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 29 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 1. EX3 がリセットされる時、PC の値が 0x010 に設定される仕組み

まず、com_rst は com_ctr が 00 の時に 1 となるワイヤー変数として宣言されている。
('COM_RST は def_ex3.v で 00 と定義されている)

ソースコード 1　cpu_ex3.v 33 行目

```
wire com_rst = (com_ctl == `COM_RST); /// reset (pc, sc, ar, 1-bit FFs)
```

つまり、com_rst=1 となる時について考えればよい。
次に、reg_lci として定義された SC では、出力する値をクリアするフラグとして sc_clr—com_rst が指定されている。
com_rst=1 の時は SC の出力 sc は 0 となる。

ソースコード 2　cpu_ex3.v 73 行目

```
reg_lci #3 SC  (clk, ~com_stop, 3'b0, sc, 1'b0, sc_clr | com_rst, ~sc_clr); /// if(
```

```
    sc_clr == 0) sc ++;
```

sc は 3to8 デコーダ DET_T によって 8 ビットワイヤ変数 t に変換される。
sc=0 の時、t[k] = 0 (k=0,1,2...7) となる。

ソースコード 3　cpu_ex3.v 94 行目

```
dec_3to8 DEC_T (sc, t, 1'b1); /// (t[k] == 1) implies sc = k;
```

t[k] = 0 (k=0,1,2...7) となる時、3 ビットワイヤ変数 bus_ctl は 000 となる。

ソースコード 4　cpu_ex3.v 217〜235 行目

```
wire bus_ar = d[4] & t[4] |   /// BUN @ t[4] : pc <- ar;
                      d[5] & t[5]; /// BSA @ t[5] : pc <- ar;
wire bus_pc = ~r & t[0] & ~com_rst | /// fetch @ t[0] : ar <- pc; (com_rst = 0)
                      r & t[1] | /// interrupt @ t[1] : mem[ar] <- pc;
                      d[5] & t[4]; /// BSA @ t[4] : mem[ar] <- pc;
wire bus_dr = d[6] & t[6]; /// ISZ @ t[6] : mem[ar] <- dr;
wire bus_ac = d[3] & t[4] | /// STA @ t[4] : mem[ar] <- ac;
                      pt & ir[10]; /// OUT : outr <- ac[7:0]
wire bus_ir  = ~r & t[2]; /// fetch @ t[2] : ar <- ir[11:0];
wire bus_mem = ~r & t[1] | /// fetch @ t[1] : ir <- mem[ar];
                      ~d[7] & i15 & t[3] | /// indirect : ar <- mem[ar];
                      d[0] & t[4] | /// AND @ t[4] : dr <- mem[ar];
                      d[1] & t[4] | /// ADD @ t[4] : dr <- mem[ar];
                      d[2] & t[4] | /// LDA @ t[4] : dr <- mem[ar];
                      d[6] & t[4]; /// ISZ @ t[4] : dr <- mem[ar];

assign bus_ctl[0]  = bus_ar | bus_dr | bus_ir | bus_mem; /// b1 | b3 | b5 | b7
assign bus_ctl[1]  = bus_pc | bus_dr | bus_mem; /// b2 | b3 | b7
assign bus_ctl[2]  = bus_ac | bus_ir | bus_mem; /// b4 | b5 | b7
```

bus_ctl が 000 の時 BUS では b0 が選択され、
bus_data には'PROGRAM_ENTRY_POINT が出力される。
('PROGRAM_ENTRY_POINT は def_ex3.v で 0x010 と定義されている。)

ソースコード 5　cpu_ex3.v 89 行目

```
bus BUS (bus_ctl, `PROGRAM_ENTRY_POINT, {4'b0, ar}, {4'b0, pc}, dr, ac, ir, 16'b0,
    mem_data, bus_data);
```

最後に、lci レジスタ PC において、
com_rst = 1 の時に bus_data がロードされ、出力 pc が bus_data の値 (0x010) になる。

ソースコード 6　cpu_ex3.v 89 行目

```
reg_lci #12 PC (clk, ~com_stop, bus_data[11:0], pc, pc_ld | com_rst, pc_clr, pc_inr);
```

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ー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 7　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
```

ロードする際にはロード信号を 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, OUTR は入力が 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 8　$\overline{R} \cdot T(0) \Rightarrow AR \leftarrow PC$ 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とわかる。

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

柿沼:

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、同様にして 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。

中田:

ADD について

まず,ADD の実行サイクル 6 で ac_add が 1 となる。

ソースコード 9　cpu_ex3.v 166 行目

```
assign ac_add  = d[1] & t[5]; /// ADD @ t[5] : ac <- ac + dr;
```

次に、alu において、ac_add=1 であるため {0,dr} と {0,ac} が加算されたものが o_add に代入される。そして、ac_nxt には o_add[15:0]、e_nxt には o_add[16] が出力される。(変更のあったレジスタ以外のレジスタ o_* は 0 であるため、変更のあったレジスタの値が ac_nxt に代入される。e_nxt は条件文に沿った値が代入される。他の命令についても同様である。)
これによって AC+DR の加算が実現する。

ソースコード 10　cpu_module.v ALU model より

```
wire   [16:0] o_add  = (ac_add)  ? ({1'b0, dr} + {1'b0, ac}) : 17'b0;
assign ac_nxt = o_and | o_add[15:0] | o_dr | o_inpr | o_cmp | o_shr | o_shl;
assign e_nxt =  (ac_add) ? o_add[16] :
                    (ac_shr) ? ac[0] :
                    (ac_shl) ? ac[15] :
                    (e_clr)  ? 1'b0 :
                    (e_cmp)  ? ~e : e;
```

LDA について

まず,LDA の実行サイクル 6 で ac_dr が 1 となる。

ソースコード 11　cpu_ex3.v 167 行目

```
assign ac_dr   = d[2] & t[5];    /// LDA @ t[5] : ac <- dr;
```

次に、alu において、ac_dr=1 であるため、dr が o_dr に代入される。そして、ac_nxt には o_dr、e_nxt には e が出力される。
これにより AC ← DR が実現する。

ソースコード 12　cpu_module.v ALU model より

```
wire   [15:0] o_dr   = (ac_dr)   ? (dr) : 16'b0;
```

```
assign ac_nxt = o_and | o_add[15:0] | o_dr | o_inpr | o_cmp | o_shr | o_shl;
assign e_nxt =  (ac_add) ? o_add[16] :
                    (ac_shr) ? ac[0] :
                    (ac_shl) ? ac[15] :
                    (e_clr)  ? 1'b0 :
                    (e_cmp)  ? ~e : e;
```

CIR について

まず,CIR の実行サイクル 4 で ac_shr が 1 となる。

ソースコード 13　cpu_ex3.v 170 行目

```
assign ac_shr  = rt & ir[7]; /// CIR : ac[14:0] <- ac[15:1], ac[15] <- e, e_nxt <- ac
    [0];
```

次に、alu において、ac_shr=1 であるため、o_shr に {e,ac[15:1]} が代入される。そして、ac_nxt には o_shr、e_nxt には ac[0] が出力される。
これにより AC の回転右シフトが実現する。

ソースコード 14　cpu_module.v ALU model より

```
wire   [15:0] o_shr  = (ac_shr)  ? ({e, ac[15:1]}) : 16'b0;
assign ac_nxt = o_and | o_add[15:0] | o_dr | o_inpr | o_cmp | o_shr | o_shl;
assign e_nxt =  (ac_add) ? o_add[16] :
                    (ac_shr) ? ac[0] :
                    (ac_shl) ? ac[15] :
                    (e_clr)  ? 1'b0 :
                    (e_cmp)  ? ~e : e;
```

CIL について

まず,CIL の実行サイクル 4 で ac_shl が 1 となる。

ソースコード 15　cpu_ex3.v 171 行目

```
assign ac_shl  = rt & ir[6]; /// CIL : ac[15:1] <- ac[14:0], ac[0]  <- e, e_nxt <- ac
    [15];
```

次に、alu において、ac_shl=1 であるため、o_shl に {ac[14:0],e} が代入される。そして、ac_nxt には o_shl、e_nxt には ac[15] が出力される。
これにより AC の回転左シフトが実現する。

ソースコード 16　cpu_module.v ALU model より

```
wire   [15:0] o_shl  = (ac_shl)  ? ({ac[14:0], e}) : 16'b0;
assign ac_nxt = o_and | o_add[15:0] | o_dr | o_inpr | o_cmp | o_shr | o_shl;
assign e_nxt =  (ac_add) ? o_add[16] :
```

```
                    (ac_shr) ? ac[0] :
                    (ac_shl) ? ac[15] :
                    (e_clr)  ? 1'b0 :
                    (e_cmp)  ? ~e : e;
```

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、関係のある個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 番はシリアル通信用となっている。
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が来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いられ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 の本来の出力に来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 の値が今度は不定となる。
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)
11×13	90	93	445
0×30	114	117	564
30×0	4	7	21
65535×65535	403	406	1985

剰余算

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)
$30 \div 7$	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[*1]	1638582[*1]
65535	245377052	245377054[*1]	1192716791[*1]

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。

1 ステップあたりの実行命令サイクル数を平均すると、約 4.75 回となった。これは 1 つの命令あたりにかかるサイクルがおよそ 4,5 回という事実を意味し、実際、EX3 の実行命令サイクルの表を見ると、多くの命令は 5 回のサイクルで終わる。平均回数が 5 回より少なくなっているのは出力などに要する割り込みサイクルであると考えられる。

[*1] CPU_MONITORING を消したため、他のものとくらべ 1 少ない

倍制度乗算

中田: 実行サイクル数

入力	ステップ数 (前回)	ステップ数 (今回)	実行サイクル数 (今回)
11×13	94	97	461
0×30	119	122	584
30×0	4	7	21
65535×65535	419	422	2049

剰余算

中田: 実行サイクル数

入力	ステップ数 (前回)	ステップ数 (今回)	実行サイクル数 (今回)
$30 \div 7$	362	365	1715
$0 \div 15$	355	358	1681
$65535 \div 1$	467	470	2225
$65535 \div 65535$	362	365	1715

※ $0 \div 15$ の前回のステップ数に誤りがありましたので訂正しました。

16 進 $\rightarrow$ 10 進

中田: 実行サイクル数

入力	ステップ数 (前回)	ステップ数 (今回)	実行サイクル数 (今回)
$000F$	487	490	2387
$FFFF$	1724	1727	8292
$00XX$	74	30877	31212
$ABCX$	107	30910	31391

素数計算

中田: 実行サイクル数

入力	ステップ数 (前回)	ステップ数 (今回)	実行サイクル数 (今回)
2	378	381	1795
64	24240	24243	115153
255	100047	100050	475274
65535	24769799	24769801[*1]	117659446 [*1]

結果について

中田:

16 進数の変換における出力がエラーとなる演算と CPU_MONITORING をコメントアウトした演算を除き、全ての演算でステップ数が 3 増えた。これは、全ての演算において演算終了後に同じ命令を 3 回繰り返してしまうのが原因である。

また、16 進数の変換における出力がエラーとなる演算では、演算終了後に同じ命令が繰り返され、verilog でのステップ数が膨大になってしまった。

考察

中田:

ステップ数と実行サイクル数について

ステップ数は実行された命令の数であり、実行サイクル数は実行された命令の命令実行サイクルを全て足し合わせた総数になるはずである。

1 つの命令に対し命令実行サイクルは 4〜7 であるが、実際、正常に終了した演算の命令実行サイクルをステップ数で割った値の平均は 4.83 となったため、実験結果は妥当であったと言える。

最後に 3 回同じ命令が繰り返されてしまう点について

test_fpga_ex3.v ではステップ数と実行サイクル数のカウントをタスク SHOW_CPU_STATUS を呼び出すことで行っている。

この SHOW_CPU_STATUS は呼び出された時にクロックが落ち、EX3 が実行状態であると実行されるタスクである。

この問題は、この SHOW_CPU_STATUS を余分に 3 回呼び出しているために発生していると考える。

以下、その 3 つの実行サイクルについて考察する。

・1 つ目の実行サイクルについて

これは次の while 文内で SHOW_CPU_STATUS を呼び出しているためであると考える。

ソースコード 17　test_fpga_ex3.v 264 行目

```
while (s | (|(~FPGA_EX3.fgo))) SHOW_CPU_STATUS(0);
```

この while 文は出力フラグレジスタが初期状態の 3 であれば、s=1 の間は繰り返され、HLT によって s=0 となるとループが終了する。

しかし、実際は最後の実行サイクルにあたる SHOW_CPU_STATUS が実行された後、クロックが立ち HLT

*1 CPU_MONITORING を消したため、他のものとくらべ 1 少ない

によって s が 0 に切り替わる前に while 文の条件式の判定が実行されてしまい、次のクロックが落ちるときに余分に SHOW_CPU_STATUS が実行されてしまうのである。これによって 1 回余分に実行サイクルが生まれていると考えられる。

・2 つ目の実行サイクルについて
これは次の文によって必ず発生する。ここでは引数 1 をとっているが、引数が 1 の場合は CPU_MONITORING のコメントアウトの有無に関わらず、内部の状態がターミナルに表示される。そのため、CPU_MONITORING をコメントアウトした場合はここでの内部の状態が表示され、実行サイクル数が 2 つ増えたものとなる。

ソースコード 18　test_fpga_ex3.v 265 行目

```
SHOW_CPU_STATUS(1);
```

・3 つ目の実行サイクルについてこれは次のタスク KEY_ACTION[1] 内の 1 つ目の SHOW_CPU_STATUS で発生していると考えられる。

ソースコード 19　test_fpga_ex3.v 268 行目

```
KEY_ACTION(1); /// run state --> stop state
```

ソースコード 20　test_fpga_ex3.v 225 行目

```
SHOW_CPU_STATUS(0); KEY[key_idx] <= 0;
```

1 つ目の SHOW_CPU_STATUS は EX3 が実行状態であるため実行されるが、次に KEY[1] ← 0 が実行され EX3 が中断状態になり、それ以降に SHOW_CPU_STATUS を呼び出しても実行されないのである。

・3 つの実行されている実行サイクルについて
HLT 命令以降は s=0 となり、sc が常にクリアされ 0 になるため、1 つ目の実行サイクルの AR ← PC が常に実行されていると考えられる。

エラー出力となる演算において、実行サイクル数が膨大になる点について

これはエラーの出力にシリアルポートを利用しているため出力に時間がかかり、その間出力フラグレジスタが 1 となりソースコード 9 の while 文が繰り返されていることが原因だと考えられる。
シリアルポート通信に時間がかかってしまう点については、シミュレーションレポート 2 と内容が被るため、ここでは省略する。

2. シリアルポートを用いた test_io1 およびパラレルポートを用いたものに変更したプログラムのシミュレーション時間の違いについての考察

test_io1:シリアルポート

実行結果

入力	ステップ数	実行サイクル数	シミュレーション時間
$ctrl - D$	37072	61991	6235000
$a \rightarrow ctrl - D$	43281	92992	9335100
$a \rightarrow b \rightarrow ctrl - D$	49475	123990	12434900

test_io1:パラレルポート

実行結果

入力	ステップ数	実行サイクル数	シミュレーション時間
$ctrl - D$	73	344	70300
$a \rightarrow ctrl - D$	123	590	94900
$a \rightarrow b \rightarrow ctrl - D$	161	776	113500

UART-RX および UART-TX の通信開始から終了までのシミュレーション時間: 3093200

考察

クロックとシミュレーション時間について

以下の記述からクロックはシミュレーション時間で 50 ごとに切り替わる。よってシミュレーション時間での 100 がクロックの 1 周期であり、シミュレーション時間を 100 で割ったものがクロックの発生回数である。

ソースコード 21　test_fpga_ex3.v 92 行目

```
always # 50 clk = ~clk;
```

シミュレーション時間の違いについて

実験結果よりシリアルポートを使用した方がシミュレーション時間がパラレルポートに比べ 100 倍近く増えていることがわかる。これは、パラレルポートでは入出力に 1 クロック分のシミュレーション時間がかかっているのに対し、シリアルポートでは UART の通信を行うごとにシミュレーション時間が 3093200 かかっているためであると考えられる。

シリアルポートにおいて、入力が 1 つの場合は UART-RX、UART-TX それぞれ一回ずつ通信を行うため、全ての UART 通信にかかるシミュレーション時間は 3093200 × 2 =6186400 となる。

複数の入力の場合は、2 つ目以降の UART-RX 通信ではその一つ前の入力の UART-TX とほぼ並列に通信が行われるため、k 個の入力があった場合、全ての UART 通信にかかるシミュレーション時間はおよそ 3093200 × (k+1) となると言える。test_io1 では入出力以外の命令にかかるシミュレーション時間は 3093200 に比べ極めて少ないため、実際に実行結果でも入力数を k 個とするとシミュレーション時間が 3093200 × (k+1) に近い値をとっている。

UART 通信にかかるシミュレーション時間について

・UART-RX について

uart_rx モジュールは 1bit ごとの入力を受け取り、8bit のデータとして返すモジュールである。この動作原理を説明する。

uart_rx では 11bit(start bit 1bit + 入力データ 8bit + parity bit 1bit + end bit 1bit) のレジスタ rx_shift が用意されており、そこに右シフトによって 1bit ごとにビットを格納していき、全てのシフトが完了すると rx_shift(8:1) を出力する。

この右シフトはボーフェイズごとに行われ、ボーフェイズは 4 つのフェイズに分かれている。1 つのフェイズではクロックの発生を baud_tick によって 702 回カウントした後、フェイズごとに違った動作を行う。フェイズ 1、フェイズ 3 ではカウント後に baud_tick を 0 にリセットし次のフェイズに移行する。フェイズ 2 ではカウント後、rx_shift の右シフトを行い、baud_tick を 0 にリセットして次のフェイズに移行する。フェイズ 4 ではカウント後、baud_tick を 0 にリセットし、bit_count を 1 増やして次のボーフェイズのフェイズ 1 に移行する。

この一連のフェイズを 1 つのボーフェイズとして、このボーフェイズが 11 回繰り返される。

つまり、1 つのボーフェイズでは (702+1) × 4=2812 回クロックが発生することになる。これは 1bit 読み込むのに必要なクロック数であり、動作周波数をボーレートで割った値 27MHz ÷ 9.6KHz=2812.5 に近い値となっている。値に誤差がある原因は、BAUD_PERIOD の値を小数点以下切り捨てにして定めているためである。また、ボーフェイズは 11 回繰り返されるため、1 回の UART-RX 通信でクロックは 2812 × 11=30932 回発生することになる。これは実行結果の UART-RX 通信の開始から終了までにクロックが発生した回数に等しい。

・UART-TX について

uart_tx モジュールは 8bit の入力を受け取り、1bit ごとのデータを返すモジュールである。この動作原理について説明する。

uart_tx にも uart_rx と同様に 11bit のレジスタ tx_shift が用意されており、tx_shift は {1,tx_parity, 入力データ 8bit,0} と初期化される。出力ポートの tx_dout は tx_shift[0] が連続代入されており、tx_dout の値は tx_shift を右シフトするごとに切り替わる。この右シフトを 11 回繰り返すことで、1bit ごとの出力を行っている。

uart_tx でもボーフェイズが存在するが 4 つのフェイズには分かれていない。uart_tx では 1 つのボーフェイズ内で baud_tick によってクロックの発生を 2811 回カウントし、次のクロックで tx_shift を右シフトし、baud_tick の値を 0 にリセットする。このボーフェイズを 11 回繰り返している。

uart_tx のボーフェイズでもクロックが 2812 回発生し、一回の UART-TX 通信でクロックは 2812 ×

11=30932 回発生する。これについても実行結果に一致している。