情報工学実験 IV

第 1 回： 16 ビット RISC マシンの実装

2018 年 11 月 27 日

柴田 裕一郎（shibata@cis.nagasaki-u.ac.jp）

1 RISC とは

本実験では RISC（Reduced Instruction Set Computer）型のマイクロプロセッサの設計と実装を行います．「RISC とは何か」を正確に定義することは現在ではやや難しくなっていますが，以下のポイントが RISC の特徴と考えていいでしょう．

- 単一固定の命令長を持つ
  命令の種類によって命令の長さが 2 ワードだったり 3 ワードだったりするものは RISC とは言いません．RISC では命令長は固定されており，命令のフィールディングも単純な構成になっていて，なるべく命令のデコード（フェッチしてきた命令が何なのかを解釈すること）が簡単になるようになっています．このことから，たとえば PDP-11 や Z80 や 6502 は RISC には分類されません．
- レジスタ・レジスタアーキテクチャである
  RISC は複数の汎用（用途の決まっていない）レジスタを持ち，加算などの基本命令のオペランド（演算の対象）はいずれもレジスタです．つまり，あるレジスタの値と，あるレジスタの値を加算して，結果をあるレジスタに保存するという具合です．これをレジスタ・レジスタアーキテクチャ，あるいは Load/Store アーキテクチャ[*1] と呼びます．

RISC という名前のもともとの意味は「命令の種類を減らしたコンピュータ」です．これは，複雑で高機能な命令をいくつも用意するのではなく，単純で粒のそろった少ない種類の命令群をパイプライン方式（工場の流れ作業のような方式）で高速に実行するという設計思想を表しています．しかし，流れ作業を乱さない命令であれば，命令の種類が多くなっても害はないので，最近の RISC には並列処理用の命令などがどんどん追加されています．したがって，今となっては命令が何種類用意されているかということはあまり RISC の本質ではなくなりつつあります．一方で，命令セットを内部で変換する技術も発達しており，CISC（Complex Instruction Set Computer: つまり RISC でないマシン）の代表選手のように言われる Intel の x86 シリーズプロセッサも，内部では命令を RISC 形式に変換して処理しています．

RISC マシンの大雑把な構成を図 1 に示します．複数の汎用レジスタが集まった「レジスタファイル」と，算術演算や論理演算を行う組合わせ回路である ALU（Arithmetic Logic Unit）に，メモリを結合した構成を持ちます．RISC マシンでは，演算はレジスタとレジスタの間で行われ，結果もレジスタに格納されます．この他に，メモリの指定した番地からデータをレジスタに取り込む（ロード）機能や，レジスタの値をメモリの指定した番地に書き込む（ストア）機能を持ちます．今回は 16 ビットの RISC マシンを考えます．つまり，ALU や各レジスタのビット幅は 16 ビットということです．

さて，今回の実験ではメモリが登場します．これまでの順序回路では，記憶素子としてレジスタ（D 型フリップフロップ）だけを用いてきましたが，より大規模なデータを記憶させるにはメモリを用います．読み書

[*1] 演算をする前にメモリからオペランドを Load しておき，演算後には改めて結果を Store しなくてはならないため．

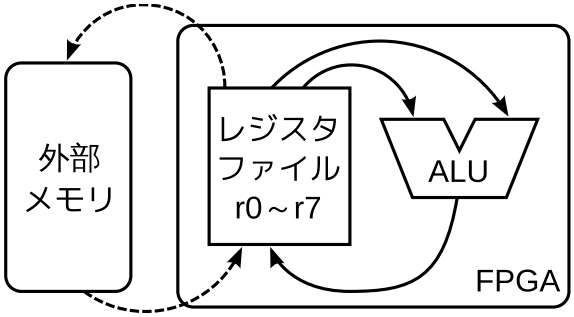

図 1　RISC マシンの概念図

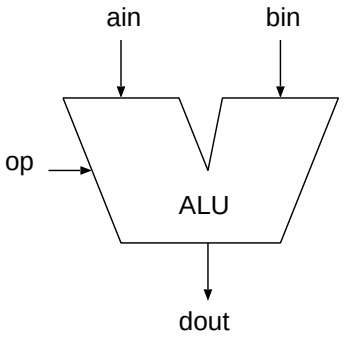

図 2　ALU のインタフェース

き可能なメモリ素子としては，以下の 2 種類がよく使われます．

- SRAM（Static RAM）
  容量は小さいが高速．インバータ（NOT 素子）でループを作って値を覚えさせる．アクセスの手順は簡単で使いやすい．プロセッサのキャッシュメモリなどに使われる．
- DRAM（Dynamic RAM）
  容量が大きいがやや低速．半導体で作ったコンデンサの電荷に値を覚えさせる．アクセスの手順がやや複雑．コンデンサはほっておくと電荷が抜けてしまうので，定期的に値を覚え直す必要があり，これをリフレッシュと呼ぶ．PC などのメインメモリ（主記憶）などに使われる．

今回は実験ボードに搭載されている SRAM を使用します．FPGA の外部にあるメモリを使うことになるので，SystemVerilog ファイルにメモリの記述を行って論理合成する必要はありません．

2　ALU の構成と記述

ALU（Arithmetic Logic Unit）は加減算や論理演算（論理和や論理積など）を行うユニットのことです．整数演算用 ALU は組合わせ回路として実現し，乗算や除算は扱わないのが一般的です．乗除算は加減算よりも複雑で処理時間も長くなるので，別途専用のユニットを用意したり，浮動小数点演算装置を使うのが普通です．

ALU のインタフェースには，図 2 に示すように，演算の対象となる 2 つの 16 ビット入力（ain，bin），演算結果を出力する 16 ビットの出力（dout），そしてどの演算を行うのか指定するための入力（op）が必要です．演算指定のための入力（op）に何ビットが必要になるかは，ALU で何種類の演算をサポートするかに依存しますが，今回は 4 ビットとし，表 1 に示す演算を実装することにします．このうち「ain をスルー」というのは，ain へ入力された値をそのまま出力させる動作のことです．「bin をスルー」も同様です．これが何の役に立つのかは後で明らかになります．また，左右両方向へ 1 ビットシフトする演算も用意します．シフトには論理シフトと演算シフトとがありますが，今回は論理シフトとします．つまり，シフトの際に空いたビットには常に ‘0’ を入れることとします．さらに，bin を左に 8 ビットシフトする演算も用意することにします．

ソースコード 1 に ALU モジュールの記述例を示します．表 1 のうち，ain のスルー，bin のスルー，左方向への 1 ビットシフトの 3 種類の演算の記述を示しています．case 文の中では，まず，op 入力が ain のスルーを指示している場合には，

表 1　ALU でサポートする演算

op の値	動作
0000	ain をスルーしてそのまま出力（A）
0001	bin をスルーしてそのまま出力（B）
0010	bin の否定（$\overline{B}$）
0011	排他的論理和（$A \oplus B$）
0100	加算（$A + B$）
0101	減算（$A - B$）
0110	bin を左へ 8 ビットシフト（$B << 8$）
0111	—
1000	bin を左へ 1 ビットシフト（$B << 1$）
1001	bin を右へ 1 ビットシフト（$B >> 1$）
1010	論理積（$A \wedge B$）
1011	論理和（$A \vee B$）

ソースコード 1　16 ビット ALU の記述例（一部）

```
1  `define ALU_THROUGH_AIN 4'b0000
2  `define ALU_THROUGH_BIN 4'b0001
3   ...
4  `define ALU_SHIFT_LEFT  4'b1000
5   ...
6
7  module alu16
8    (
9     input wire [15:0]   ain, bin,
10    input wire [3:0]    op,
11    output logic [15:0] dout
12    );
13
14    always_comb begin
15       case (op)
16         `ALU_THROUGH_AIN: dout <= ain;
17         `ALU_THROUGH_BIN: dout <= bin;
18          ...
19         `ALU_SHIFT_LEFT:  dout <= bin << 1;
20          ...
21         default:          dout <= 16'bx;
22       endcase
23    end
24 endmodule
```

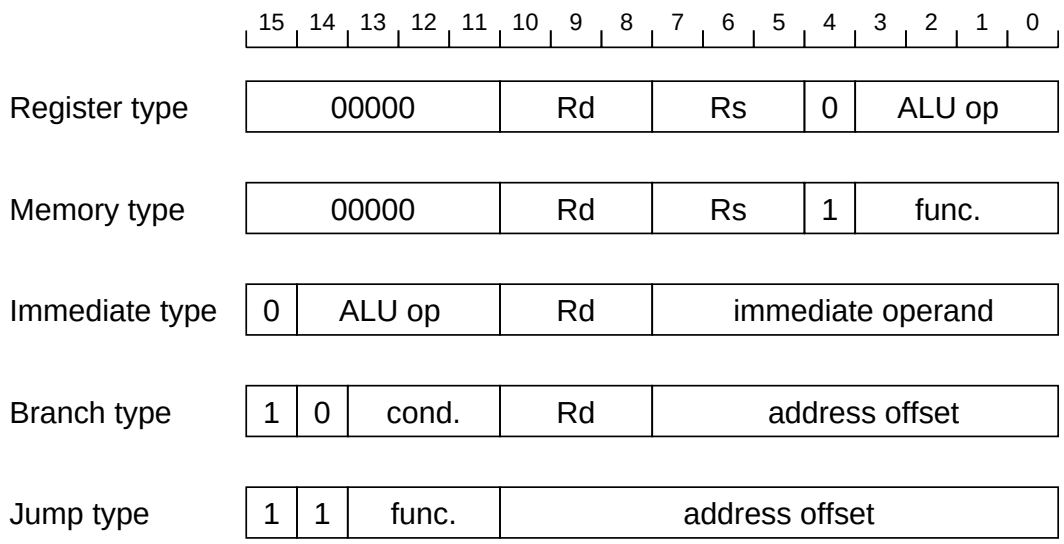

図 3　RISC16 の命令フィールディング

```
‘ALU_THROUGH_AIN: dout <= ain;
```

のように出力に ain を直接代入しています．ここで，‘ALU_THROUGH_AIN は，1 行目を見ると分かるように，ALU の演算指定の「ain をスルーしてそのまま出力」の符号（4’b0000）に定義されています．bin のスルーも同様に実現できます．

左シフトは以下のような記述で実現しています．

```
‘ALU_SHIFT_LEFT:  dout <= bin << 1;
```

記号 « は C 言語と同様のシフト演算子です．

また，case 文の最後には，いずれの条件も成立しなかったときに実行される default の記述があります．

```
default:          dout <= 16’bx;
```

16 ビットの不定値 “x” が代入されていますが，この不定値の代入は don’t care を意味します．実際には，論理合成の際に，なるべく回路が簡単になるように適当な値が選ばれて代入されることになります．

次に汎用レジスタについて考えます．今回は 16 ビットのレジスタを 8 本持ったレジスタファイルを構成し，それぞれ r0, r1, . . . , r7 と呼ぶことにします．アドレスバスのビット幅は 16 ビットとし，メモリ空間を 64K（$= 2^{16}$）バイトとします．以降，今回設計する 16 ビット RISC を RISC16 と呼ぶことにします．

3　命令セット

図 3 に RISC16 の命令フィールディングを示します．フィールディング自体は大きく分ければ 3 種類ですが，ここでは 5 種類の命令タイプに分けて説明します．なお，以下の説明で「命令の X ビット目」とは，命令コードの LSB を 0 ビット目，MSB を 15 ビット目として数えています．

（1）レジスタ命令タイプ

表 2　レジスタ命令タイプの主な命令

命令コード	ニーモニック	意味	動作
`0000000000000000`	NOP	No Operation	有効な操作は行われない（r0 を r0 に格納）
`00000dddsss00001`	MV *d*, *s*	Move	レジスタ s の内容を d に格納
`00000dddsss00010`	NOT *d*, *s*	Not	レジスタ s の反転を d に格納
`00000dddsss00011`	XOR *d*, *s*	Exclusive OR	レジスタ d と s の排他的論理和を d に格納
`00000dddsss00100`	ADD *d*, *s*	Add	レジスタ d と s の和を d に格納
`00000dddsss00101`	SUB *d*, *s*	Subtract	レジスタ d から s を引いて d に格納
`00000dddsss01000`	SL *d*, *s*	Shift Left	レジスタ s を 1 ビット左シフトして d に格納
`00000dddsss01001`	SR *d*, *s*	Shift Right	レジスタ s を 1 ビット右シフトして d に格納
`00000dddsss01010`	AND *d*, *s*	AND	レジスタ d と s の論理積を d に格納
`00000dddsss01011`	OR *d*, *s*	OR	レジスタ d と s の論理和を d に格納

表 3　メモリ命令タイプの主な命令

命令コード	ニーモニック	意味	動作
`00000dddsss10000`	ST *d*, (*s*)	Store	レジスタ d の内容を s で示す番地に格納
`00000dddsss10001`	LD *d*, (*s*)	Load	レジスタ s で示す番地の内容を d に格納

2 つのレジスタの値を演算する命令です．レジスタは 8 本なので，それぞれ 3 ビットのフィールド（Rd と Rs）で指定できます．もう少し命令長が長ければ結果を格納するレジスタも別途指定できますが，今回は命令が 16 ビットなのでオペランドの一方のレジスタに結果を上書きすることにします．命令の 3 ビット目から 0 ビット目までは ALU の op 入力に直接接続し，実行すべき演算を指定します．

（2）メモリ命令タイプ

メモリにアクセスする命令です．フィールディングはレジスタ命令タイプと同じですが，4 ビット目が ‘1’ になっています．命令の 3 ビット目から 0 ビット目まで（func.）の部分で，命令機能を区別します．今回はとりあえずロード（LD）とストア（ST）の 2 種類だけを実装します．LD や ST がアクセスする実効番地は，あらかじめ Rs のフィールドで指定されたレジスタの中に格納しておくものとします．すなわち**レジスタ間接アドレッシング**を用います．

（3）イミディエイト命令タイプ

片方のオペランドを命令中に直接与える形式の命令です．レジスタは 1 つだけ指定され，演算結果はこのレジスタに上書きされます．即値のフィールドは 8 ビットの幅を持っています．ADDI 以外の命令では上位 8 ビットに ‘0’ が埋められます（ゼロ拡張）．ADDI 命令では**符号拡張**します．ALU の op 入力には命令の 14 ビット目から 11 ビット目を接続すれば OK です．なお，LUI 命令では ALU に新しく付け加えた 8 ビット左シフト演算が用いられ，フィールド Rd で指定されたレジスタの上位 8 ビットに即値が格納されます．このときの下位 8 ビットは 0 になります．

（4）分岐（ブランチ）命令タイプ

フィールディングはイミディエイト命令タイプと同じです．10 ビット目から 8 ビット目（Rd のフィールド）で指定された番号のレジスタに格納されている値を調べ，分岐するかどうかを決定します．分岐

表 4　イミディエイト命令タイプの主な命令

命令コード	ニーモニック	意味	動作
00100dddxxxxxxxx	ADDI *d*, #X	Add Immediate	レジスタ d と値 X の和を d に格納
01010dddxxxxxxxx	ANDI *d*, #X	And Immediate	レジスタ d と値 X の論理積を d に格納
01011dddxxxxxxxx	ORI *d*, #X	Or Immediate	レジスタ d と値 X の論理和を d に格納
00001dddxxxxxxxx	LLI *d*, #X	Load Lower Immediate	値 X をレジスタ d に格納
00110dddxxxxxxxx	LUI *d*, #X	Load Upper Immediate	値 X をレジスタ d の上位 8 ビットに格納

表 5　分岐（ブランチ）命令タイプの主な命令

命令コード	ニーモニック	意味	動作
10000dddxxxxxxxx	BNEZ *d*, X	Branch on Not Equal Zero	レジスタ d が非 0 なら PC に X を足す
10001dddxxxxxxxx	BEQZ *d*, X	Branch on Equal Zero	レジスタ d が 0 なら PC に X を足す
10010dddxxxxxxxx	BMI *d*, X	Branch on Minus	レジスタ d が負なら PC に X を足す
10011dddxxxxxxxx	BPL *d*, X	Branch on Plus	レジスタ d が非負なら PC に X を足す

表 6　ジャンプ命令タイプの主な命令

命令コード	ニーモニック	意味	動作
11000xxxxxxxxxxx	JMP X	Jump	無条件に PC に X を足して相対ジャンプ

先のアドレスには**相対アドレス指定**を用い，命令の下位 8 ビットを現在の PC の値に加えることで分岐を実現します．現在より前の命令に戻ることも許すために，命令の下位 8 ビットは**符号拡張**する必要があります．

（5）ジャンプ命令タイプ

条件なしにジャンプする命令のタイプです．やはり相対アドレス指定を用います．なるべく遠まで飛べるように，アドレスオフセット（PC に加える値）に 11 ビットを割り当てています．

4　RISC16 の構成と動作

図 4 に RISC16 の構成を示します．今回の設計では，古典的な 3 バス構成をベースにしています．LD/ST 系の命令では 4 状態で，それ以外の命令では 3 状態で命令が実行されます．詳細は配布した SystemVerilog コードを参考にすれば分かると思いますが，代表的な命令について動作を簡単に説明します．

4.1　ADD 命令の動作

（1）命令フェッチ

PC（プログラムカウンタ）の内容を addr（アドレスバス）に出力し oe（出力有効）をアクティブにします．そして，メモリから din（データバス）に出力された命令コードを IR（命令レジスタ）に取り込

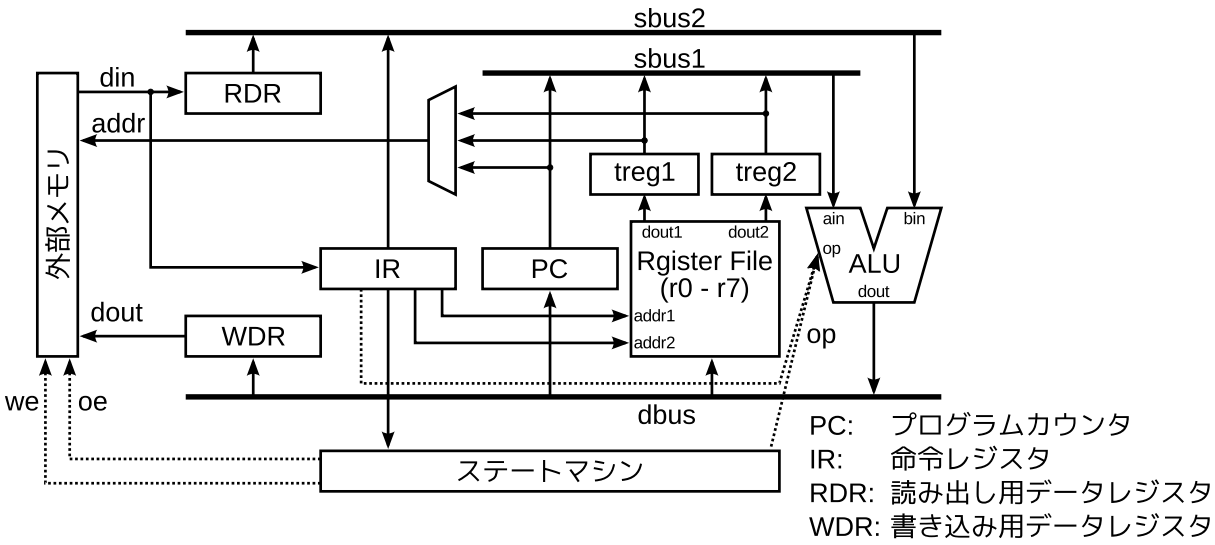

図 4　RISC16 の構成

みます．これと同時に，次の命令のフェッチのために，PC の値を sbus1 に出力し，sbus2 に ‘2’ を出力し，ALU の op を加算に設定します．そして次のクロックの立ち上がりで PC に dbus の値を取り込むことで PC に 2 を足します．

（2）レジスタフェッチ

IR（命令レジスタ）のレジスタ指定フィールド（`[10:8]` と `[7:5]`）を使ってレジスタファイルから指定された 2 つのレジスタの値を読み出します．このように，レジスタファイルは通常のメモリと違って 2 つの値を同時に読み出す機能が必要になります．このようなメモリをデュアルポートメモリ（dualport memory）と呼びます．読み出された 2 つの値は，それぞれ treg1，treg2 という一時保存用のレジスタに書き込まれます．

なお，イミディエイトタイプの命令の場合，レジスタは 1 つだけ読み出せばよいのですが，2 つとも読み出しても害はありません．わざわざ命令の種類を判定して動作を変えるとそれだけハードウェアが複雑になるので，意味がなくてもレジスタを 2 つとも読み出すように設計しています．

（3）実行

sbus1 に treg1 の値を，sbus2 に treg2 の値を出力し，IR の 3 ビット目から 0 ビット目を ALU の op に接続して加算を行います．次のクロックの立ち上がりで，ALU から dbus に出力された結果を IR の 10 ビット目から 8 ビット目で指定されたレジスタに書き込み ADD 命令の実行が終わります．状態は再び命令フェッチに遷移し次の命令についての処理を行います．なお，イミディエイトタイプの場合には sbus2 には treg2 ではなく，IR の 7 ビット目から 0 ビット目を 16 ビットに拡張して接続します．この際，符号拡張するかゼロ拡張するかは命令によって異なります．

4.2　LD 命令の動作

（1）命令フェッチ

ADD 命令の動作と同じです．

（2）レジスタフェッチ
ADD 命令の動作と同じです．
（3）実行
treg2 の値を addr に出力し oe をアクティブにします．このことによって，treg2 の中に入っていた値をアドレスとしてメモリにアクセスします（レジスタ間接アドレッシング）．メモリからの出力である din を RDR （読み出しデータレジスタ）に接続して次のクロックの立ち上がりで取り込みます．
（4）書き戻し
RDR に取り込んだ値を sbus2 に出力し，ALU をスルーさせて IR の 10 ビット目から 8 ビット目で指定されたレジスタに書き込みます．

4.3 ST 命令の動作

（1）命令フェッチ
ADD 命令の動作と同じです．
（2）レジスタフェッチ
ADD 命令の動作と同じです．
（3）実行
treg1 の値を sbus1 に出力し ALU をスルーさせます．この値を次のクロックの立ち上がりで WDR（書き込みデータレジスタ）に取り込みます．
（4）書き戻し
addr に treg2 の値を出力し，we（書き込み有効信号）をアクティブにします．もちろん dout（書き込み用データバス）には WDR の内容を出力します．

4.4 分岐命令の動作

（1）命令フェッチ
ADD 命令の動作と同じです．
（2）レジスタフェッチ
ADD 命令の動作と同じです．
（3）実行
sbus1 に PC の値を出力し sbus2 に IR の下位 8 ビットを 16 ビットに符号拡張した値を出力します．ただし，ジャンプ命令タイプは図 3 に示したように IR の下位 11 ビットを 16 ビットに符号拡張します．ALU の op は加算に設定します．treg1 の値が分岐の条件を満たすならば次のクロックの立ち上がりで PC に加算の結果を取り込みます．

5 メモリマップド I/O

さて，これらの命令を使うことによって、メモリ上に配置されたデータを入力として計算処理を行い，その結果をメモリ上に保存することができるようになりました．コンピュータの処理の基本はこの繰り返しです

ソースコード 2　メモリ初期化ファイルの例（簡単な乗算）

```
@00 00001011 00100000 // LLI r3, #0x20
@02 00000000 01110001 // LD r0, (r3)
@04 00001100 00100010 // LLI r4, #0x22
@06 00000001 10010001 // LD r1, (r4)
@08 00001101 00100100 // LLI r5, #0x24
@0a 00000010 10110001 // LD r2, (r5)
@0c 00000010 00000100 // ADD r2, r0
@0e 00100001 11111111 // ADDI r1, #−1
@10 10000001 11111010 // BNEZ r1, #−6
@12 00000010 10110000 // ST r2, (r5)
@14 00110110 00000010 // LUI r6, #0x02
@16 00000010 11010000 // ST r2, (r6)
@18 11000111 11111110 // JMP #−2
@1a 00000000 00000000 //
@1c 00000000 00000000 //
@1e 00000000 00000000 //
@20 00010010 00110100 // m
@22 00000000 00000100 // n
@24 00000000 00000000 // m ∗ n
@26 00000000 00000000 //
```

が，プロセッサとメモリだけではなんとなく地味です．人間とのインタフェースを考えると，もう少し入出力（I/O）を充実させたいところです．たとえば，計算の結果を 7 セグメント LED に数字で表示するにはどうすればよいのでしょうか．

1 つ目の考え方は，入出力のための専用命令を追加することです．例えば「ACC の内容を LED に表示する命令」のようなものを付け加えればよさそうです．しかし，プロセッサが複雑になり少しややこしそうです．できれば命令を増やすことなく LED に表示できるようにしたいですが，そんなことは可能でしょうか．

これを可能にするのがメモリマップド I/O の考え方です．各 I/O（入出力）デバイスにそれぞれ固有のアドレスを割り当てて，通常の LD 命令や ST 命令で入力や出力を行うのです．

今回のシステムでは 7 セグメント LED に $200_{(16)}$ 番地が割り当てられています．つまり $200_{(16)}$ 番地へ何かデータをストアすると，そのデータが LED に表示されるのです．この仕組みを使うことで，計算の結果を 7 セグメント LED に表示させて確認することができるようになります．ただし LED は出力専門のデバイスなので，$200_{(16)}$ 番地からデータをロードした場合の動作は不定になります．

6　プログラミング例

6.1　シミュレーションスクリプト

ソースコード 2 に簡単な乗算のプログラム例をします．r0 に変数 m の値，r1 に変数 n の値，r2 に積の値が割り当てられており，r3，r4，r5 にはそれぞれの変数の番地を格納しています．たとえば，$20_{(16)}$ 番地の値を r0 に格納するには，まず LLI 命令で r3 に $20_{(16)}$ をセットしそれから LD 命令で読み出しを行う，というように 2 つの命令が必要になります．最後に乗算の結果を LED に表示するため，LED が割り当てられている $200_{(16)}$ 番地にもストアを行っています．このとき，r6 に $200_{(16)}$ を格納していますが，$200_{(16)}$ は 2 進数で 8 ビットを超えるため LLI 命令ではセットできません．そこで LUI 命令で $2_{(16)}$ をセットしています．そうする

```
 ....
# ==== clock: 0 state: IF ====
#  pc:0000 ir:0000000000000000 treg1:0000 treg2:0000 rdr:0000 wdr:0000 oe:1 we:0
#  sbus1:0000 sbus2:0002 dbus:0002 addr:0000 din:0b20 dout:xxxx led:000000
#  regs: 0000 0000 0000 0000 0000 0000 0000 0000
#  mem[00-07]: 0b 20 00 71 0c 22 01 91
#  mem[08-0f]: 0d 24 02 b1 02 04 21 ff
#  mem[10-17]: 81 fa 02 b0 36 02 02 d0
#  mem[18-1f]: c7 fe 00 00 00 00 00 00
#  mem[20-27]: 12 34 00 04 00 00 00 00
#
# ==== clock: 1 state: RF ====
#  pc:0002 ir:0000101100100000 treg1:0000 treg2:0000 rdr:0000 wdr:0000 oe:0 we:0
#  sbus1:xxxx sbus2:xxxx dbus:xxxx addr:xxxx din:xxxx dout:xxxx led:000000
#  regs: 0000 0000 0000 0000 0000 0000 0000 0000
#  mem[00-07]: 0b 20 00 71 0c 22 01 91
#  mem[08-0f]: 0d 24 02 b1 02 04 21 ff
#  mem[10-17]: 81 fa 02 b0 36 02 02 d0
#  mem[18-1f]: c7 fe 00 00 00 00 00 00
#  mem[20-27]: 12 34 00 04 00 00 00 00
 ....
# ==== clock: 67 state: WB ====
#  pc:0018 ir:0000001011010000 treg1:48d0 treg2:0200 rdr:0000 wdr:48d0 oe:0 we:1
#  sbus1:xxxx sbus2:0000 dbus:0000 addr:0200 din:xxxx dout:48d0 led:000000
#  regs: 1234 0000 48d0 0020 0022 0024 0200 0000
#  mem[00-07]: 0b 20 00 71 0c 22 01 91
#  mem[08-0f]: 0d 24 02 b1 02 04 21 ff
#  mem[10-17]: 81 fa 02 b0 36 02 02 d0
#  mem[18-1f]: c7 fe 00 00 00 00 00 00
#  mem[20-27]: 12 34 00 04 48 d0 00 00
#
# ==== clock: 68 state: IF ====
#  pc:0018 ir:0000001011010000 treg1:48d0 treg2:0200 rdr:0000 wdr:48d0 oe:1 we:0
#  sbus1:0018 sbus2:0002 dbus:001a addr:0018 din:c7fe dout:xxxx led:0048d0
#  regs: 1234 0000 48d0 0020 0022 0024 0200 0000
#  mem[00-07]: 0b 20 00 71 0c 22 01 91
#  mem[08-0f]: 0d 24 02 b1 02 04 21 ff
#  mem[10-17]: 81 fa 02 b0 36 02 02 d0
#  mem[18-1f]: c7 fe 00 00 00 00 00 00
#  mem[20-27]: 12 34 00 04 48 d0 00 00
 ....
```

図 5　シミュレーションの実行例

と，左に 8 ビットシフトされた $200_{(16)}$ が r6 に格納されるわけです．

$18_{(16)}$ 番地の JMP 命令ではダイナミックストップのための無限ループを実現しています．命令フェッチのときに PC に 2 が加算されるので，オフセットを –2 とすると自分自身の番地にジャンプすることになります．

6.2　実行例

図 5 にソースコード 2 のプログラムを実行させたシミュレーションログを示します．乗算結果（$48d0_{(16)}$）が $24_{(16)}$ 番地に格納されていることが分かります．また，68 クロックサイクル目で LED にも表示されている

ことが分かります.

7 RISC16 の SystemVerilog 記述

```
1  `default_nettype none
2  typedef enum {IF, RF, EX, WB} state_t;
3  `define ALU_THROUGH_AIN 4'b0000
4   ...
5
6  module risc16
7    (
8     input wire          clk,
9     input wire          rst,
10    input wire [15:0]   din,
11    output logic [15:0] dout,
12    output logic [15:0] addr,
13    output logic        oe,
14    output logic        we
15    );
16
17    state_t state;
18    reg [15:0]          pc, ir, rdr, wdr, treg1, treg2;
19    logic [15:0]        sbus1, sbus2;                    // source buses
20    wire [15:0]         dbus;                            // destination bus
21    logic [3:0]         op;                              // ALU operation
22    wire [15:0]         reg_file_dout1, reg_file_dout2; // register file outputs
23    logic               pc_we, ir_we, rdr_we, wdr_we, treg_we, reg_file_we;
24
25    alu16 alu16_inst
26      (
27       .ain(sbus1),
28       .bin(sbus2),
29       .op(op),
30       .dout(dbus)
31       );
32
33    reg_file reg_file_inst
34      (
35       .clk(clk),
36       .rst(rst),
37       .addr1(ir[10:8]),
38       .addr2(ir[7:5]),
39       .din(dbus),
40       .dout1(reg_file_dout1),
41       .dout2(reg_file_dout2),
42       .we(reg_file_we)
43       );
44
45    // state control
46    always_ff @(posedge clk) begin
47       if (rst)
48         state <= IF;
49       else begin
50          case (state)
51            IF: state <= RF;
52            RF: state <= EX;
```

```
53            EX: if ( ... ) // load or store instructions
54                  state <= WB;
55                else
56                  state <= IF;
57            WB: state <= IF;
58          endcase
59      end
60    end
61
62    // program counter
63    always_ff @(posedge clk) begin
64      if (rst)
65        pc <= 16'h0;
66      else if (pc_we)
67        pc <= dbus;
68    end
69
70    // instruction register
71    always_ff @(posedge clk) begin
72      if (rst)
73        ir <= 16'd0;
74      else if (ir_we)
75        ir <= din;
76    end
77
78    // read data register
79    always_ff @(posedge clk) begin
80      if (rst)
81        rdr <= 16'd0;
82      else if (rdr_we)
83        rdr <= din;
84    end
85
86    // write data register
87    always_ff @(posedge clk) begin
88      if (rst)
89        wdr <= 16'd0;
90      else if (wdr_we)
91        wdr <= dbus;
92    end
93
94    // temporal output registers from register file
95    always_ff @(posedge clk) begin
96      if (rst) begin
97          treg1 <= 16'h0;
98          treg2 <= 16'h0;
99      end
100     else if (treg_we) begin
101         treg1 <= reg_file_dout1;
102         treg2 <= reg_file_dout2;
103     end
104   end
105
106   always_comb begin
107     case (state)
108       IF: begin /* instruction fetch */
109           addr <= pc;
```

```
110             sbus1 <= pc;
111             sbus2 <= 16'h0002;
112             op <= `ALU_ADD;
113             dout <= 16'dx;
114             pc_we <= 1'b1;
115             ir_we <= 1'b1;
116             rdr_we <= 1'b0;
117             wdr_we <= 1'b0;
118             treg_we <= 1'b0;
119             reg_file_we <= 1'b0;
120             oe <= 1'b1;
121             we <= 1'b0;
122           end
123
124           ...
125
126       endcase
127     end
128 endmodule
129
130 module reg_file
131   (
132     input wire          clk, rst,
133     input wire [2:0]    addr1, addr2,
134     input wire [15:0]   din,
135     output logic [15:0] dout1, dout2,
136     input wire          we
137     );
138
139     reg [15:0]          register0, register1;
140     reg [15:0]          register2, register3;
141     reg [15:0]          register4, register5;
142     reg [15:0]          register6, register7;
143
144     always_comb begin
145       case (addr1)
146         3'h0: dout1 <= register0;
147         3'h1: dout1 <= register1;
148         3'h2: dout1 <= register2;
149         3'h3: dout1 <= register3;
150         3'h4: dout1 <= register4;
151         3'h5: dout1 <= register5;
152         3'h6: dout1 <= register6;
153         3'h7: dout1 <= register7;
154       endcase
155     end
156
157     always_comb begin
158       case (addr2)
159         3'h0: dout2 <= register0;
160         3'h1: dout2 <= register1;
161         3'h2: dout2 <= register2;
162         3'h3: dout2 <= register3;
163         3'h4: dout2 <= register4;
164         3'h5: dout2 <= register5;
165         3'h6: dout2 <= register6;
166         3'h7: dout2 <= register7;
```

```
167         endcase
168     end
169
170     always_ff @(posedge clk)
171       if (rst) begin
172          register0 <= 16'h0;
173          register1 <= 16'h0;
174          register2 <= 16'h0;
175          register3 <= 16'h0;
176          register4 <= 16'h0;
177          register5 <= 16'h0;
178          register6 <= 16'h0;
179          register7 <= 16'h0;
180       end
181       else if (we) begin
182          case (addr1)
183            3'h0: register0 <= din;
184            3'h1: register1 <= din;
185            3'h2: register2 <= din;
186            3'h3: register3 <= din;
187            3'h4: register4 <= din;
188            3'h5: register5 <= din;
189            3'h6: register6 <= din;
190            3'h7: register7 <= din;
191          endcase
192       end
193 endmodule
194
195
196 module alu16
197  ...
198 endmodule
199 `default_nettype wire
```