な助言をくださった大崎綾斗さん、門屋陽丈さんに感謝いたします。

## 付録 A テストプログラム

本節では、各テストプログラムのアセンブリコードとその動作、および実行結果について説明する。

### A.1 load\_store

#### A.1.1 プログラム

このプログラムは基本的なロード・ストア命令の動作を確認するためのテストである。プログラムでは、データメモリから値をロードし、別のアドレスにストアする処理を行う。

```
.set noreorder
```

```

.global _start
_start:
lw $s0, 0($s7)
lw $s1, 4($s7)
addi $t0, $s7, 8
sw $s1, 0($t0)
loop:
j loop

.data
a: .word 10
b: .word 20
array1: .space 16

```

#### プログラムの動作：

- lw \$s0, 0(\$s7): データメモリのアドレス\$s7+0 から値をロードし、\$s0 に格納
- lw \$s1, 4(\$s7): データメモリのアドレス\$s7+4 から値をロードし、\$s1 に格納
- addi \$t0, \$s7, 8: \$s7+8 をアドレスとして\$t0 に計算
- sw \$s1, 0(\$t0): \$s1 の値を\$t0 が示すアドレスにストア

#### A.1.2 テスト結果

```

run -all
# Time PC Instruction ALU_Result
# 0 ns 0x00000000 0b00100000111001110000000000000000 0x00000000
# 25000 ns 0x00000004 0b10001110111100000000000000000000 0x00000000
# 35000 ns 0x00000008 0b100011101111000100000000000000100 0x00000001
# 45000 ns 0x0000000c 0b001000101110100000000000000001000 0x00000008
# 55000 ns 0x00000010 0b101011010001000100000000000000000 0x00000002
# 65000 ns 0x00000014 0b00001000000000000000000000000101 0xxxxxxxxx
# ==== Simulation Results ====
# $s0 register (R16) final value: 0x0000000a
# $s1 register (R17) final value: 0x00000014
# DMem[0] final value: 0x0000000a
# Simulation finished.

```



## A.2 arithmetic

### A.2.1 プログラム

このプログラムは基本的な算術演算命令の動作を確認するためのテストである。3つの値を読み込んで加算を行い、結果をレジスタに格納する。

```
.set noreorder

.global _start
_start:
lw $s0, 0($s7)
lw $s1, 4($s7)
lw $s2, 8($s7)
add $t0, $s0, $s1
add $s3, $t0, $s2
loop:
j loop

.data
a: .word 1
b: .word 2
c: .word 3
d: .word 0
```

#### プログラムの動作：

- 変数 a, b, c の値 (1, 2, 3) をそれぞれ \$s0, \$s1, \$s2 にロード
- add \$t0, \$s0, \$s1: a + b の結果を \$t0 に格納
- add \$s3, \$t0, \$s2: (a + b) + c の結果を \$s3 に格納 (期待値: 6)

### A.2.2 テスト結果

```
run -all
# Time PC Instruction ALU_Result
# 0 ns 0x00000000 0b10001110111100000000000000000000 0x00000000
# 25000 ns 0x00000004 0b10001110111100010000000000000100 0x00000001
# 35000 ns 0x00000008 0b10001110111100100000000000001000 0x00000002
# 45000 ns 0x0000000c 0b00000010000100010100000000100000 0x00000003
# 55000 ns 0x00000010 0b00000001000100101001100000100000 0x00000006
# 65000 ns 0x00000014 0b0000100000000000000000000000101 0xxxxxxxxx
# ==== Simulation Results ====
```

```
# $s3 register (R19) final value: 0x00000006
# Simulation finished.
```

## A.3 array

### A.3.1 プログラム

このプログラムは配列のアクセスとインデックス計算の動作を確認するためのテストである。

```
.set noreorder

.global _start
_start:
sll $t0, $s0, 2
add $t0, $s7, $t0
lw $s1, 0($t0)
lw $s2, 20($s7)
loop:
j loop

.data
a: .word 10
b: .word 20
array1: .word 0,1,2,3,4,5,6,7,8,9
```

プログラムの動作：

- sll \$t0, \$s0, 2: インデックス\$s0を4倍（左シフト2ビット）してワードアドレスに変換
- add \$t0, \$s7, \$t0: 配列の基底アドレス\$s7にオフセットを加算
- lw \$s1, 0(\$t0): 計算されたアドレスから配列要素をロード
- lw \$s2, 20(\$s7): 配列の5番目の要素（20バイトオフセット）をロード

### A.3.2 テスト結果

```
run -all
# Time PC Instruction ALU_Result
# 0 ns 0x00000000 0b0000000000001000000100000000 0x00000008
# 25000 ns 0x00000004 0b00000010111010000100000000100000 0x00000008
# 35000 ns 0x00000008 0b10001101000100010000000000000000 0x00000002
# 45000 ns 0x0000000c 0b100011101111001000000000000010100 0x00000005
# 55000 ns 0x00000010 0b000010000000000000000000000000100 0xxxxxxx
# ==== Simulation Results ====
```

```
# $s1 register (R17) final value: 0x00000002
# $s2 register (R18) final value: 0x00000005
# Simulation finished.
```

## A.4 if\_then\_else

### A.4.1 プログラム

このプログラムは条件分岐命令の動作を確認するためのテストである。

```
.set noreorder

.global _start
_start:
beq $s0, $s1, L1
add $s2, $zero, $s0
j N1
L1: add $s2, $zero, $s1
N1:
loop:
j loop

.data
a: .word 10
b: .word 20
array1: .word 0,1,2,3,4,5,6,7,8,9
```

#### プログラムの動作：

- beq \$s0, \$s1, L1: \$s0 と \$s1 が等しい場合は L1 にジャンプ
- 等しくない場合：\$s2 = \$s0 を実行して N1 にジャンプ
- 等しい場合 (L1)：\$s2 = \$s1 を実行

### A.4.2 テスト結果（真の場合）

```
run -all
# Time PC Instruction ALU_Result
# 0 ns 0x00000000 0b00010010000100010000000000000010 0x00000000
# 25000 ns 0x0000000c 0b000000000000100011001000000100000 0x0000000a
# 35000 ns 0x00000010 0b0000100000000000000000000000100 0xxxxxxxxx
# ==== Simulation Results ====
```

```
# $s0 register (R16) final value: 0x0000000a
# $s1 register (R17) final value: 0x0000000a
# $s2 register (R18) final value: 0x0000000a
# Simulation finished.
```

#### A.4.3 テスト結果（偽の場合）

```
run -all
# Time PC Instruction ALU_Result
# 0 ns 0x00000000 0b00010010000100010000000000000010 0xffffffff
# 25000 ns 0x00000004 0b00000000000100001001000000100000 0x0000000a
# 35000 ns 0x00000008 0b0000100000000000000000000000100 0xxxxxxxxx
# 45000 ns 0x00000010 0b0000100000000000000000000000100 0xxxxxxxxx
# ==== Simulation Results ====
# $s0 register (R16) final value: 0x0000000a
# $s1 register (R17) final value: 0x0000000b
# $s2 register (R18) final value: 0x0000000a
# Simulation finished.
```

### A.5 while

#### A.5.1 プログラム

このプログラムはループ処理と条件判定の動作を確認するためのテストである。

```
.set noreorder

.global _start
_start:
L1: slti $t0, $s0, 10
    beq $t0, $zero, N1
    sll $t0, $s0, 2
    add $t0, $s7, $t0
    sw $s0, 0($t0)
    addi $s0, $s0, 1
    j L1
N1:
loop:
    j loop

.data
```

```

a: .word 10
b: .word 20
array1: .space 40

```

#### プログラムの動作：

- `slti $t0, $s0, 10`:  $\$s0 < 10$  の条件判定結果を  $\$t0$  に格納
- 条件が偽 ( $\$s0 \geq 10$ ) の場合はループを終了して N1 にジャンプ
- 条件が真の場合：配列に  $\$s0$  の値を格納し、 $\$s0$  をインクリメントしてループを継続

#### A.5.2 テスト結果

```

run -all
# Time PC Instruction ALU_Result
# 0 ns 0x00000000 0b00101010000010000000000000001010 0x00000001
# 25000 ns 0x00000004 0b0001000100000000000000000000101 0x00000001
# 35000 ns 0x00000008 0b000000000000100000100000010000000 0x00000000
# 45000 ns 0x0000000c 0b00000010111010000100000000100000 0x00000000
# 55000 ns 0x00000010 0b10101101000100000000000000000000 0x00000000
# 65000 ns 0x00000014 0b00100010000100000000000000000001 0x00000001
# 75000 ns 0x00000018 0b00001000000000000000000000000000 0xxxxxxxx
# 85000 ns 0x00000000 0b00101010000010000000000000001010 0x00000001
# 95000 ns 0x00000004 0b0001000100000000000000000000101 0x00000001
# 105000 ns 0x00000008 0b00000000000010000010000001000000 0x00000004
# 115000 ns 0x0000000c 0b00000010111010000100000000100000 0x00000004
# 125000 ns 0x00000010 0b10101101000100000000000000000000 0x00000001
# 135000 ns 0x00000014 0b00100010000100000000000000000001 0x00000002
# 145000 ns 0x00000018 0b00001000000000000000000000000000 0xxxxxxxx
# 155000 ns 0x00000000 0b00101010000010000000000000001010 0x00000001
# 165000 ns 0x00000004 0b0001000100000000000000000000101 0x00000001
# 175000 ns 0x00000008 0b00000000000010000010000001000000 0x00000008
# 185000 ns 0x0000000c 0b00000010111010000100000000100000 0x00000008
# 195000 ns 0x00000010 0b10101101000100000000000000000000 0x00000002
# 205000 ns 0x00000014 0b00100010000100000000000000000001 0x00000003
# 215000 ns 0x00000018 0b00001000000000000000000000000000 0xxxxxxxx
# 225000 ns 0x00000000 0b00101010000010000000000000001010 0x00000001
# 235000 ns 0x00000004 0b0001000100000000000000000000101 0x00000001
# 245000 ns 0x00000008 0b00000000000010000010000001000000 0x0000000c
# 255000 ns 0x0000000c 0b00000010111010000100000000100000 0x0000000c
# 265000 ns 0x00000010 0b10101101000100000000000000000000 0x00000003

```

```

# 275000 ns 0x00000014 0b0010001000010000000000000000000001 0x00000004
# 285000 ns 0x00000018 0b00001000000000000000000000000000 0xxxxxxxxx
# 295000 ns 0x00000000 0b001010100000100000000000000001010 0x00000001
# 305000 ns 0x00000004 0b0001000100000000000000000000000101 0x00000001
# 315000 ns 0x00000008 0b00000000000100000100000010000000 0x00000010
# 325000 ns 0x0000000c 0b00000010111010000100000000100000 0x00000010
# 335000 ns 0x00000010 0b10101101000100000000000000000000 0x00000004
# 345000 ns 0x00000014 0b0010001000010000000000000000000001 0x00000005
# 355000 ns 0x00000018 0b00001000000000000000000000000000 0xxxxxxxxx
# 365000 ns 0x00000000 0b001010100000100000000000000001010 0x00000001
# 375000 ns 0x00000004 0b0001000100000000000000000000000101 0x00000001
# 385000 ns 0x00000008 0b00000000000010000010000001000000 0x00000014
# 395000 ns 0x0000000c 0b00000010111010000100000000100000 0x00000014
# 405000 ns 0x00000010 0b10101101000100000000000000000000 0x00000005
# 415000 ns 0x00000014 0b0010001000010000000000000000000001 0x00000006
# 425000 ns 0x00000018 0b00001000000000000000000000000000 0xxxxxxxxx
# 435000 ns 0x00000000 0b001010100000100000000000000001010 0x00000001
# 445000 ns 0x00000004 0b0001000100000000000000000000000101 0x00000001
# 455000 ns 0x00000008 0b00000000000100000100000010000000 0x00000018
# 465000 ns 0x0000000c 0b00000010111010000100000000100000 0x00000018
# 475000 ns 0x00000010 0b10101101000100000000000000000000 0x00000006
# 485000 ns 0x00000014 0b0010001000010000000000000000000001 0x00000007
# 495000 ns 0x00000018 0b00001000000000000000000000000000 0xxxxxxxxx
# 505000 ns 0x00000000 0b001010100000100000000000000001010 0x00000001
# 515000 ns 0x00000004 0b0001000100000000000000000000000101 0x00000001
# 525000 ns 0x00000008 0b00000000000010000010000001000000 0x0000001c
# 535000 ns 0x0000000c 0b00000010111010000100000000100000 0x0000001c
# 545000 ns 0x00000010 0b10101101000100000000000000000000 0x00000007
# 555000 ns 0x00000014 0b0010001000010000000000000000000001 0x00000008
# 565000 ns 0x00000018 0b00001000000000000000000000000000 0xxxxxxxxx
# 575000 ns 0x00000000 0b001010100000100000000000000001010 0x00000001
# 585000 ns 0x00000004 0b0001000100000000000000000000000101 0x00000001
# 595000 ns 0x00000008 0b00000000000010000010000001000000 0x00000020
# 605000 ns 0x0000000c 0b00000010111010000100000000100000 0x00000020
# 615000 ns 0x00000010 0b10101101000100000000000000000000 0x00000008
# 625000 ns 0x00000014 0b0010001000010000000000000000000001 0x00000009
# 635000 ns 0x00000018 0b00001000000000000000000000000000 0xxxxxxxxx
# 645000 ns 0x00000000 0b001010100000100000000000000001010 0x00000001
# 655000 ns 0x00000004 0b0001000100000000000000000000000101 0x00000001
# 665000 ns 0x00000008 0b00000000000010000010000001000000 0x00000024

```

```

# 675000 ns 0x0000000c 0b00000010111010000100000000100000 0x00000024
# 685000 ns 0x00000010 0b10101101000100000000000000000000 0x00000009
# 695000 ns 0x00000014 0b00100010000100000000000000000000 0x0000000a
# 705000 ns 0x00000018 0b00001000000000000000000000000000 0xxxxxxxxx
# 715000 ns 0x00000000 0b001010100000100000000000000001010 0x00000000
# 725000 ns 0x00000004 0b0001000100000000000000000000000101 0x00000000
# 735000 ns 0x0000001c 0b00001000000000000000000000000111 0xxxxxxxxx
# ==== Simulation Results ====
# $s0 register (R16) final value: 0x0000000a
# DMem[0] final value: 0x00000000
# DMem[1] final value: 0x00000001
# DMem[2] final value: 0x00000002
# DMem[3] final value: 0x00000003
# DMem[4] final value: 0x00000004
# DMem[5] final value: 0x00000005
# DMem[6] final value: 0x00000006
# DMem[7] final value: 0x00000007
# DMem[8] final value: 0x00000008
# DMem[9] final value: 0x00000009
# Simulation finished.

```

## A.6 function

### A.6.1 プログラム

このプログラムは関数呼び出しとスタック操作の動作を確認するためのテストである。1 から  $n$  までの和を計算する関数を実装している。

```

.set noreorder

.global _start
_start:
    add $a0, $zero, $s0
    jal sum
    add $s1, $zero, $v0
loop:
j loop
sum:addi $sp, $sp, -8
    sw $s0, 0($sp)
    sw $s1, 4($sp)
    add $s1, $zero, $zero

```

```

    add $s0, $zero, $zero
L1: slt $t0, $s0, $a0
    beq $t0, $zero, N1
    add $s1, $s1, $s0
    addi $s1, $s1, 1
    addi $s0, $s0, 1
    j L1
N1: add $v0, $zero, $s1
    lw $s1, 4($sp)
    lw $s0, 0($sp)
    addi $sp, $sp, 8
    jr $ra

```

```

.data
a: .word 10
b: .word 20
array1: .space 40

```

#### プログラムの動作：

- メイン部分：引数を\$a0 に設定して sum 関数を呼び出し、戻り値を\$s1 に格納
- sum 関数：スタックにレジスタを退避し、1 から n までの和を計算して\$v0 に結果を返す
- 関数終了時にはスタックからレジスタを復元し、呼び出し元に戻る

#### A.6.2 テスト結果

```

run -all
# Time PC Instruction ALU_Result
# 0 ns 0x00000000 0b0000000000001000000010000000 0x0000000a
# 25000 ns 0x00000004 0b0000110000000000000000000000100 0xxxxxxxxx
# 35000 ns 0x00000010 0b00100011101111011111111111111000 0xffffffff8
# 45000 ns 0x00000014 0b10101111101100000000000000000000 0xffffffffe
# 55000 ns 0x00000018 0b101011111011000100000000000000100 0xfffffffff
# 65000 ns 0x0000001c 0b000000000000000001000100000100000 0x00000000
# 75000 ns 0x00000020 0b000000000000000001000000000100000 0x00000000
# 85000 ns 0x00000024 0b000000100000010001000000000101010 0x00000001
# 95000 ns 0x00000028 0b00010001000000000000000000000100 0x00000001
# 105000 ns 0x0000002c 0b00000010001100001000100000100000 0x00000000
# 115000 ns 0x00000030 0b001000100011000100000000000000001 0x00000001
# 125000 ns 0x00000034 0b001000100001000000000000000000001 0x00000001

```



[illegible]

```

# 535000 ns 0x00000030 0b00100010001100010000000000000001 0x00000024
# 545000 ns 0x00000034 0b00100010000100000000000000000001 0x00000008
# 555000 ns 0x00000038 0b000010000000000000000000000001001 0xxxxxxxxx
# 565000 ns 0x00000024 0b00000010000001000100000000101010 0x00000001
# 575000 ns 0x00000028 0b00010001000000000000000000000100 0x00000001
# 585000 ns 0x0000002c 0b00000010001100001000100000100000 0x0000002c
# 595000 ns 0x00000030 0b00100010001100010000000000000001 0x0000002d
# 605000 ns 0x00000034 0b00100010000100000000000000000001 0x00000009
# 615000 ns 0x00000038 0b000010000000000000000000000001001 0xxxxxxxxx
# 625000 ns 0x00000024 0b00000010000001000100000000101010 0x00000001
# 635000 ns 0x00000028 0b00010001000000000000000000000100 0x00000001
# 645000 ns 0x0000002c 0b00000010001100001000100000100000 0x00000036
# 655000 ns 0x00000030 0b00100010001100010000000000000001 0x00000037
# 665000 ns 0x00000034 0b00100010000100000000000000000001 0x0000000a
# 675000 ns 0x00000038 0b000010000000000000000000000001001 0xxxxxxxxx
# 685000 ns 0x00000024 0b00000010000001000100000000101010 0x00000000
# 695000 ns 0x00000028 0b00010001000000000000000000000100 0x00000000
# 705000 ns 0x0000003c 0b00000000000010001000100000010000 0x00000037
# 715000 ns 0x00000040 0b10001111101100010000000000000100 0xffffffff
# 725000 ns 0x00000044 0b10001111101100000000000000000000 0xfffffffffe
# 735000 ns 0x00000048 0b00100011101111010000000000001000 0x00000000
# 745000 ns 0x0000004c 0b00000011111000000000000000001000 0xxxxxxxxx
# 755000 ns 0x00000008 0b00000000000000101000100000100000 0x00000037
# 765000 ns 0x0000000c 0b0000100000000000000000000000011 0xxxxxxxxx
# ==== Simulation Results ====
# $s1 register (R17) final value: 0x00000037
# Simulation finished.

```

## A.7 recursion

### A.7.1 プログラム

このプログラムは再帰関数呼び出しの動作を確認するためのテストである。再帰を用いて 1 から n までの和を計算する。

```

.set noreorder

.global _start
_start:
    add $a0, $zero, $s0
    jal sum

```

```

    add $s1, $zero, $v0
loop:
j loop
sum:addi $sp, $sp, -8
    sw $a0, 0($sp)
    sw $ra, 4($sp)
    slti $t0, $a0, 1 # Check if a0 < 1
    beq $t0, $zero, L1 # If a0 >= 1, go to L1
    lw $ra, 4($sp)
    lw $a0, 0($sp)
    add $v0, $zero, $zero
    addi $sp, $sp, 8
    jr $ra
L1: addi $a0, $a0, -1
    jal sum
    lw $a0, 0($sp)
    lw $ra, 4($sp)
    add $v0, $v0, $a0
    addi $sp, $sp, 8
    jr $ra

.data
a: .word 10
b: .word 20
array1: .space 40

```

#### プログラムの動作：

- 引数が 1 未満の場合は 0 を返してベースケースとする
- 引数が 1 以上の場合は、引数を 1 減らして再帰呼び出しを行い、その結果に現在の引数値を加算
- 各再帰レベルでスタックに引数と戻りアドレスを保存・復元

#### A.7.2 テスト結果

```

run -all
# Time PC Instruction ALU_Result
# 0 ns 0x00000000 0b0000000000001000000010000000 0x0000000a
# 25000 ns 0x00000004 0b000011000000000000000000000000 0xxxxxxx
# 35000 ns 0x00000010 0b00100011101111011111111111111000 0xffffffff8

```

```

# 45000 ns 0x00000014 0b10101111101001000000000000000000 0xfffffffffe
# 55000 ns 0x00000018 0b10101111101111110000000000000100 0xffffffffff
# 65000 ns 0x0000001c 0b00101000100010000000000000000001 0x00000000
# 75000 ns 0x00000020 0b00010001000000000000000000000101 0x00000000
# 85000 ns 0x00000038 0b00100000100001001111111111111111 0x00000009
# 95000 ns 0x0000003c 0b00001100000000000000000000000100 0xxxxxxxxx
# 105000 ns 0x00000010 0b00100011101111011111111111111000 0xfffffffff0
# 115000 ns 0x00000014 0b10101111101001000000000000000000 0xfffffffffc
# 125000 ns 0x00000018 0b10101111101111110000000000000100 0xfffffffffd
# 135000 ns 0x0000001c 0b00101000100010000000000000000001 0x00000000
# 145000 ns 0x00000020 0b00010001000000000000000000000101 0x00000000
# 155000 ns 0x00000038 0b00100000100001001111111111111111 0x00000008
# 165000 ns 0x0000003c 0b00001100000000000000000000000100 0xxxxxxxxx
# 175000 ns 0x00000010 0b00100011101111011111111111111000 0xffffffffe8
# 185000 ns 0x00000014 0b10101111101001000000000000000000 0xfffffffffa
# 195000 ns 0x00000018 0b10101111101111110000000000000100 0xfffffffffb
# 205000 ns 0x0000001c 0b00101000100010000000000000000001 0x00000000
# 215000 ns 0x00000020 0b00010001000000000000000000000101 0x00000000
# 225000 ns 0x00000038 0b00100000100001001111111111111111 0x00000007
# 235000 ns 0x0000003c 0b00001100000000000000000000000100 0xxxxxxxxx
# 245000 ns 0x00000010 0b00100011101111011111111111111000 0xffffffffe0
# 255000 ns 0x00000014 0b10101111101001000000000000000000 0xfffffffff8
# 265000 ns 0x00000018 0b10101111101111110000000000000100 0xfffffffff9
# 275000 ns 0x0000001c 0b00101000100010000000000000000001 0x00000000
# 285000 ns 0x00000020 0b00010001000000000000000000000101 0x00000000
# 295000 ns 0x00000038 0b00100000100001001111111111111111 0x00000006
# 305000 ns 0x0000003c 0b00001100000000000000000000000100 0xxxxxxxxx
# 315000 ns 0x00000010 0b00100011101111011111111111111000 0xffffffffd8
# 325000 ns 0x00000014 0b10101111101001000000000000000000 0xfffffffff6
# 335000 ns 0x00000018 0b10101111101111110000000000000100 0xfffffffff7
# 345000 ns 0x0000001c 0b00101000100010000000000000000001 0x00000000
# 355000 ns 0x00000020 0b00010001000000000000000000000101 0x00000000
# 365000 ns 0x00000038 0b00100000100001001111111111111111 0x00000005
# 375000 ns 0x0000003c 0b00001100000000000000000000000100 0xxxxxxxxx
# 385000 ns 0x00000010 0b00100011101111011111111111111000 0xffffffffd0
# 395000 ns 0x00000014 0b10101111101001000000000000000000 0xfffffffff4
# 405000 ns 0x00000018 0b10101111101111110000000000000100 0xfffffffff5
# 415000 ns 0x0000001c 0b00101000100010000000000000000001 0x00000000
# 425000 ns 0x00000020 0b00010001000000000000000000000101 0x00000000
# 435000 ns 0x00000038 0b00100000100001001111111111111111 0x00000004

```

[illegible]

```

# 845000 ns 0x00000044 0b100011111011111100000000000000100 0xffffffffed
# 855000 ns 0x00000048 0b00000000010001000001000000100000 0x00000001
# 865000 ns 0x0000004c 0b00100011101111010000000000001000 0xffffffffb8
# 875000 ns 0x00000050 0b00000011111000000000000000001000 0xxxxxxxxx
# 885000 ns 0x00000040 0b10001111101001000000000000000000 0xffffffffee
# 895000 ns 0x00000044 0b10001111101111110000000000000100 0xffffffffef
# 905000 ns 0x00000048 0b00000000010001000001000000100000 0x00000003
# 915000 ns 0x0000004c 0b00100011101111010000000000001000 0xffffffffc0
# 925000 ns 0x00000050 0b00000011111000000000000000001000 0xxxxxxxxx
# 935000 ns 0x00000040 0b10001111101001000000000000000000 0xfffffffff0
# 945000 ns 0x00000044 0b10001111101111110000000000000100 0xfffffffff1
# 955000 ns 0x00000048 0b00000000010001000001000000100000 0x00000006
# 965000 ns 0x0000004c 0b00100011101111010000000000001000 0xffffffffc8
# 975000 ns 0x00000050 0b00000011111000000000000000001000 0xxxxxxxxx
# 985000 ns 0x00000040 0b10001111101001000000000000000000 0xfffffffff2
# 995000 ns 0x00000044 0b10001111101111110000000000000100 0xfffffffff3
# 1005000 ns 0x00000048 0b00000000010001000001000000100000 0x0000000a
# 1015000 ns 0x0000004c 0b00100011101111010000000000001000 0xffffffffd0
# 1025000 ns 0x00000050 0b00000011111000000000000000001000 0xxxxxxxxx
# 1035000 ns 0x00000040 0b10001111101001000000000000000000 0xfffffffff4
# 1045000 ns 0x00000044 0b10001111101111110000000000000100 0xfffffffff5
# 1055000 ns 0x00000048 0b00000000010001000001000000100000 0x0000000f
# 1065000 ns 0x0000004c 0b00100011101111010000000000001000 0xffffffffd8
# 1075000 ns 0x00000050 0b00000011111000000000000000001000 0xxxxxxxxx
# 1085000 ns 0x00000040 0b10001111101001000000000000000000 0xfffffffff6
# 1095000 ns 0x00000044 0b10001111101111110000000000000100 0xfffffffff7
# 1105000 ns 0x00000048 0b00000000010001000001000000100000 0x00000015
# 1115000 ns 0x0000004c 0b00100011101111010000000000001000 0xffffffffe0
# 1125000 ns 0x00000050 0b00000011111000000000000000001000 0xxxxxxxxx
# 1135000 ns 0x00000040 0b10001111101001000000000000000000 0xfffffffff8
# 1145000 ns 0x00000044 0b10001111101111110000000000000100 0xfffffffff9
# 1155000 ns 0x00000048 0b00000000010001000001000000100000 0x0000001c
# 1165000 ns 0x0000004c 0b00100011101111010000000000001000 0xffffffffe8
# 1175000 ns 0x00000050 0b00000011111000000000000000001000 0xxxxxxxxx
# 1185000 ns 0x00000040 0b10001111101001000000000000000000 0xfffffffffa
# 1195000 ns 0x00000044 0b10001111101111110000000000000100 0xfffffffffb
# 1205000 ns 0x00000048 0b00000000010001000001000000100000 0x00000024
# 1215000 ns 0x0000004c 0b00100011101111010000000000001000 0xfffffffff0
# 1225000 ns 0x00000050 0b00000011111000000000000000001000 0xxxxxxxxx
# 1235000 ns 0x00000040 0b10001111101001000000000000000000 0xfffffffffc

```

```

# 1245000 ns 0x00000044 0b100011111011111100000000000000100 0xffffffffd
# 1255000 ns 0x00000048 0b00000000010001000001000000100000 0x0000002d
# 1265000 ns 0x0000004c 0b00100011101111010000000000001000 0xfffffffff8
# 1275000 ns 0x00000050 0b0000000111110000000000000000001000 0xxxxxxxxx
# 1285000 ns 0x00000040 0b1000111110100100000000000000000000 0xfffffffffe
# 1295000 ns 0x00000044 0b100011111011111100000000000000100 0xffffffffff
# 1305000 ns 0x00000048 0b00000000010001000001000000100000 0x00000037
# 1315000 ns 0x0000004c 0b00100011101111010000000000001000 0x00000000
# 1325000 ns 0x00000050 0b0000000111110000000000000000001000 0xxxxxxxxx
# 1335000 ns 0x00000008 0b000000000000000101000100000100000 0x00000037
# 1345000 ns 0x0000000c 0b0000100000000000000000000000000011 0xxxxxxxxx
# ==== Simulation Results ====
# $s1 register (R17) final value: 0x00000037
# Simulation finished.

```

## A.8 hanoi

### A.8.1 プログラム

このプログラムはハノイの塔を解く再帰アルゴリズムを実装し、複雑な再帰処理の動作を確認するためのテストである。

```

.set noreorder

.global _start
_start:
    # init
    addi $a0, $zero, 3 # $a0 = $zero + 3, n
    addi $a1, $zero, 3 # $a1 = $zero + 1, from
    addi $a2, $zero, 0 # $a2 = $zero + 2, to
    addi    $a3, $zero, 0          # $a3 = $zero + 0, aux
    addi    $t1, $zero, 0          # for count
    # call
    jal hanoi # jump to hanoi and save position to $ra

loop:
    j loop # jump to loop
hanoi:
    addi $sp, $sp, -20 # $sp = $sp + -20
    sw $a0, 0($sp)
    sw    $a1, 4($sp)

```

```

sw      $a2, 8($sp)
sw      $a3, 12($sp)
sw      $ra, 16($sp)
slti $t0, $a0, 2 # $t0 = ($a0 < 2) ? 1 : 0
beq $t0, $zero, L1 # if $t0 == $zero then goto L1
add $a2, $a3, $zero # $a2 = $a3 + $zero
addi $t1, $t1, 1 # $t1 = $t1 + 1
lw $a2, 8($sp)
lw $ra, 16($sp)
addi $sp, $sp, 20 # $sp = $sp + 20
jr $ra # jump to $ra
L1:
addi $a0, $a0, -1 # $a0 = $a0 + -1
lw $a2, 12($sp)
lw $a3, 8($sp)
jal hanoi # jump to hanoi and save position to $ra
lw $a0, 0($sp)
addi $t1, $t1, 1 # $t1 = $t1 + 1
addi $a0, $a0, -1 # $a0 = $a0 + -1
lw $a1, 8($sp)
lw $a2, 4($sp)
lw $a3, 12($sp)
jal hanoi # jump to hanoi and save position to $ra
lw $a0, 0($sp)
lw $a1, 4($sp)
lw $a2, 8($sp)
lw $a3, 12($sp)
lw $ra, 16($sp)
addi $sp, $sp, 20 # $sp = $sp + 20
jr $ra # jump to $ra

```

#### プログラムの動作：

- 3 枚の円盤のハノイの塔問題を解く
- \$a0: 円盤の枚数、\$a1: 移動元、\$a2: 移動先、\$a3: 補助杆
- \$t1: 移動回数カウンタ（期待値：7 回）
- 再帰の深さに応じてスタックに多くの引数と戻りアドレスを保存
- ベースケース（円盤が 1 枚）では直接移動を実行



## A.8.2 テスト結果

run -all

# Time PC Instruction ALU\_Result

```
# 0 ns 0x00000000 0b00100000000001000000000000000011 0x00000003
# 25000 ns 0x00000004 0b00100000000001010000000000000011 0x00000003
# 35000 ns 0x00000008 0b00100000000001100000000000000000 0x00000000
# 45000 ns 0x0000000c 0b00100000000001110000000000000000 0x00000000
# 55000 ns 0x00000010 0b00100000000001001000000000000000 0x00000000
# 65000 ns 0x00000014 0b000011000000000000000000000000111 0xxxxxxxxx
# 75000 ns 0x0000001c 0b00100011101111011111111111101100 0xffffffffec
# 85000 ns 0x00000020 0b10101111101001000000000000000000 0xfffffffffb
# 95000 ns 0x00000024 0b101011111010010100000000000000100 0xfffffffffc
# 105000 ns 0x00000028 0b1010111110100110000000000000001000 0xfffffffffd
# 115000 ns 0x0000002c 0b1010111110100111000000000000001100 0xfffffffffe
# 125000 ns 0x00000030 0b101011111011111100000000000010000 0xffffffffff
# 135000 ns 0x00000034 0b001010001000100000000000000000010 0x00000000
# 145000 ns 0x00000038 0b000100010000000000000000000000110 0x00000000
# 155000 ns 0x00000054 0b00100000100001001111111111111111 0x00000002
# 165000 ns 0x00000058 0b1000111110100110000000000000001100 0xfffffffffe
# 175000 ns 0x0000005c 0b1000111110100111000000000000001000 0xfffffffffd
# 185000 ns 0x00000060 0b000011000000000000000000000000111 0xxxxxxxxx
# 195000 ns 0x0000001c 0b00100011101111011111111111101100 0xffffffffd8
# 205000 ns 0x00000020 0b10101111101001000000000000000000 0xfffffffff6
# 215000 ns 0x00000024 0b101011111010010100000000000000100 0xfffffffff7
# 225000 ns 0x00000028 0b1010111110100110000000000000001000 0xfffffffff8
# 235000 ns 0x0000002c 0b1010111110100111000000000000001100 0xfffffffff9
# 245000 ns 0x00000030 0b101011111011111100000000000010000 0xfffffffffa
# 255000 ns 0x00000034 0b001010001000100000000000000000010 0x00000000
# 265000 ns 0x00000038 0b0001000100000000000000000000000110 0x00000000
# 275000 ns 0x00000054 0b00100000100001001111111111111111 0x00000001
# 285000 ns 0x00000058 0b1000111110100110000000000000001100 0xfffffffff9
# 295000 ns 0x0000005c 0b1000111110100111000000000000001000 0xfffffffff8
# 305000 ns 0x00000060 0b000011000000000000000000000000111 0xxxxxxxxx
# 315000 ns 0x0000001c 0b00100011101111101111111111101100 0xffffffffc4
# 325000 ns 0x00000020 0b10101111101001000000000000000000 0xfffffffff1
# 335000 ns 0x00000024 0b101011111010010100000000000000100 0xfffffffff2
# 345000 ns 0x00000028 0b1010111110100110000000000000001000 0xfffffffff3
# 355000 ns 0x0000002c 0b1010111110100111000000000000001100 0xfffffffff4
# 365000 ns 0x00000030 0b101011111011111100000000000010000 0xfffffffff5
```

```

# 375000 ns 0x00000034 0b00101000100010000000000000000010 0x00000001
# 385000 ns 0x00000038 0b000100010000000000000000000000110 0x00000001
# 395000 ns 0x0000003c 0b00000000111000000011000000100000 0x00000000
# 405000 ns 0x00000040 0b00100001001010010000000000000001 0x00000001
# 415000 ns 0x00000044 0b10001111101001100000000000001000 0xfffffffff3
# 425000 ns 0x00000048 0b10001111101111110000000000010000 0xfffffffff5
# 435000 ns 0x0000004c 0b001000111011111010000000000010100 0xfffffffffd8
# 445000 ns 0x00000050 0b00000011111000000000000000001000 0xxxxxxxxxx
# 455000 ns 0x00000064 0b10001111101001000000000000000000 0xfffffffff6
# 465000 ns 0x00000068 0b00100001001010010000000000000001 0x00000002
# 475000 ns 0x0000006c 0b00100000100001001111111111111111 0x00000001
# 485000 ns 0x00000070 0b10001111101001010000000000001000 0xfffffffff8
# 495000 ns 0x00000074 0b10001111101001100000000000000100 0xfffffffff7
# 505000 ns 0x00000078 0b10001111101001110000000000001100 0xfffffffff9
# 515000 ns 0x0000007c 0b0000110000000000000000000000111 0xxxxxxxxxx
# 525000 ns 0x0000001c 0b00100011101111011111111111101100 0xfffffffffc4
# 535000 ns 0x00000020 0b10101111101001000000000000000000 0xfffffffff1
# 545000 ns 0x00000024 0b10101111101001010000000000000100 0xfffffffff2
# 555000 ns 0x00000028 0b10101111101001100000000000001000 0xfffffffff3
# 565000 ns 0x0000002c 0b10101111101001110000000000001100 0xfffffffff4
# 575000 ns 0x00000030 0b10101111101111110000000000010000 0xfffffffff5
# 585000 ns 0x00000034 0b0010100010001000000000000000010 0x00000001
# 595000 ns 0x00000038 0b00010001000000000000000000000110 0x00000001
# 605000 ns 0x0000003c 0b00000000111000000011000000100000 0x00000000
# 615000 ns 0x00000040 0b00100001001010010000000000000001 0x00000003
# 625000 ns 0x00000044 0b10001111101001100000000000001000 0xfffffffff3
# 635000 ns 0x00000048 0b10001111101111110000000000010000 0xfffffffff5
# 645000 ns 0x0000004c 0b001000111011111010000000000010100 0xfffffffffd8
# 655000 ns 0x00000050 0b00000011111000000000000000001000 0xxxxxxxxxx
# 665000 ns 0x00000080 0b10001111101001000000000000000000 0xfffffffff6
# 675000 ns 0x00000084 0b10001111101001010000000000000100 0xfffffffff7
# 685000 ns 0x00000088 0b10001111101001100000000000001000 0xfffffffff8
# 695000 ns 0x0000008c 0b10001111101001110000000000001100 0xfffffffff9
# 705000 ns 0x00000090 0b10001111101111110000000000010000 0xfffffffffa
# 715000 ns 0x00000094 0b001000111011111010000000000010100 0xfffffffffec
# 725000 ns 0x00000098 0b00000011111000000000000000001000 0xxxxxxxxxx
# 735000 ns 0x00000064 0b10001111101001000000000000000000 0xfffffffffb
# 745000 ns 0x00000068 0b00100001001010010000000000000001 0x00000004
# 755000 ns 0x0000006c 0b00100000100001001111111111111111 0x00000002
# 765000 ns 0x00000070 0b10001111101001010000000000001000 0xfffffffffd

```

```

# 775000 ns 0x00000074 0b10001111101001100000000000000100 0xffffffffc
# 785000 ns 0x00000078 0b100011111010011100000000000001100 0xfffffffffe
# 795000 ns 0x0000007c 0b00001100000000000000000000000111 0xxxxxxxx
# 805000 ns 0x0000001c 0b0010001110111101111111111101100 0xffffffffd8
# 815000 ns 0x00000020 0b10101111101001000000000000000000 0xfffffffff6
# 825000 ns 0x00000024 0b10101111101001010000000000000100 0xfffffffff7
# 835000 ns 0x00000028 0b101011111010011000000000000001000 0xfffffffff8
# 845000 ns 0x0000002c 0b101011111010011100000000000001100 0xfffffffff9
# 855000 ns 0x00000030 0b1010111110111111000000000000010000 0xfffffffffa
# 865000 ns 0x00000034 0b001010001000100000000000000000010 0x00000000
# 875000 ns 0x00000038 0b0001000100000000000000000000000110 0x00000000
# 885000 ns 0x00000054 0b0010000010000100111111111111111 0x00000001
# 895000 ns 0x00000058 0b100011111010011000000000000001100 0xfffffffff9
# 905000 ns 0x0000005c 0b100011111010011100000000000001000 0xfffffffff8
# 915000 ns 0x00000060 0b00001100000000000000000000000111 0xxxxxxxx
# 925000 ns 0x0000001c 0b0010001110111101111111111101100 0xffffffffc4
# 935000 ns 0x00000020 0b10101111101001000000000000000000 0xfffffffff1
# 945000 ns 0x00000024 0b10101111101001010000000000000100 0xfffffffff2
# 955000 ns 0x00000028 0b101011111010011000000000000001000 0xfffffffff3
# 965000 ns 0x0000002c 0b101011111010011100000000000001100 0xfffffffff4
# 975000 ns 0x00000030 0b1010111110111111000000000000010000 0xfffffffff5
# 985000 ns 0x00000034 0b001010001000100000000000000000010 0x00000001
# 995000 ns 0x00000038 0b0001000100000000000000000000000110 0x00000001
# 1005000 ns 0x0000003c 0b00000000111000000011000000100000 0x00000003
# 1015000 ns 0x00000040 0b00100001001010010000000000000001 0x00000005
# 1025000 ns 0x00000044 0b100011111010011000000000000001000 0xfffffffff3
# 1035000 ns 0x00000048 0b1000111110111111000000000000010000 0xfffffffff5
# 1045000 ns 0x0000004c 0b0010001110111101000000000000010100 0xffffffffd8
# 1055000 ns 0x00000050 0b00000001111100000000000000000001000 0xxxxxxxx
# 1065000 ns 0x00000064 0b10001111101001000000000000000000 0xfffffffff6
# 1075000 ns 0x00000068 0b00100001001010010000000000000001 0x00000006
# 1085000 ns 0x0000006c 0b0010000010000100111111111111111 0x00000001
# 1095000 ns 0x00000070 0b100011111010010100000000000001000 0xfffffffff8
# 1105000 ns 0x00000074 0b10001111101001100000000000000100 0xfffffffff7
# 1115000 ns 0x00000078 0b100011111010011100000000000001100 0xfffffffff9
# 1125000 ns 0x0000007c 0b00001100000000000000000000000111 0xxxxxxxx
# 1135000 ns 0x0000001c 0b0010001110111101111111111101100 0xffffffffc4
# 1145000 ns 0x00000020 0b10101111101001000000000000000000 0xfffffffff1
# 1155000 ns 0x00000024 0b10101111101001010000000000000100 0xfffffffff2
# 1165000 ns 0x00000028 0b101011111010011000000000000001000 0xfffffffff3

```

```

# 1175000 ns 0x0000002c 0b10101111101001110000000000001100 0xfffffffff4
# 1185000 ns 0x00000030 0b101011111011111100000000000010000 0xfffffffff5
# 1195000 ns 0x00000034 0b00101000100010000000000000000010 0x000000001
# 1205000 ns 0x00000038 0b000100010000000000000000000000110 0x000000001
# 1215000 ns 0x0000003c 0b00000000111000000011000000100000 0x000000000
# 1225000 ns 0x00000040 0b001000010010100100000000000000001 0x000000007
# 1235000 ns 0x00000044 0b1000111110100110000000000000001000 0xfffffffff3
# 1245000 ns 0x00000048 0b100011111011111100000000000010000 0xfffffffff5
# 1255000 ns 0x0000004c 0b001000111011110100000000000010100 0xfffffffffd8
# 1265000 ns 0x00000050 0b0000001111100000000000000000001000 0xxxxxxxxxx
# 1275000 ns 0x00000080 0b100011111010010000000000000000000 0xfffffffff6
# 1285000 ns 0x00000084 0b100011111010010100000000000000100 0xfffffffff7
# 1295000 ns 0x00000088 0b1000111110100110000000000000001000 0xfffffffff8
# 1305000 ns 0x0000008c 0b1000111110100111000000000000001100 0xfffffffff9
# 1315000 ns 0x00000090 0b100011111011111100000000000010000 0xfffffffffa
# 1325000 ns 0x00000094 0b001000111011110100000000000010100 0xfffffffffec
# 1335000 ns 0x00000098 0b0000001111100000000000000000001000 0xxxxxxxxxx
# 1345000 ns 0x00000080 0b100011111010010000000000000000000 0xfffffffffb
# 1355000 ns 0x00000084 0b100011111010010100000000000000100 0xfffffffffc
# 1365000 ns 0x00000088 0b1000111110100110000000000000001000 0xfffffffffd
# 1375000 ns 0x0000008c 0b1000111110100111000000000000001100 0xfffffffffe
# 1385000 ns 0x00000090 0b100011111011111100000000000010000 0xffffffffff
# 1395000 ns 0x00000094 0b001000111011110100000000000010100 0x000000000
# 1405000 ns 0x00000098 0b0000001111100000000000000000001000 0xxxxxxxxxx
# 1415000 ns 0x00000018 0b000010000000000000000000000000110 0xxxxxxxxxx
# ==== Simulation Results ====
# $t1 register (R9) final value: 0x00000007
# Simulation finished.

```

## 参考文献

- [1] 成瀬正. コンピュータアーキテクチャ. 森北出版, 第1版, 2016.
- [2] Cpulator mips system simulator. <https://cpulator.01xz.net/?sys=mipsr5b>. Accessed: 2025-08-16.