情報工学実験 A（ハードウェア）報告書

学生番号: 09430509
提出者: 今田　将也
E-mail: psc06fz8@s.okayama-u.ac.jp

提出日: 2020 年 6 月 11 日（木）
締切日: 2020 年 6 月 11 日（木）12:00

概 要

本稿では，情報工学実験 A（ハードウェア実験）において作成した 32 ビットマイクロプロセッサ p32 の設計についてまとめる．

1　はじめに

本実験では，ハードウェア記述言語と CAD（computer aided design）ツールを利用したマイクロプロセッサ設計を通して，論理回路，コンピュータアーキテクチャ，およびコンピュータシステムに関する理解を深めることを目的とする．

本報告書では，ハードウェア記述言語と CAD ツールを用いた論理回路の設計と，32 ビットマイクロプロセッサの設計について報告する．本報告書の構成は次のとおりである．

第 2 章 本実験で設計したプロセッサの概要について述べる．

第 3 章 本実験における実施内容の状況報告を行う

第 4 章 アセンブリで作成したプログラミング課題に関しての報告を行う．

第 5 章 発展課題で取り組んだ課題について報告する．

第 6 章 諸事項について検討を行い，それについての考察を記述する．

第 7 章 設計の際に工夫した点や特に注力した点について報告する．

第 8 章 本実験を通して，これまでから一層理解が深まった部分などについて記述する．

第 9 章 本報告のまとめと今後の課題を述べる．

第 10 章 本実験で作成したアセンブリ言語プログラム，設計した SFL 記述，テスト用スクリプト，テスト結果，論理合成時の出力等のファイルの一覧を掲載する．

2　設計したプロセッサの概要

本実験で設計する 32 ビット RISC マイクロプロセッサ p32m1，p32m2 および，p32p1 の FSL 記述の概要について述べる．

p32m1 単純なマルチサイクルで実装されるプロセッサ．単一サイクルで１命令を５サイクルで行う．パイプライン実行は無い．

p32m2 1 命令を 2〜5 サイクルで実行するマルチサイクルのプロセッサ．パイプライン実行は無い．

p32p1 パイプライン実行により実装されたプロセッサ．5 ステージパイプラインとフォワーディング機構を持つ．

p32m1，p32m2 のプロセッサは 1 つの命令が終了するまで次の命令を実行せず，その命令が終了してから次の命令を実行するような方式になっている。

p32p1 はパイプライン実行を考慮した設計を行う．今回設計するプロセッサは 5 つのステージを持つため，同時で最大 5 命令実行することになる．

本プロセッサで使用するサブモジュールは、以下である。

- レジスタファイル
  - regs32x32（32 ビット x32 本の汎用レジスタ）
  - 入力は 1 ポート
  - 出力は 2 ポート
- 演算ユニット
  - p32ExecUnit
  - マイクロプロセッサ p32 用の実行ユニットモジュール
- デコードユニット
  - p32DecodeUnit
  - 命令をデコードし，レジスタファイルにアクセスして必要なレジスタを読み取る．処理すべき命令は IF から受け取り，その命令から命令コードとオペランドを展開し，必要に応じてレジスタの内容を取り出す．
- alu32（32 ビット ALU）
  - 32 ビットの算術・論理演算
- shift32（32 ビットシフタ）
  - 論理シフト・算術シフト
- add32（32 ビット加算器）
  - 全加算器

命令形式には R 形式，I 形式，J 形式がある．

また，遅延ロードがある．例えば，ロードされるデータはロード命令の MEM ステージが終わるまで使用できない．ステージについては以降で説明している．なお，分岐命令についても，ID ステージで計算するため，ID ステージが終わるまで利用できない．

2.1 実装した基本命令セット

本実験で実装した基本命令セットを表 1 に示す．

- 加算命令　add, addu, addi, addiu

- 減算命令　sub, subu
- 論理積演算　and,andi
- 論理和演算　or,ori
- 排他的論理和演算　xor,xori
- 否定演算　nor
- 比較命令

  slt,slti 符号付き３２ビット整数を比較する Rs レジスタが Rt レジスタより小さい場合に 1

  sltu, sltiu 符号無し３２ビット整数を比較する Rs レジスタが Rt レジスタより小さい場合に 1

- シフト命令

  sll,sllv 0 を下位ビットに挿入し shamt ビットもしくは指定レジスタの下位５ビットで指定されるビットだけ左シフト

  srl,srlv 0 を上位ビットに挿入し shamt ビットもしくは指定レジスタの下位５ビットで指定されるビットだけ右シフト

  sra,srav 上位ビットを符号拡張し shamt ビットもしくは指定レジスタの下位５ビットで指定されるビットだけ右シフト

- ロードストア命令

  lw, lb 有効アドレスで指定するメモリ位置のワードもしくはバイトを読み出す

  sw, sb 有効アドレスで指定するメモリ位置のワードもしくはバイトを書き出す

- 分岐命令

  beq 比較対象の２つのレジスタの内容の値が等しい場合に１命令遅れてターゲットアドレスに分岐

  bne 比較対象の２つのレジスタの内容の値が等しくない場合に１命令遅れてターゲットアドレスに分岐

- ジャンプ命令

  j 26 ビットターゲット・アドレスを 2 ビット左詰めして，現在のプログラム・カウンタの上位 4 ビットと合成しジャンプ先のアドレスを計算．計算されたアドレスに 1 命令遅れで無条件に分岐

  jr 1 命令遅れで汎用レジスタ rs に格納されたアドレスへ無条件で分岐

  jal 26 ビットターゲット・アドレスを 2 ビット左詰めして，現在のプログラム・カウンタの上位 4 ビットと合成しジャンプ先のアドレスを計算．計算されたアドレスに 1 命令遅れで無条件に分岐．遅延スロットの後の命令のアドレスはリンクレジスタに格納

  jalr 1 命令遅れで汎用レジスタ rs に格納されたアドレスへ無条件で分岐．遅延スロットの後の命令のアドレスを汎用レジスタ rd に格納

- システムコール

  syscall システムコール・トラップを発生させ，制御を無条件で例外ハンドラに移す

表 1: 命令一覧

命令種別	実装した命令
加算命令	add, addu, addi, addiu
減算命令	sub, subu
論理演算命令	and,andi, or,ori, xor, xori, nor
比較	slt, slti, sltu, sltiu
シフト	sll,sllv, srl,srlv, sra, srav
ロード・ストア	lw, sw, lb, sb
分岐	beq, bne
ジャンプ	j, jr, jal, jalr
例外，システムコール	syscall

2.2 内部構造の概略

5 ステージパイプラインおよびマルチサイクル構造は以下のステージにより構成されている．

- IF ステージ: 命令メモリから命令をフェッチ
- ID ステージ: レジスタの値を取り出したり，分岐処理を行う
- EX ステージ: 演算を実行
- MEM ステージ: メモリアクセス関連の処理を行う
- WB ステージ: レジスタに書き込む

2.3 処理方式の概略

単一サイクル ５ステージに分かれているが，マルチサイクルとは異なり命令に必要なクロック数は５サイクルで変わらない

マルチサイクル マルチサイクル方式のマイクロプロセッサは 1 命令を複数のステップに分け，各ステップを 1 クロックサイクルで実現し，５ステージに分かれているステップを繰り返しながら行う方式である．命令に必要なサイクル数が変わる．

パイプライン方式 乗除算命令を除くすべての命令を５サイクルで並列に実行 (IF/ID/EX/MEM/WB) する．毎サイクル毎に命令をフェッチし 1 サイクル当たり 1 命令を実行する．依存関係のある命令が同時刻にパイプライン内に存在する場合，先行命令の演算結果を利用するには，先行命令が WB ステージにてレジスタファイルに結果を書き込むまで利用できないことを，データフォワーディング機構により解決している

データフォワーディングとは，パイプライン処理において次のクロックサイクルで次の命令を実行できないデータハザードを解消するための，演算結果やメモリからロードした値がレジスタファイルに書き込まれるタイミング以前で利用できるように，必要な値を ID ステージより後段のステージ（EX/MEM/WB）からバイパスする機構のことである．

3　実施状況の報告

どの課題に取り組んだのか，またその状況について，表 2 にまとめて報告する．

表 2: プログラミング課題，設計課題および発展課題の実施状況

課題	状況
（プログラミング課題）	
1.【プログラミング課題１】N 個の語の加算	(2) 完了
2.【プログラミング課題２】N 語のメモリコピー	(2) 完了
3.【プログラミング課題３】乗算	(2) 完了
（設計課題２）	
4.【設計課題２－１】３２ビット加算器　add32	(2) 完了
5.【設計課題２－２】３２ビット ALU　alu32	(2) 完了
6.【設計課題２－３】３２ビットシフタ shift32	(2) 完了
（発展課題２）	
7.【発展課題２－１】３２ビット整数乗算器　mult32	(2) 完了
8.【発展課題２－２】３２ビット整数除算器　div32	(0) 未実施
（設計課題３）	
9.【設計課題３－１】レジスタファイル　regs32x32	(2) 完了
１０.【設計課題３－２】実行ユニット　p32ExecUnit	(2) 完了
１１.【設計課題３－３】デコードユニット p32DecodeUnit	(2) 完了
（設計課題４）	
１２.【設計課題４－１】プロセッサ　p32m1	(2) 完了
１３.【設計課題４－２】プロセッサ　p32m2	(2) 完了
１４.【設計課題４－３】プロセッサ p32p1	(2) 完了
（発展課題４）	
１２.【発展課題４－１】改良	(2) 完了
１３.【発展課題４－２】乗算機能の実装	(0) 未実施

4　課題に関する報告

4.1　プログラミング課題に関する報告

アセンブリ言語でプログラミングする際，遅延スロットに注意した．

- 遅延ロード
  - ロード命令は，ロードされるデータを別の命令が使用できるようになるまでに 1 サイクルの遅延（待ち時間）がある
- 遅延分岐
  - ジャンプ命令および分岐命令は，分岐を実行する場合には命令とターゲットアドレスを取り出す間，1 サイクルの遅延がある

この問題は，nop 命令を差し込むことで遅延を処理した．

4.1.1 プログラミング課題１

データセグメントのアドレス 0x10004100 から置かれた N 個のワードデータの加算を行なう p32 アセンブリ言語のプログラムを作成し，MAPS シミュレータにて動作を確認した．

2 通りの方法を実施した，１つ目は main に直接書くバージョンである．ここでは，データの個数，データの格納されている先頭アドレス及び計算結果格納用のレジスタを用意して，それぞれに初期化を行い，LOOP 処理を実装した．MAPS では，run を行ったあと，0x10004000 番地を print することで結果を確認できる．

２つ目は，サブルーチンとして呼び出す方法である．main で行う方法と特に変わることは，MIPS のときの move が使えなかったため add 命令を用いて値をゼロと加算することで擬似的に移動したようにして実装をした．

4.1.2 プログラミング課題２

アドレス src から始まる N ワードのデータをアドレス dest から始まる領域にコピーする C 言語の以下の関数に対応するアセンブリ言語プログラムを作成し，MAPS シミュレータで動作を確認した．

```
int *memcopy(int *dest, int *src, int n);
```

まず dest の値，src の値，n の値を読み出す．LOOP 処理で加算するアドレスを 4 ずつ追加して，dest と src をそれぞれ１バイトずつずらしながらワードを読み込んで繰り返す．最後に dest を返して呼び出し元に返している．

4.1.3 プログラミング課題３

2 個の 32 ビットの無符号整数値を受取り，それらの積を計算し， 64 ビットの乗算結果を返すサブルーチンを作成した．サブルーチンへの引数は被乗数を`$a0` で，乗数を`$a1` でそれぞれ与え，結果は，上位ワードを`$v0` に，下位ワードを`$v1` に格納している．なお，作成したプログラムは MAPS シミュレータにて動作確認を行った．

動作内容としてはまず，かけられる数とかける数を用意した後に，かける数の最下位ビットが１かどうか判定する．１ならば計算を行うので，結果のレジスタに足し込む．その後結果の上位ワードを１ビット右シフトさせる．その後，規定の回数分のループが終了していなければ，結果の下位ビットとかける数を右シフトさせる．乗数（例えば 10, 2 進数で 1010）のビットが立っているとき，被乗数のを桁を揃えて（これがシフトに相当）足し込んでいく．乗数の各ビットを LSB 側（最下位ビット側）からみて，ビットがたっていたら（ビットが 1 なら） 被乗数を足し込んでいく繰り返しの処理を行っている．

4.2 プロセッサ設計課題に関する報告

マイクロプロセッサ p32 のモジュールとなる p32m1，p32m2，p32p1 の設計を行った．

設計に際しての動作確認については，gitbucket 上に配布されているブランチをフォークし，verilog, vvp, sim については MAPS を用いて，シミュレーション結果と比較用パターンの差分を確認し，差分がないことを確認した．

そして，各プロセッサを FPGA 向けに論理合成，配置配線，および静的タイミング解析を行なった．このプロセッサは後に改良を行ったため，ここでは改良前の諸量を表 3 にまとめる．

表 3: FPGA への論理合成などで得られた諸量のまとめ

	最大動作周波数	Fmax（MHz）			
モジュール	85 ℃ Model	0 ℃ Model	LE 数（使用率）	CF 数（使用率）	レジスタ数（使用率）
プロセッサ p32m1	61.36	67.40	3680（3 %）	3631（3 %）	1379（1 %）
プロセッサ p32m2	62.19	68.04	3719（3 %）	3687（3 %）	1379（1 %）
プロセッサ p32p1	64.88	71.15	4052（4 %）	4025（4 %）	1416（1 %）

4.2.1 考察

他の人との結果と比較しても特に特筆して優れていた点，及び低い点は見当たらなかった．add32 のサブモジュールや，shift32 のサブモジュールは，算術演算子及び論理演算による実装が可能であるため，差が出るとすればそこだと考えプロセッサ設計に予めふくまれてソースコードに則った部分の改良よりも，サブモジュールの改良を行った．

具体的には，１つ目は算術演算子により実装していた shift32 サブモジュールを，論理演算によるシフト演算へ変更を行った．２つ目に，算術演算子により実装していた add32 加算器のサブモジュールは，論理演算に落とし込む事ができるため，論理演算による全加算器を作成．全加算器は１ビットに対応するため，２ビットの加算サブモジュールを作成し，add32 を改良した．すると p32m1，p32p1 について表 4 のように改良できた．

表 4: FPGA への論理合成などで得られた諸量のまとめ【改良後】

	最大動作周波数	Fmax（MHz）			
モジュール	85 ℃ Model	0 ℃ Model	LE 数（使用率）	CF 数（使用率）	レジスタ数（使用率）
プロセッサ p32m1	71.67	77.56	3838（3 %）	3804（3 %）	1379（1 %）
プロセッサ p32m2	69.57	75.35	3881（3 %）	3864（3 %）	1379（1 %）

最大動作周波数については，８５℃ Model 及び０℃ Model ともに大方 10Mhz 程度向上させることができた．しかし，両者のレジスタ数は変わらなかったにもかかわらず，LE 数および CF 数については１００～２００程度増加した．これは，算術演算子ではなく論理演算を用いたシフト命令や加算命令により，論理回路が複雑化したため，FPGA 合成の際に増加したものと考えられる．

なお，検討課題４－１を行っている際にここでは diff が出なかったものの，バグが合ったためその修正について 6.6 章に於いて述べている．

また，p32p1 の p32m1 との大きな違いは，IF ステージにて，次のステージ (ID) へ relay すると同時に次の命令を読み込む IF ステージを生成し，次の命令を処理するジョブを生成すること，WB ステージでは，ジョブを終了（finish）データフォワーディング機構（フォワードユニット）があることが挙げられる．

5 追加課題や発展課題に関する報告

発展課題については，発展課題２－１および発展課題４－１について行った．

5.1 発展課題２－１に関する報告

mult32 サブモジュールについては，算術演算子*を用いた形式で実装することが可能であったが，もう一つ組み合わせ回路により，加算器を各ビットに設定し，乗数の１が立っているビットがあれば，そ

れを被乗数のビット列と連結させる．はみ出した部分との整合性をとるために必要なビット数分右シフトを行わせて実装し，2000 パターン解析を行った．

5.2 発展課題４－１に関する報告

こちらについては，p32m1 および p32m2，p32p1 を考察する際に述べているため，4.2.1 章を参照されたい．

6 検討・考察

検討および考察に際し，検討課題１－１，１－２，１－３，検討課題２－１，検討課題４－１について報告をする．

なお，発展課題４－１についての報告の 4.2.1 章にて演算器およびプロセッサの改良を行っている．

6.1 検討課題１－１

メモリから値をレジスタにロードする命令（lw）の直後に，ロードした値を使うようなコードを書いている場合に lw の直後にロードした値を使うと，lw はまだ完了していないので add 命令移動前の値が入っていたり何もない場合があり，参照できないという不具合が発生する．

6.2 検討課題１－２

ジャンプ/分岐命令の直後（遅延スロット）に，nop 命令が書かれている．この nop 命令を取り除いた際にジャンプ命令は 1 サイクル分遅れるため，beq の判定が終わる前に lw 命令が実行されてしまい，`$t4` レジスタに目的とは異なる値が入ってしまう．そしてその後 beq による移動が起こる．

6.3 検討課題１－３

ロード命令やジャンプ/分岐命令の直後（遅延スロット）には，nop 以外の命令を置くことで，削減できる．削減できる箇所としては，lw で読み込みに使うレジスタを使わない命令（例えば lw に利用しないレジスタへの add 命令など）なら lw の後に実行しても問題がないので，lw に使うレジスタを利用していないループカウンタをへらす処理などを行うようにすることで実行命令数の削減が期待できる．

確認すると，ループのないところの nop 命令を n 個削減すると n 個の命令数が，ループに関わるところの nop 命令を削減するとループ数× n 個の命令が削減できた．

6.4 検討課題２－１

演算子&，|，~，^のこれらの演算子を用いずに全加算器と同じ機能を記述する方法としては，論理演算で行っている部分を if 文による記述に変更することが考えられる．

```
def add(a,b,ci): Unit = {
  alt {
    a == 1 && b == 1: co = 0b1
    a == 1 && ci == 1: co = 0b1
    b == 1 && ci == 1: co = 0b1
```

```
      else: co = 0b0
    }
    alt {
      a == 1 && b == 0 && ci == 0: out = 0b1
      a == 0 && b == 1 && ci == 0: out = 0b1
      a == 0 && b == 0 && ci == 1: out = 0b1
      a == 1 && b == 1 && ci == 1: out = 0b1
      else: out = 0b0
    }
  }
```

上記の様に，add 関数の中身で，alt を使い Switch のように a，b，ci，のそれぞれの入力値に応じて出力する値を決めることで論理演算と同等の処理を実現した．

6.5 検討課題４－１

メモリのアクセスに遅延がある場合，プログラム実行に要するサイクル数はどのようになるか検討した．具体的な方法は，テストベンチの FSL 記述を書き換えて行った．IMEM_LATENCY は命令メモリのアクセス遅延を，DMEM_LATENCY はデータメモリのアクセス遅延をそれぞれ指定の値だけ行う．

6.6 検討結果

まず，両者の値が０つまり，遅延がないときの各命令は５サイクルで動いていた．

命令メモリのアクセス遅延を１にして実行してみた．すると，読み込む前に `read_req` が発生し，１サイクル分遅れていることがわかった．このことから，n 個の命令があり m サイクル分の遅延があるとすると，$(5+m)*n$ サイクルが必要となることがわかる．

次に，データメモリへのアクセス遅延を１にして実行してみた．

すると，SW,SB の命令において，バグが見つかり，処理が永遠に終わらなかった．WB へ渡す引数の値を修正することで対応できた．そして，修正後のプログラムで結果を見てみると，命令メモリのときとは異なり，`S d_read` の命令が実行されているときのみ遅延が発生していた．このことから，メモリからの読み込みに x サイクルの遅延があり，lw,lb 等の命令が y 個ある n 個の命令の実行には $5*n+x*y$ サイクルが必要となる．

7 工夫した点や特に力を注いだ点

- 実験を実施するにあたって，まず Slack で実験をリモートで行うための事前準備に参加した．大まかには，Slack での情報共有および ScrapBox への情報の保存を行った．そして，具体的には，Mac および Unix 系統についてしか書かれていなかった実験サーバへの SSH 接続のドキュメントを，大半の生徒が使うであろう Windows を想定したドキュメントを，外部サービス Qiita を使い，MarkDown 記述を用いて作成した．多くの生徒が参照した有益なドキュメントとなった．

- また，実験サーバ上の CUI 環境でのプログラム制作が億劫だったため，リモートの環境をローカルで実現しようと，wsl を Windows 環境に入れ込み，SCP コマンドなどを用いて，実験サーバ上の環境と同じものを実現しようと工夫した．その後，タイミング解析などに使うソフトをローカルで実現しようとしたが断念．Visual Studio Code にて，SSH 接続先のリモート環境のファイル

も操作可能ということを見つけ，以降はコマンドラインの実行，ファイルの編集は VScode 上で行った．ファイルも定期保存されるため，不慮の事故でファイルの変更を失うことがなくスムーズに実験を行えた気がする．

- プログラム課題，及び設計課題上の工夫としては，基本穴埋め形式だったので，それを埋めるだけではなくきちんと構造を理解するよう努めた．オンラインと言う特性を活かし，Slack や講義内で質問を行い解決する努力をした．

8 本実験を実施して得られたこと

8.1 目標達成度

表 5: 目的達成度と自己評価

項目	達成度（自己評価）
１．ハードウェア設計処理全般（処理の概要と流れ）	5
２．ハードウェア記述言語（FSL）	6
３．ハードウェア設計ツール類使用方法	6
４．プロセッサの命令セットアーキテクチャ	6
５．アセンブリ言語とそれを用いたプログラミング	6
６．プロセッサの動作原理	5
７．プロセッサの設計	5
８．実験報告書	6
９．その他	6
１０．総合（項目１．～９．の合計）	５１

8.2 得られたこと

- 本実験の目的である，ハードウェア記述言語と CAD（computer aided design）ツールを利用したマイクロプロセッサ設計を通して，論理回路，コンピュータアーキテクチャ，およびコンピュータシステムに関する理解を深めることについて達成することができた．

- ２年次のコンピュータアーキテクチャおよびオペレーティングシステムの講義で学習していたシステムコールの呼び出しや各種命令の各種命令の実行サイクルについて５サイクルで命令が実行されることや，パイプライン方式でのプロセッサの実装とマルチサイクルでのプロセッサの実装方式が異なることについて，プログラムを介してその小さな世界を垣間見ることができ，方式の違いによるサイクル数の違いについて詳しく理解することができた．

- 算術演算はすべて，論理演算に落とし込めるという知見を，add32 サブモジュールの全加算器を作成することから得ることができた．シフト命令についても，用意されている演算子ではない論理的な実装を行うことでシフトの実装内容を理解できるようになった．

- 掛け算がどのようにシフト演算や足し算を用いて行っているのか，当初は理解し難いものだったが，ScrapBox などでの他の生徒からのアドバイスや先生からの解説により理解を深めることができた．

9　おわりに

今回は，初となるオンラインでの実験であり，色々と戸惑うこともあったが Slack での質問などでなんとか実験の目的を達成することができた．自身の環境の改善など，実験の本質とは異なるものの様々なツール及びコマンドについても知ることができとても有意義なものであった．

FSL 言語や慣れないプロセッサの設計で，時間が多くかかってしまったため解くことができなかった除算の実装や，プロセッサの乗算の実装や検討課題があるため，それらが今後の課題として残った．なお，p32p1 プロセッサの改良についても行いたい．

パイプライン方式やマルチサイクルといった技術は現在のプロセッサには欠かせないものとなっているということ，また，マルチコアの並列プロセッサの実装についての理解なども深めて行きたい．

10　作成した設計記述，プログラム等のリポジトリ名について

全てのモジュールは本実験で利用した gitbucket の実験用のリポジトリで管理を行った．
URL:http://jikken1.arc.cs.okayama-u.ac.jp/gitbucket/09430509

表 6: 課題に対応するディレクトリ

課題	URL 以下のディレクトリ
（プログラミング課題）	
1.【プログラミング課題１】N 個の語の加算	`\prog-kadai1`
2.【プログラミング課題２】N 語のメモリコピー	`\prog-kadai2`
3.【プログラミング課題３】乗算	`\prog-kadai3`
（設計課題２）	
4.【設計課題２－１】３２ビット加算器　add32	`\add32`
5.【設計課題２－２】３２ビット ALU　alu32	`\alu32`
6.【設計課題２－３】３２ビットシフタ shift32	`\shift32`
（発展課題２）	
7.【発展課題２－１】３２ビット整数乗算器　mult32	`\mult32`
8.【発展課題２－２】３２ビット整数除算器　div32	
（設計課題３）	
9.【設計課題３－１】レジスタファイル　regs32x32	`\regs32x32`
１０.【設計課題３－２】実行ユニット　p32ExecUnit	`\p32ExecUnit`
１１.【設計課題３－３】デコードユニット p32DecodeUnit	`\p32DecodeUnit`
（設計課題４）	
１２.【設計課題４－１】プロセッサ　p32m1	`\p32m1`
１３.【設計課題４－２】プロセッサ　p32m2	`\p32m2`
１４.【設計課題４－３】プロセッサ p32p1	`\p32p1`

参考文献

[1] ScrapBox 及び講義資料を参考にした, `https://scrapbox.io/jikken-a/`