

## 情報実験第三 1.B

情報工学科 15\_03602 柿沼 建太郎

情報工学科 15\_10588 中田 光

2017 年 5 月 25 日

### 各課題担当者

各課題と担当者を表として以下に示す。

課題番号/名前	柿沼	中田
解析 1		○
解析 2	○	
解析 3	○	
解析 4	○	
解析 5	○	
シミュレーション 1	○	○
シミュレーション 2		○

### Verilog 記述解析レポート (1,2,3,4,5)

#### 課題 2. 命令フェッチサイクルの動作が Verilog コード上でどのように実現されているか

Verilog では条件実行論理式を 'wire' あるいは 'assign' 文の右辺に記述することで、条件が変化するとワイヤのつながれた先の回路に無待機で情報が伝播する。

```
|| wire ワイヤー名 = 条件式;  
|| assign ワイヤー名 = 条件式;
```

'reg\_lci' または 'reg\_lci\_nxt' で宣言された module は LCI レジスタである。

そこに 0 を代入する際にはそのクリア信号を 1 にし、インクリメントする際にはインクリメント信号を 1 にする。

#### ソースコード 1 AR の場合

```
|| assign ar_clr = r & t[0]; //r & t[0] が true のときかつその時に限り AR をクリア  
|| assign ar_inr = d[5] & t[4]; //この条件のときのみインクリメントする
```

ロードするにはロード信号を 1 にすればよいが、ロード先がレジスタによって異なる。  
AR, PC, DR, IR, OTR は入力 bus\_data に繋がれており、LCI のロード信号をオンにした上で bus\_ctl の値を操作することで間接的にロードする内容を操作する。

ソースコード 2  $\overline{R} \cdot T(0) \Rightarrow AR \leftarrow PC$  の場合

```
|| assign ar_ld = ~r & t[0] |
||             ~r & t[2] |
||             ~d[7] & i15 & t[3];
|| wire bus_pc = ~r & t[0] & ~com_rst |
||             r & t[1] |
||             d[5] & t[4];
```

AC については、ALU 内で演算結果を格納するためのレジスタなので、ac\_ld が 1 になったときは ALU の計算結果をロードするようになる。

なお、ac\_ld は

```
|| assign ac_ld = ac_and | ac_add | ac_dr | ac_inpr | ac_cmp | ac_shr | ac_shl;
```

となっているため、ALU から結果をロードする以外にロードの機会はなく、事実上 AC ロード機能はないことがわかる。

SC については、入力先とロード信号の部分に 3'b0 と 1'b0 が与えられているので、ロード機能は使われない。

LCI ではない D-Flip-Flop で実現されたレジスタは 'reg\_dff' という module で宣言される。INPR, I, E, R, S, IEN, FGI, FGO, IOT, IMSK がそれにあたる。

これらに対する代入は、各レジスタに対して xx\_nxt という名前のワイヤーに条件式を充てておくことで実現される。I についてのみ ir[15] が割り当てられているが、これも ir の中身を見れば条件式と実質同様であることがわかる。

命令フェッチサイクルは条件と代入文の連続から成り、代入文は以上の機能を以て実現される。

### 課題 3. ADD, LDA, CIR, CIL において、alu モジュールがどのような動作をするか

#### ADD について

関連する Verilog コードを抜粋する。

```
|| wire [16:0] o_add = (ac_add) ? ({1'b0, dr} + {1'b0, ac}) : 17'b0;
|| assign ac_nxt = ... | o_add[15:0] | ...;
|| assign e_nxt = (ac_dd) | o_add[16] : ...;
```

alu モジュールに繋がれた入力信号 ac\_add が立っていると、dr と ac を 17 ビットとして加算されたものが o\_add に代入される。

ac\_xx 信号が同時に 2 つ以上立たないと仮定すれば、ac\_nxt には加算結果の下位 16bit がそのまま代入される。

また、同様に e\_nxt に o\_add の最上位ビットが代入される。

これによって、 $[E, AC] = [0, AC] + [0, DR]$  という計算が実現される。

## LDA について

関連する Verilog コードを抜粋する。

```
|| wire [16:0] o_dr = (ac_dr) ? (dr) : 16'b0;  
|| assign ac_nxt = ... | o_dr | ...;  
|| assign e_nxt = ... : e;
```

alu モジュールに繋がれた入力信号 ac\_dr が立っていると、dr の中身がそのまま o\_dr へ代入される。  
ac\_nxt に同様に o\_dr がそのまま代入され、e\_nxt についてはどの条件にも触れないため、維持される。  
これによって、AC = DR という計算が実現される。

## CIR について

関連する Verilog コードを抜粋する。

```
|| wire [16:0] o_shr = (ac_shr) ? ({e, ac[15:1]}) : 16'b0;  
|| assign ac_nxt = ... | o_shr | ...;  
|| assign e_nxt = ... : (ac_shr) ? ac[0] : ...;
```

alu モジュールに繋がれた入力信号 ac\_shr が立っていると、[E, AC[15 : 1] が o\_shr へ代入される。  
ac\_nxt に同様に o\_shr が代入され、e\_nxt には ac の最下位ビットが代入される。  
これによって、[AC, E] = [E, AC] という計算が実現される。

## CIL について

関連する Verilog コードを抜粋する。

```
|| wire [16:0] o_shl = (ac_shl) ? ({ac[14:0], e}) : 16'b0;  
|| assign ac_nxt = ... | o_shl;  
|| assign e_nxt = ... : (ac_shl) ? ac[15] : ...;
```

alu モジュールに繋がれた入力信号 ac\_shl が立っていると、[AC[14 : 0], E] が o\_shl へ代入される。  
ac\_nxt に同様に o\_shl が代入され、e\_nxt には ac の最上位ビットが代入される。  
これによって、[E, AC] = [AC, E] という計算が実現される。

## 課題 4. FGI レジスタについて , fgi\_set, pt, ir, iot, fgi それぞれの信号の組合せで出力値が決定する仕組み

入出力の Verilog コードについて、関係のある箇所のみ以下に抜粋する。

```
|| reg_dff #2 FGI (clk, ~com_stop, fgi_nxt & {2{~com_rst}} , fgi); /// reset value  
|| = 00;  
|| assign fgi_nxt[0] = (fgi_set[0]) ? 1 : /// fgi_set[0] : fgi[0] <- 1  
|| (pt & ir[11] & ~iot) ? 0 : /// INP : fgi[0] <- 0  
|| fgi[0]; /// unchanged  
|| assign fgi_nxt[1] = (fgi_set[1]) ? 1 : /// fgi_set[1] : fgi[1] <- 1  
|| (pt & ir[11] & iot) ? 0 : /// INP : fgi[1] <- 0  
|| fgi[1]; /// unchanged  
|| edge_to_pulse #(4,0) FGP (clk, {fgi_bsy, fgo_bsy}, {fgi_set, fgo_set});  
|| wire pt = d[7] & i15 & t[3]; /// @ t[3] : implies IO register-insn type
```

```

|| assign iot_nxt = (pt & ir[5]) ? 1          : /// SIO : iot  <- 1
||                (pt & ir[4]) ? 0          : /// PIO : iot  <- 0
||                iot;                      /// unchanged

```

FGI は 2 ビットの D-Flip-Flop 型のレジスタで、0 番はパラレル通信、1 番はシリアル通信となっている。pt は間接アドレスフェッチサイクルのときに立つフラグで、ir[11] はワンホットコードのうち INP 命令かどうかを示す bit である。

IOT は現在使用中なのがシリアルかパラレルかを示すフラグであり、SIO 命令が来ると 1 に、PIO 命令が来ると 0 になることから、0 のときはパラレル通信で 1 のときはシリアル通信を表現する。

fgi\_set はであるようなフラグである。

このプログラムでは edge\_to\_pulse をネガティブエッジパルス生成器としてインスタンス化しており、出力信号に fgi\_set が充ててあるため fgi\_set は fgi\_bsy の値が 1 から 0 になったクロックでのみ 1 を示す。

したがって、各 FGI は、busy 状態が解除されたら 1、INP 命令が来ておりかつ間接アドレスフェッチサイクルまで来ていてかつ IOT によって自身が選択されていたら 0 に、そうでなければ維持される。

## 課題 5. AR レジスタの出力信号が ar と ar\_nxt の 2 つある理由

ex3 の CPU の中でレジスタは大きく分ければ LCI レジスタと DFF レジスタが使われており、それらはそれぞれ 'reg\_lci' と 'reg\_dff' という名前で定義されている。

しかし AR レジスタに限り、'reg\_lci\_nxt' という名前の module を使っている。

'reg\_lci\_nxt' は 'reg\_lci' に加え、「まだ代入されていない値」を出力するという機能がついている。

一般的にレジスタへの代入は、レジスタへの入力信号に値を入れたまま待機し、クロックの立ち上がりきた瞬間に出力にそれが反映される。

しかし、'reg\_lci\_nxt' ではまだ出力信号に来ていないような待機状態の値も出力するという機能がある。この必要性について考える。

まず、メモリのロード先アドレスを示すワイヤー 'mem\_addr' について次のような記述がある。

```

|| wire [11:0] mem_addr = (com_stop) ? com_addr : (mem_we) ? ar : ar_nxt;

```

'mem\_we' はメモリに対して書き込みを行うときに立つフラグであるため、メモリは書き込み時には ar を、読み込み時には ar\_nxt を使うことがわかる。

このことから、メモリを読むときだけはクロックの立ち上がりを待ってはもらえない理由があると考えた。

ここで、間接アドレスフェッチサイクルからメモリ参照命令実行サイクルへ移行するタイミングについて考える。

まず、間接アドレスフェッチの立ち上がりでメモリからの読み出しが行われる (メモリは同期式なので立ち上がりの瞬間にしか出力の値は変更されない)。

次の立ち上がりで AR への書き込みが起きる。

すると、メモリ参照命令実行サイクルの始まりでは AR への書き込みと読み出しが同時に行われることになる。

もしメモリからの読み出しに `ar` を使っていたとすると、書き込み中の値を使用することになるため、値移行中の不定値を使用することになりメモリ読み出しが破綻する。

しかし `ar_nxt` を使うことにより、まだ `AR` の本来の出力にきていない値を使用することができるため安全に読み出しができる。

ではすべての時に `ar_nxt` を使えばよいかというとそれでは問題が起きる場合がある。

BSA 命令の実行サイクルを見てみると、 $AR \leftarrow AR + 1, M[AR] \leftarrow PC$  となっている。`AR` への書き込みが起きている途中は `ar_nxt` の値が今度是不定となる。

したがって、書き込みのときには不定ではない `ar` を使う必要がある。

## Verilog シミュレーションレポート (1,2)

### 1. 4 つの課題プログラムそれぞれについて 4 つ以上の異なる入力に対する実行サイクル数とそれに関する考察

#### 倍制度乗算

柿沼: 実行サイクル数

入力 ( $X \times Y$ )	ステップ数 (前回)	ステップ数 (今回)	実行サイクル数 (今回)
$11 \times 13$	90	93	445
$0 \times 30$	114	117	564
$30 \times 0$	4	7	21
$65535 \times 65535$	403	406	1985

#### 剰余算

柿沼: 実行サイクル数

入力 ( $X \div Y$ )	ステップ数 (前回)	ステップ数 (今回)	実行サイクル数 (今回)
$30 \div 7$	340	343	1654
$0 \div 15$	336	339	1634
$65535 \div 1$	400	403	1954
$65535 \div 65535$	340	343	1654

16 進 → 10 進

柿沼: 実行サイクル数

入力 ( $X \times Y$ )	ステップ数 (前回)	ステップ数 (今回)	実行サイクル数 (今回)
$0F$	783	786	3832
$FFFF$	1957	1960	9583
$XX$	67	70	370
$ABCX$	157	160	820

素数計算

柿沼: 実行サイクル数

入力 ( $X \times Y$ )	ステップ数 (前回)	ステップ数 (今回)	実行サイクル数 (今回)
2	27	30	129
64	61307	61310	297992
255	337101	337103 <sup>*1</sup>	1638582 <sup>*1</sup>
65535	245377052	245377054 <sup>*1</sup>	1192716791 <sup>*1</sup>

考察

柿沼: まずステップ数は前回の計測とくらべてすべて 3 ずつ増えている。最後に HLT が来てから余計に 3 回 HLT 命令のところで止まっているのが確認されたため、そこが増えた回数である。脚注 [1] でも触れているが、デバッグ用のスイッチの切り替えにより増える数値が 1 3 まで変化することがわかっているので、おそらく Verilog シミュレーターのデバッグの仕様であると考えられるが、それ以上のことはわからなかった。ステップ数と実行命令サイクル数については

---

<sup>\*1</sup> CPU\_MONITORING を消したため、他のものと同様 1 少ない