卒 業 研 究 報 告

題　　目

ディジタル回路の設計

指 導 教 員

原 央 教授

報 告 者

木村 知史

平成　13 年　2 月　9 日

高知工科大学 電子・光システム工学科

目次

1. 概要

デジタル回路設計のために必要な知識獲得のため、大きく 3 つの項目について調査し、実験した。1 つは現在の主流である MOS 集積回路で、設計のための基盤となる MOS トランジスタの動作について調査した。もう 1 つは実際の回路で設計および測定の技術を取得するために、標準 IC を使った回路の作成、そしてもう少し規模の大きなものとして 8bitCPU アーキテクチャの設計をした。

2. MOS トランジスタの基礎

2.1 MOS トランジスタの構造と特性

(1) ドレイン電流特性

MOS FET は半導体に絶縁膜を隔ててゲート電極から電圧を加えることによって、半導体表面にチャネル（伝導層）を形成して電流を制御する。

MOS FET の構造を図 2.1 に示す。

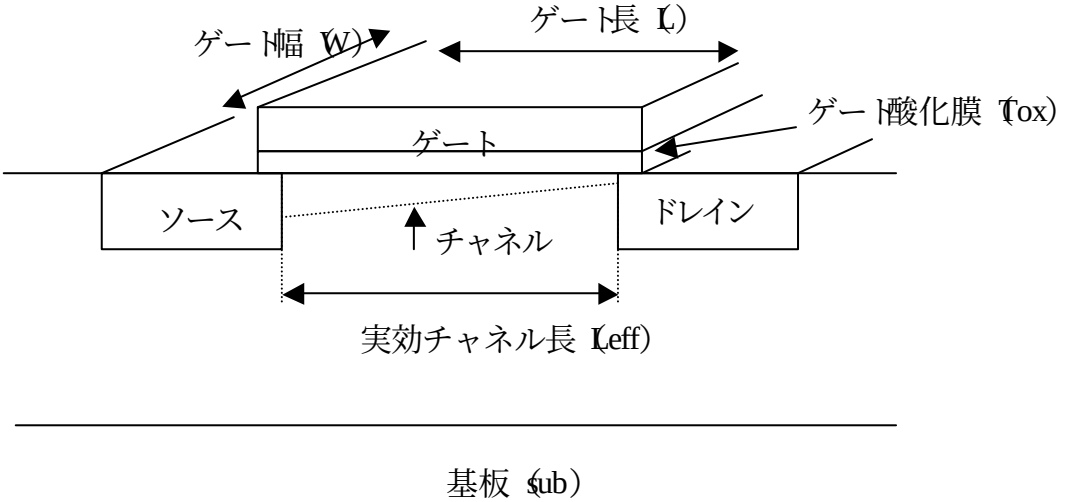


図2.1 MOS FETの断面構造

MOS FET は、電流の担い手（キャリア）となるのが電子か正孔かで、N チャネル MOS FET（NMOS）、P チャネル MOS FET（PMOS）に分かれる。シリコン基板上に形成された高不純物濃度の拡散層（ソース、ドレイン）が、N 形であれば NMOS、P 形であれば PMOS である。ソース、ドレインは構造の違いではなく、チャネルにキャリアを供給するのがソース、チャネルからキャリアが流れ込むのがドレインである。また NMOS の場合、基板は P 形、PMOS の場合は N 形である。

ゲートは Al などの金属や Poly-Si などで、ゲート・基板間に電位差を与えるとチャネルがソース・ドレイン間に形成され、チャネルにキャリアが移動することによって電流となる。チャネルの形成は、例えば NMOS であれば、ソース、基板に 0V、ドレインに 5V であったとして、ゲートに 0V から正電圧をかけていくと、ゲート酸化膜下の P 形基板のキャリアである正孔が反発し、少数キャリアである電子が引き寄せられ、チャネルが形成される。この時の電圧がスレッショルド電圧（Vth）である。

図 2.2 に NMOS のドレイン電流特性を示す。

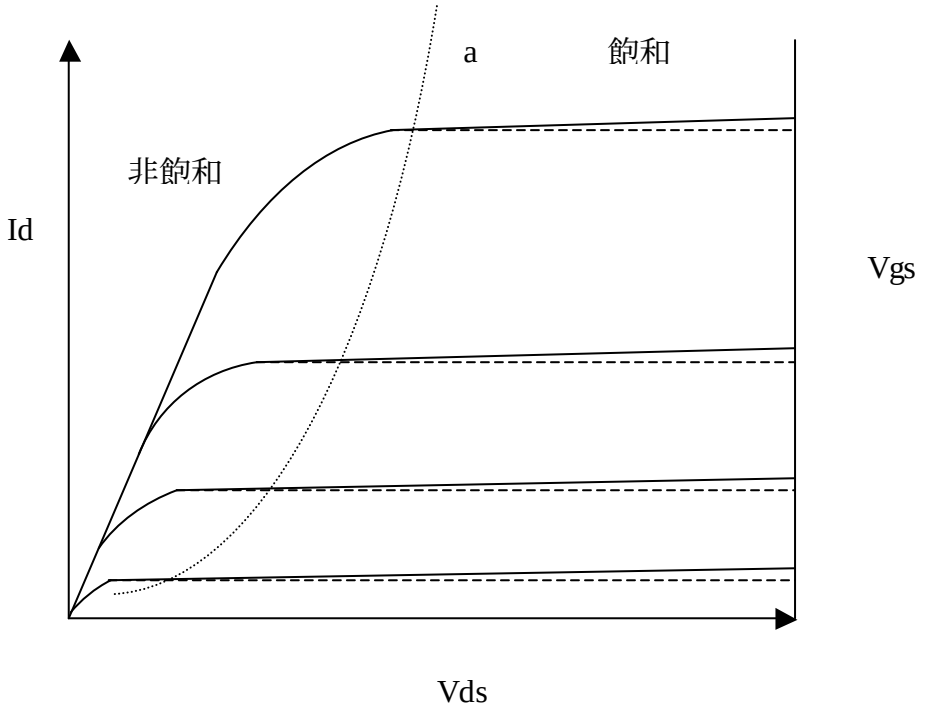


図2.2　ドレイン電流特性

a の点線を境にして飽和と非飽和の状態がある。非飽和の状態では、ドレイン電流（Id）は

$$Id = \mu \frac{W}{L} \frac{\varepsilon_{ox}}{T_{ox}} \left\{ (Vgs-Vth)Vds - \frac{Vds^2}{2} \right\}$$

で表され、Vgs はゲート・ソース間の電圧、Vds はソース・ドレイン間の電圧、μは電子の移動度、Tox、εox、はそれぞれゲート酸化膜の膜厚、比誘電率である。非飽和の状態では、各 Vgs におけるグラフの形は、原点を通る Vds の 2 次関数であり、頂点における Vds の値は Vgs-Vth である。また頂点（すなわち飽和が開始するとき）のドレイン電圧と、そのときの Id の関係を示したのがグラフ a である。Vds > Vgs-Vth となると、飽和状態となり、この時の Id は

$$Id = \mu \frac{W}{L} \frac{\varepsilon_{ox}}{T_{ox}} (Vgs-Vth)^2$$

となり、Id は Vds によらずほぼ一定になる。このときドレイン近傍でチャネルは切れており、電子が集まっていない。図 2.3 のように、ソース近傍、ドレイン近傍の酸化膜（Tox）ではコンデンサが形成されており、それぞれ Cs、Cd とすると、Cs の両端の電圧は Vg-Vs-Vth、Cd の両端の電圧は Vg-Vd-Vth である。

Vs、基板に 0V、Vg は Vth を超える値でチャネルを形成させ、Vd に正電圧を与える。Cd のドレイン側には正電圧が加わるため、Cd の両端にかかる電圧は Cs より低く、ソース近傍に集まる電子よりドレイン近傍の電子のほうが少ない。Vd = Vg-Vth のとき、ドレイン近傍でチャネルができない。しかしソースから流れ込む電流は一定であるから、ソース・ドレイン間の電圧によりドレイン近傍の電界は上昇するため、ソース近傍の電子とドレイン近傍の電子は速度が違う。

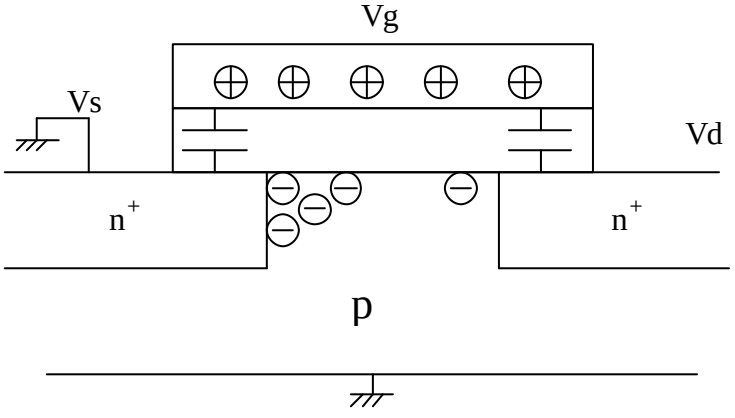


図2.3　実効チャネル

さらにVdに電圧をかけると、電子が集まる実質的なチャネルはさらに短くなる。この実効的なチャネル長の変化（チャネル長の変調）のためにVdの増加とともにIdは増加し、図2.2の飽和の状態で、点線のように一定でなく、実線のようにわずかに右上がりになる。チャネルが短いトランジスタはど、このようま電流の増加は大きい。

PMOSでは印加する電圧がNMOSと逆である、キャリアが正孔であるなどが違うだけで動作は変わらない。これからの議論はすべてNMOSのみで進める。

（2） しきい値電圧

図2.4にNMOSのエネルギーバンド図を示す。簡単に話を進めるため、真空の準位と酸化膜の伝導体の底を一致させて取り扱う。金属の仕事関数φmとSiの仕事関数φsは等しいものとし、Efはフェルミ準位、Ecは伝導帯、Evは価電子帯、Eiは真性フェルミ準位、真空準位は電子が固体中から出るために必要なエネルギーである。

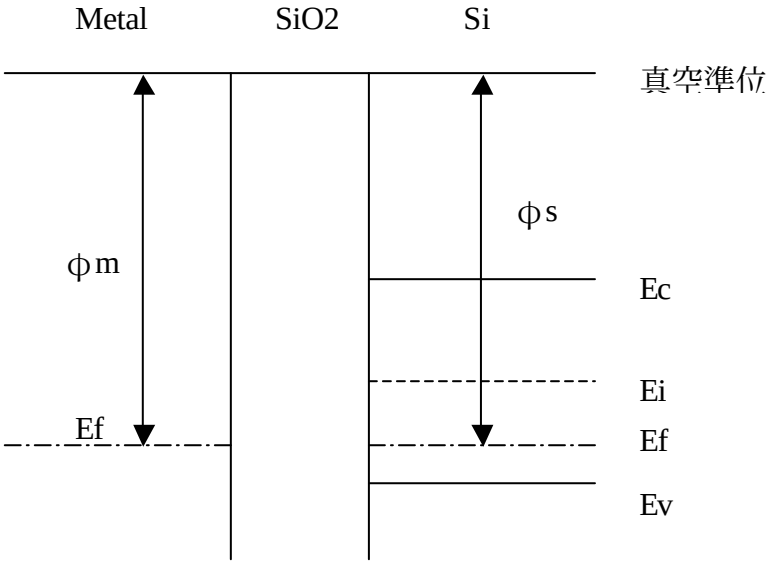


図2.4　エネルギーバンド図1

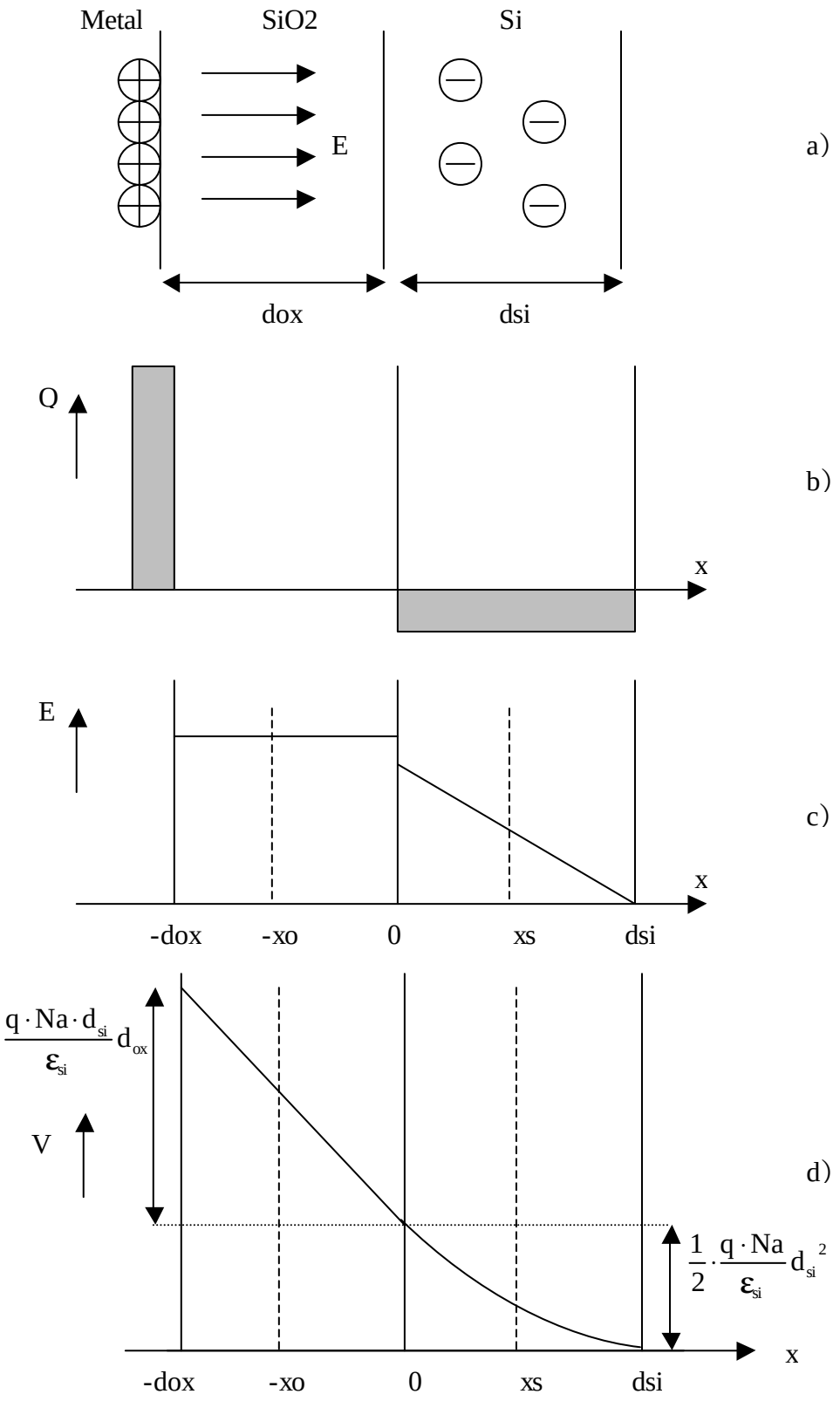


図2.5　電荷・電界・電位分布

ゲートに正電圧(Va)を加えたとき、電荷 Q の分布から電位分布を求める。図 2.5 はその過程を示したものである。（a）はゲート電極と SiO2 の境界に正電荷が局在し、Si 表面にそれと等価な負電荷が現れた状態で、（b）は電荷の分布を表している。Si は P 形であり、負電荷はアクセプタイオンである。図中の dox は SiO2 の厚さ、dsi は空乏層幅である。また Si の不純物濃度（Na）は一定で、SiO2 中に電荷はないとする。

SiO2、Si にかかる電界 E を求める。E は単位面積を通過する電気力線の本数と向きであり、Eox を SiO2 の電界、Esi を Si の電界とすると、ガウスの法則

$$E=\frac{Q}{\varepsilon}$$

より、

$$E_{ox}=\frac{Q_m}{\varepsilon_{ox}}$$

であり、電荷量 Qm は Si の電荷量 Qsi と等価であるため、

$$E_{ox}=\frac{q\cdot Na\cdot d_{si}}{\varepsilon_{ox}} \qquad (2.1)$$

となる。

図 2.5 は（c）は縦軸 E、横軸 x としたもので、xo は、-dox から 0、xs は 0 から dsi を動く点で、SiO2 中は電荷がないため xo を通る電気力線は一定である。Si 中の xs を通る電気力線は、一様に電荷があるため場所によって違う。よって Esi は、

$$E_{si}=\frac{Q_{si}}{\varepsilon_{si}}$$

$$=\frac{q\cdot Na\cdot(d_{si}-x_s)}{\varepsilon_{si}} \qquad (2.2)$$

となる。

x = 0 で Eox と Esi が連続していないが、これは誘電率 ε ox、ε si の差によるものである。

電位は電界を積分することによって得られる。SiO2 の電位を Vox、Si の電位を Vsi とすると、

$$V_{ox}=-\int_0^{-x_o}E_{ox}dx=\int_{-x_o}^{0}E_{ox}dx$$

$$=\int_{-x_o}^{0}\frac{q\cdot Na\cdot d_{si}}{\varepsilon_{ox}}dx$$

$$= \frac{q \cdot Na \cdot d_{si}}{\varepsilon_{ox}} \cdot x_o \qquad (2.3)$$

$x_o = d_{ox}$ **のとき**

$$V_{ox} = \frac{q \cdot Na \cdot d_{si}}{\varepsilon_{ox}} \cdot d_{ox} \quad (2.4)$$

$$Vsi = -\int_{d_{si}}^{x_s} E_{si} dx = -\int_{d_{si}}^{x_s} \frac{qNa(d_{si} - x_o)}{} dx$$

$$= \int_{d_{si}}^{x_s} \frac{q \cdot Na \cdot (d_{si} - x_{si})}{\varepsilon_{xi}} dx$$

$$= \left[\frac{1}{2} \frac{q \cdot Na}{\varepsilon_{si}} (d_{si} \cdot x - \frac{1}{2} x^2) \right]_{x_s}^{d_{si}} = \frac{1}{2} \frac{q \cdot Na}{\varepsilon_{si}} (\frac{1}{2} d_{si}^2 - d_{si} \cdot x_s + \frac{1}{2} x_s^2)$$

$$= \frac{1}{2} \frac{q \cdot Na}{\varepsilon_{si}} (d_{si} - x_s)^2 \qquad (2.5)$$

$xs = 0$ **のとき**

$$V_{si} = \frac{q \cdot Na \cdot d_{si}}{\varepsilon_{si}} \cdot \frac{1}{2} d_{si} \qquad (2.6)$$

式 2.3**、**2.5 **をグラフにしたものが（**d**）である。式** 2.3 **は** x = 0 **を通る左上がりの直線で、式** 2.5 **は** x = 0 **を通り、頂点が**(d_{si},0) **の** 2 **次曲線である。また** 2 **つの線は** x = 0 **で連続している。**

図 2.6 にφm とφs に差がある場合のエネルギーバンド図を示す。

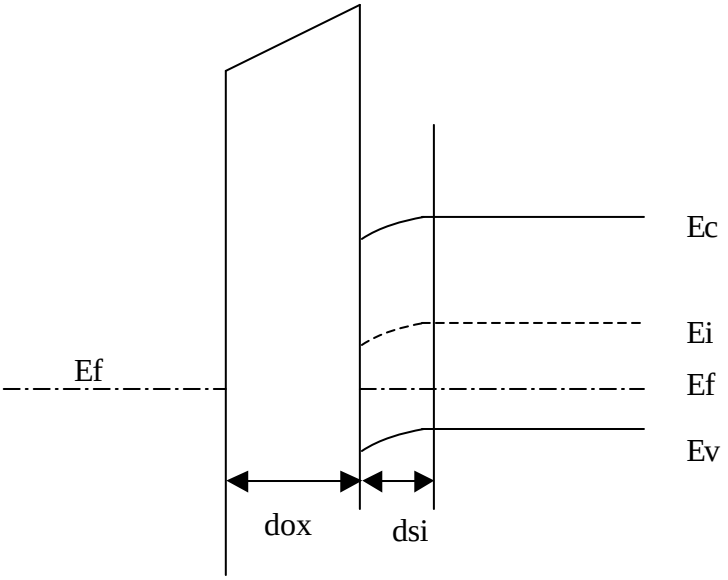


図2.6　エネルギーバンド図 2

φm とφs の差があるときは（例えばφm<φs の場合）、双方のフェルミ準位が一致するため、図 2.6 のようにエネルギーバンドは曲がる。このときの Vox と Vsi の和は、フェルミレベルを一致させるために発生したもので、φs－φm に等しい。エネルギーバンドを図 2.4 のように平坦にするには、ゲートの Ef のレベルを上げて Vox と Vsi を打ち消さなくてはならない。このためゲートに負電圧（Va）をかける。そのときの Va は

$$f_s - f_m = \mathrm{Va}$$

$$\mathrm{Va} = f_m - f_s \qquad (2.7)$$

であり、Va をフラットバンド電圧という。

フェルミ準位が一致した状態で、金属と Si の真空準位とフェルミ準位の電位差を考える。図 2.7 はフェルミ準位が一致した状態である。

$$\Phi_m = \chi_s - V_{ox} - y_s + \frac{1}{2}Eg + \Phi_p$$

$$\Phi_m - \chi_s + V_{ox} + y_s - \frac{1}{2}Eg - \Phi_p = 0 \qquad (2.8)$$

$$V_{ox} + y_s = -\Phi_m + (\chi_s + \frac{1}{2}Eg + \Phi_p)$$

$(\chi_s + \frac{1}{2}Eg + \phi_p)$ は Si の仕事関数であるから、

$$V_{ox} + y_s = -\phi_m + \phi_s \qquad (2.9)$$

ys は式 2.6 の Vsi と等しい。

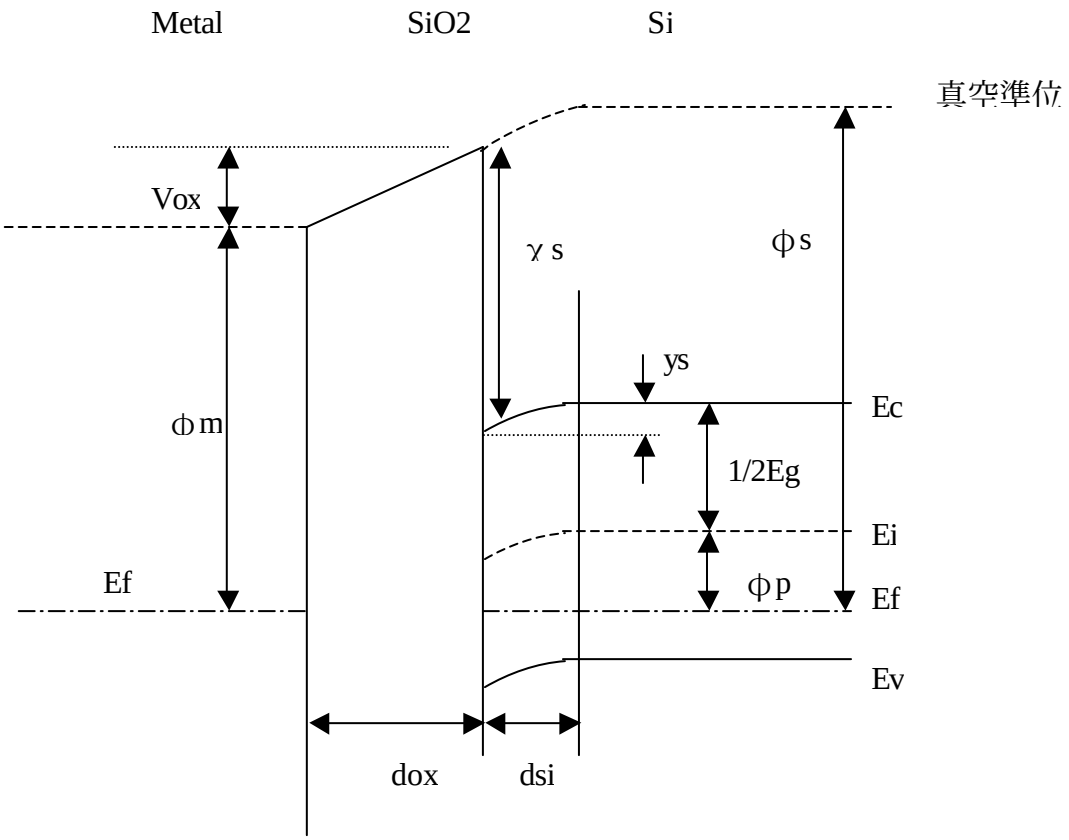


図2.7　エネルギーバンド図 3

ゲートに電圧を加えると、空乏層幅 dsi が増加し、Si のエネルギーバンドが下がる（ys が増加する）。加えた電圧を Vg とすると、式 2.8 より

$$\phi_m - \chi_s + V_{ox} + y_s - \frac{1}{2}Eg - \phi_p = Vg \qquad (2.10)$$

となり、式 2.10 は

$$V_{ox} + y_s = Vg - \phi_m + \phi_s$$

$$V_{ox} + y_s = Vg - Va \qquad (2.11)$$

となる。式 2.11 より、エネルギーバンドの曲がりはバンドがフラットな状態からの差分であることがわかる。さらに電圧を加えると、半導体の表面で Ei が Ef を下まわり、少数キャリアである電子の密度より多数キャリアの正孔の密度が大きくなり、P 形から N 形へ反転する。この反転層における電子密度と正孔密度が等しくなるとき、すなわち

$$y_s = 2f_p \qquad (2.12)$$

であるときの電圧をしきい値電圧（Vth）とし、このときの空乏層幅を dm とすると、式 2.11、式 2.4 より、

$$V_{ox} + 2f_p = V_{th} - Va$$

$$V_{th} = Q \cdot \frac{d_{ox}}{\varepsilon_{ox}} + 2f_p + Va \qquad (2.13)$$

であり、式 2.12 より、

$$2f_p = \frac{q \cdot Na \cdot d_m}{\varepsilon_{si}} \cdot \frac{1}{2} d_m$$

$$d_m = \sqrt{\frac{4 \cdot \varepsilon_{si} \cdot f_p}{q \cdot Na}}$$

であり、電荷 Q は

$$Q = -q \cdot Na \cdot d_m = -\sqrt{4 f_p \cdot \varepsilon_{si} \cdot q \cdot Na} \qquad (2.14)$$

よって式 2.13 の Vth は、式 2.7・2.8・2.14 より、

$$V_{th} = \frac{d_{ox}}{\varepsilon_{ox}} \sqrt{4 \cdot f_p \cdot \varepsilon_{si} \cdot q \cdot Na} + f_m - \chi_s - \frac{1}{2} E_g + f_p$$

となる。

3. サイコロ・マシンの製作

3.1 サイコロ・マシンとは

LED をサイの目状に配置し、スタートボタンで１から６の目をルーレットのように表示させ、ストップボタンで現在表示されている１から６の目の状態で止まるものである。

3.2 目的

サイコロ・マシンの設計、製作、測定を通じて、標準ロジック IC や発振器等の部品の種類や使用法、回路シミュレータ multiSim を用いて回路解析、プリント基板上への実装、回路の計測などの技術を習得する。

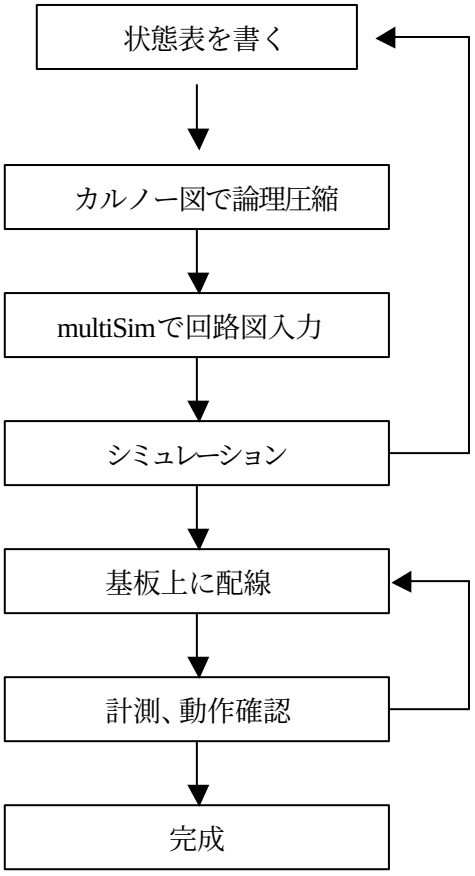


図 3.1 製作フロー

3.3 製作の経緯とフロー

まず何を設計するかであるが、標準ロジック IC をただ組み合わせたものは面白くないと思い、同期式のステートマシンにすることにした。また、回路の動作が見た目に分かりやすいように LED を用いたいと思った。そしてステートマシンと LED の駆動を考えたとき、サイの目が１から６の６個の状態があり、その状態が遷移していく回路、すなわちサイコロ・マシンに至った。また、スタ

ートボタンとストップボタン、リセットボタンの３つの外部入力で制御させることにした。

サイコロ・マシン製作のフローは図 3.1 に示す。

3.4 状態図の作成

３つの外部入力と LED への出力は決定したが（図２）、次はその入力と出力を制御する
回路を決定しなくてはならない。そこでまず考えたのは、図 3.2 に示すように、制御部を１つのモジュールとし、入力から出力までを一気に行う方法である。

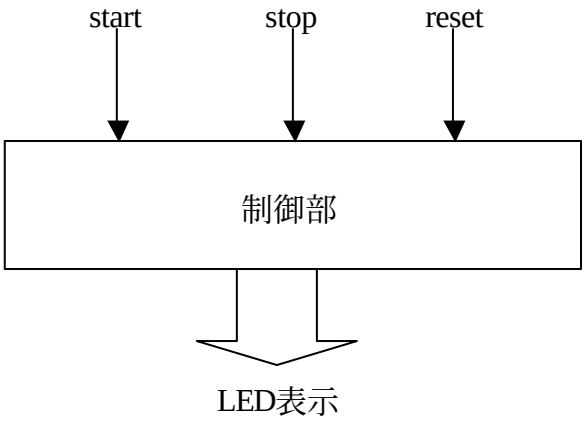


図3.2　ブロック図

そこで図 3.3 に示すように、サイコロの目に対応する遷移図で表すことにした。

この方法で、遷移図を書いてカルノー図で圧縮し、multiSim で回路図を書こうとしたが、手作業で行うとなると、論理圧縮の際、細かなミスが多く発生し、そのミスが全体の回路に影響を及ぼす。

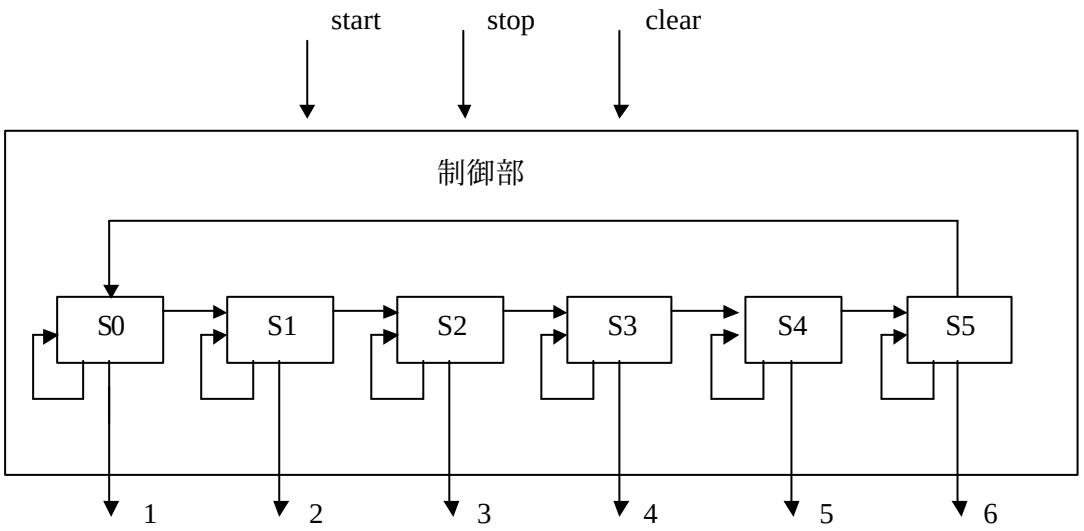


図3.3　状態遷移図１

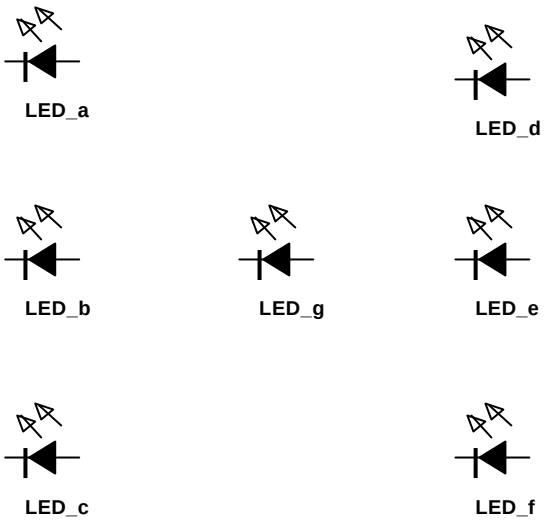


図3.4 LEDの配置

サイコロの目は１から６の６つであるから、定義する状態は６つである。ここではS0、S2・・・S５とする。S0の状態から出力された信号は、図４の中央のLED__ｇを駆動することにより「１」を表現し、状態S1（図３）ではLED__ｃ、LED__ｄを駆動して、「２」を表現、状態S2ではLED__ｃ、LED__ｄ、LED__ｇを駆動させ「３」を表現、同様に状態S3でLED__ａ、ＬＥＤ__ｃ、ＬＥＤ__ｄ、ＬＥＤ__ｆで「４」、状態Ｓ４でＬＥＤ__ａ、ＬＥＤ__ｃ、ＬＥＤ__ｄ、ＬＥＤ__ｆ、ＬＥＤ__ｇ、状態Ｓ５でＬＥＤ__ａ、ＬＥＤ__ｂ、ＬＥＤ__ｃ、ＬＥＤ__ｄ、ＬＥＤ__e、ＬＥＤ__ｆで「６」をそれぞれ表現する。

制御部の大まかな動作を述べる。まずstart信号が入力されると、stop信号が入力されるまで、S0～S5の６つの状態を遷移し、それに合わせてLEDは「１」から「６」を表示する。状態を遷移する時間は短いため、サイコロの表示は判断できないようにする。しかし短すぎてもすべてのLEDが光っているようにしか見えないので、適当に点滅しているような速度でなければならない。そしてstop信号が入力されると、その時点での状態を保ち続け、LEDを点灯させたままにする。この状態からさらにstartを入力すると、ふたたび状態遷移を始め、stop入力で状態保持をする。

reset入力は、stop入力で状態が保持されているときに入力し、点灯したLEDを非点灯にするためのものである。制御部に７番目の遷移状態としてS6を加えて、resetが入力されるとS6に遷移し、全LEDを非点灯とすることも考えられる。しかし回路が複雑になる。前にも書いたが、１つのステートマシンで入力の制御からLEDの制御までを手作業で設計しようとすると、ミスも生じやすく、１つのミスで全体が狂い、何回もやり直すことになる。

設計、テスト、デバッグのやり易さを考えると、図3.5のように制御部を分ける方法がある。制御部１は入力信号の制御、制御部１からの出力を用いて制御部２の状態を遷移させ、制御部２の出力信号をLED駆動部でLED駆動のため

の信号にする。

これが今回の製作で採用した方法である。

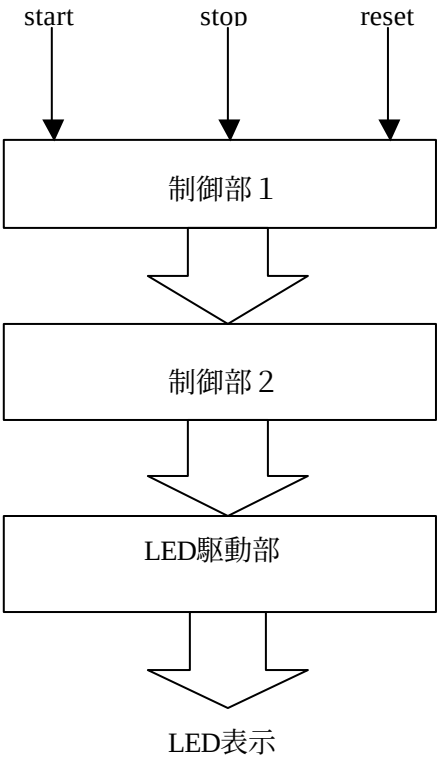


図3.5　ブロック図２

図 3.6 に状態遷移図を示した。制御部１は入力が start、stop、reset の３つ、出力が st_sp、zero の２つで、C0、C1、C2 の 3 つの状態を遷移する回路である。st_sp は C2 の状態をデコードしたもので、zero は C0 の状態をデコードしたものである。

現在の状態が C0 とすると、Start の信号が入れば、C0 から C2 へ移り、C2 の状態をデコードして（st_sp）、st_sp を入力として制御部２を動作させ、外部から stop の入力がない限り C2 を保ち続け、これにより st_sp も“１”を保持し、制御部２は S0～S5 を遷移し続ける。ここで stop 信号が入力されると、制御部１の状態 C2 から C0 へ遷移し、それに伴い st_sp も“0”となり、制御部２の状態 S0～S5 のいずれかの状態を保持する。またこの状態で start の入力があると、制御部１の状態 C0 から C2 へ移り、制御部２の状態 S0～S5 が遷移を始める。

制御部１の状態が C0 の時（stop の入力後）、reset の入力があると、C0 の状態から C1 へと移り、C1 をデコードした信号（zero）が LED 駆動回路への入力信号となり、保持されていた LED の点灯（「１」から「６」のいずれかの状態を点灯させている）を、非点灯の状態、つまりすべての LED を光らせなくする。制御部１の状態 C1 で、start の入力があると、状態 C2 へと遷移する。

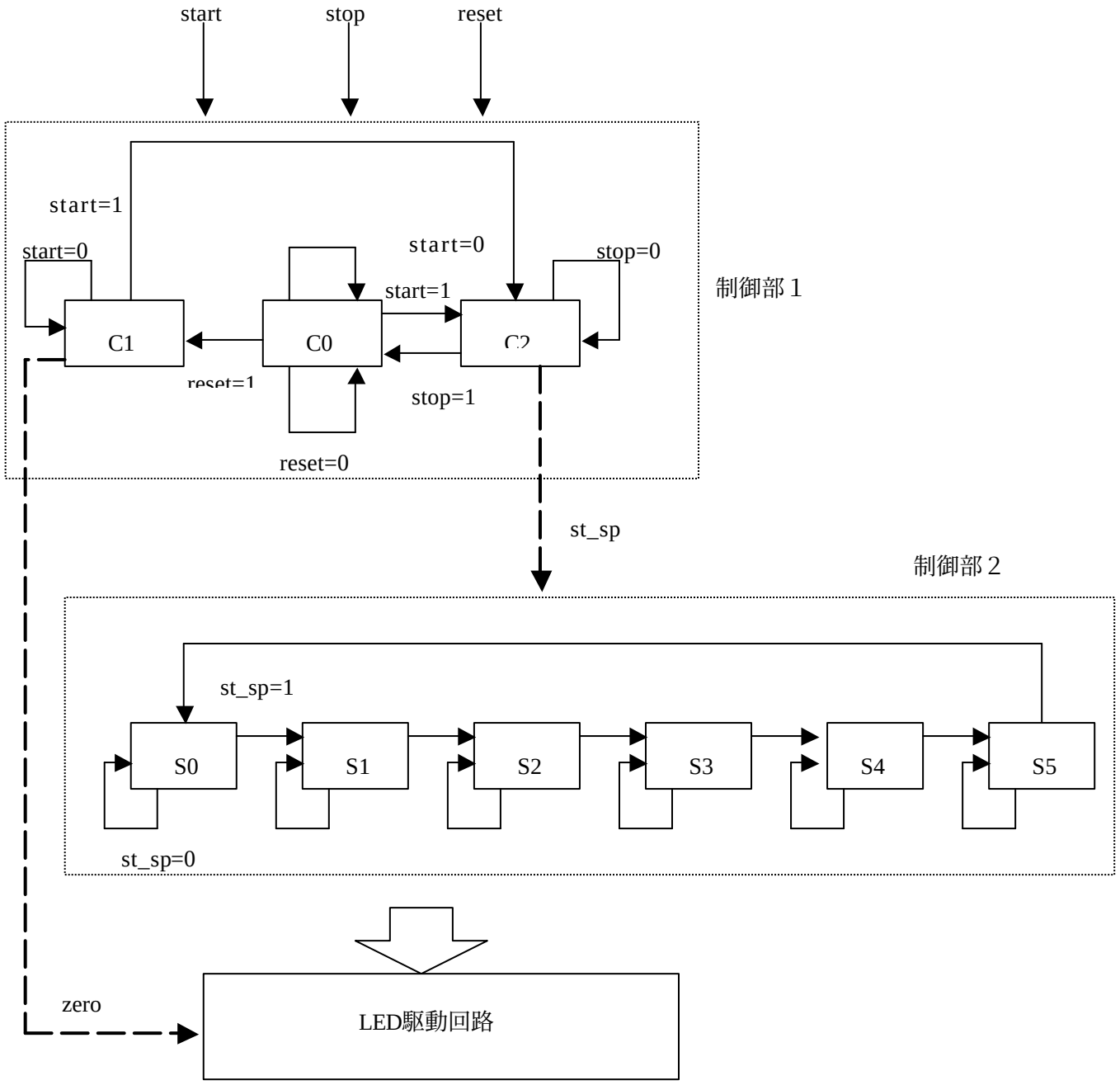


図 3.6　状態遷移図２

3.5 回路設計

状態遷移の状態図をもとにして、制御部１、制御部２、LED **駆動回路のそれぞれの真理値表とカルノー図、そして回路図を作成する。**

3.5.1　制御部１の回路

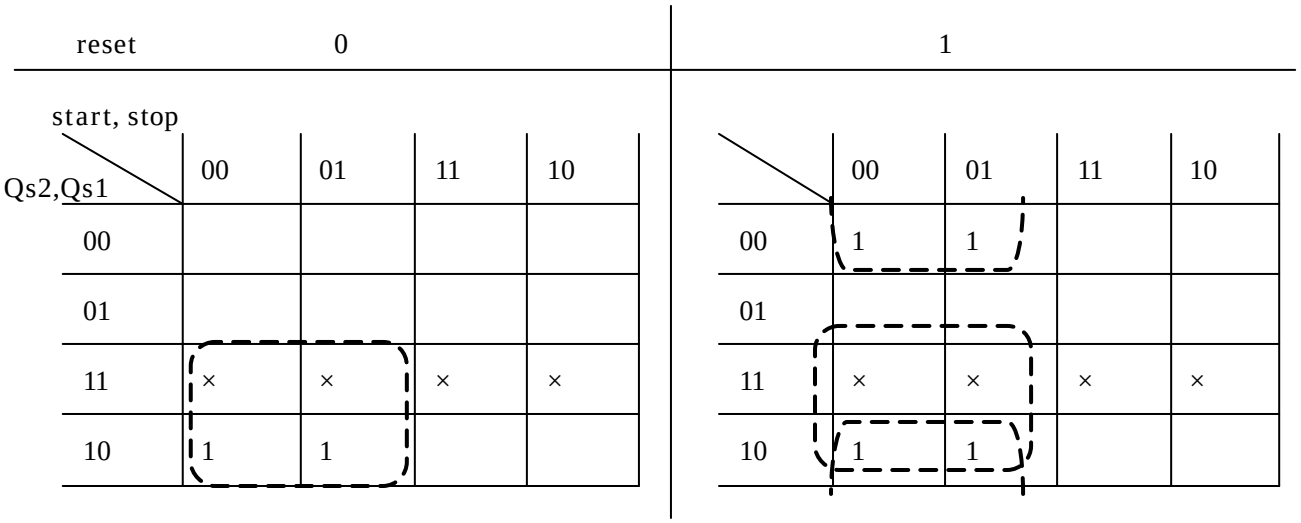
制御部１の真理値表を表 3.1**、カルノー図を図** 3.7**、回路を図** 3.8 **に示す。**

状態 C0**、**C1**、**C2 **をそれぞれ**"00"**、**"01"**、**"10" **に割り当て、**"11"**の状態は不定とする。また、**D **フリップフロップへの入力はそれぞれ** Ds２**、**Ds１**（表** 3.1 **の“次の状態”）とする。**

表 3.1 制御部 1の真理値表

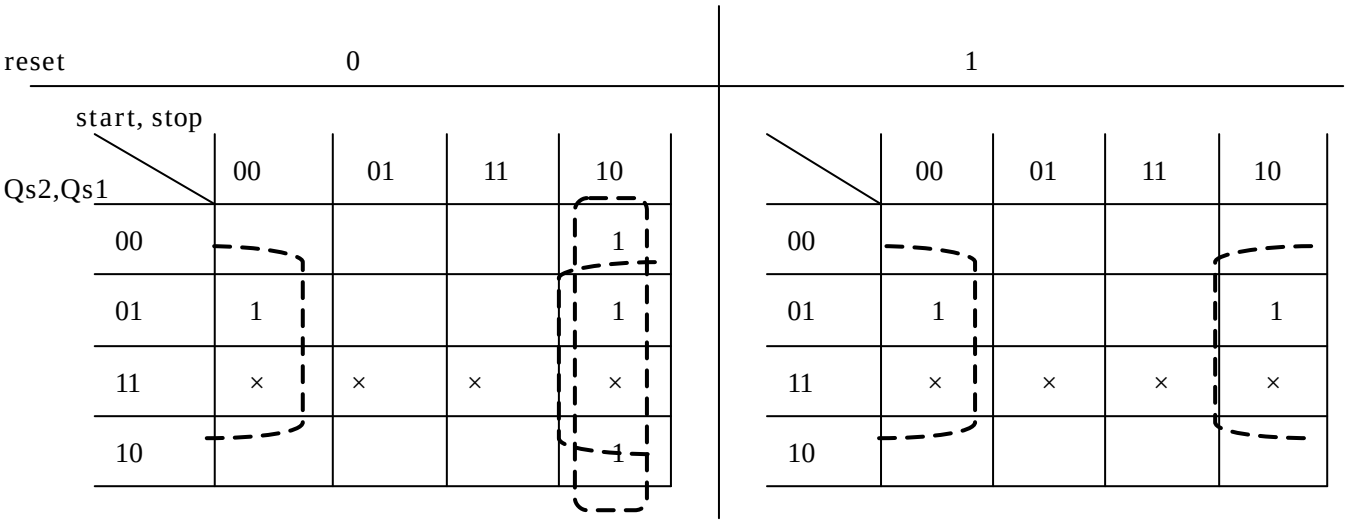
現在の状態		入力			次の状態 D-FFへの入力）	
Qs2	Qs1	stop	start	reset	Qs2+ Ds2）	Qs1+ Ds1）
0 (C0)	0	0	0	0	0	0
		0	0	1	1	0
		0	1	0	0	1
		0	1	1	0	0
		1	0	0	0	0
		1	0	1	1	0
		1	1	0	0	0
		1	1	1	0	0
0 (C2)	1	0	0	0	0	1
		0	0	1	0	1
		0	1	0	0	1
		0	1	1	0	1
		1	0	0	0	0
		1	0	1	0	0
		1	1	0	0	0
		1	1	1	0	0
1 (C1)	0	0	0	0	1	0
		0	0	1	1	0
		0	1	0	0	1
		0	1	1	0	0
		1	0	0	1	0
		1	0	1	1	0
		1	1	0	0	0
		1	1	1	0	0
1	1	0	0	0	×	×
		0	0	1	×	×
		0	1	0	×	×
		0	1	1	×	×
		1	0	0	×	×
		1	0	1	×	×
		1	1	0	×	×
		1	1	1	×	×

Ds2 について



$$Ds2 = Qs1 \cdot \overline{start} + \overline{Qs1} \cdot \overline{start} \cdot reset$$

Ds1 について



$$Ds1 = start \cdot \overline{stop} \cdot \overline{reset} + Qs1 \cdot \overline{stop}$$

図 3.7　表 3.1 のカルノー図

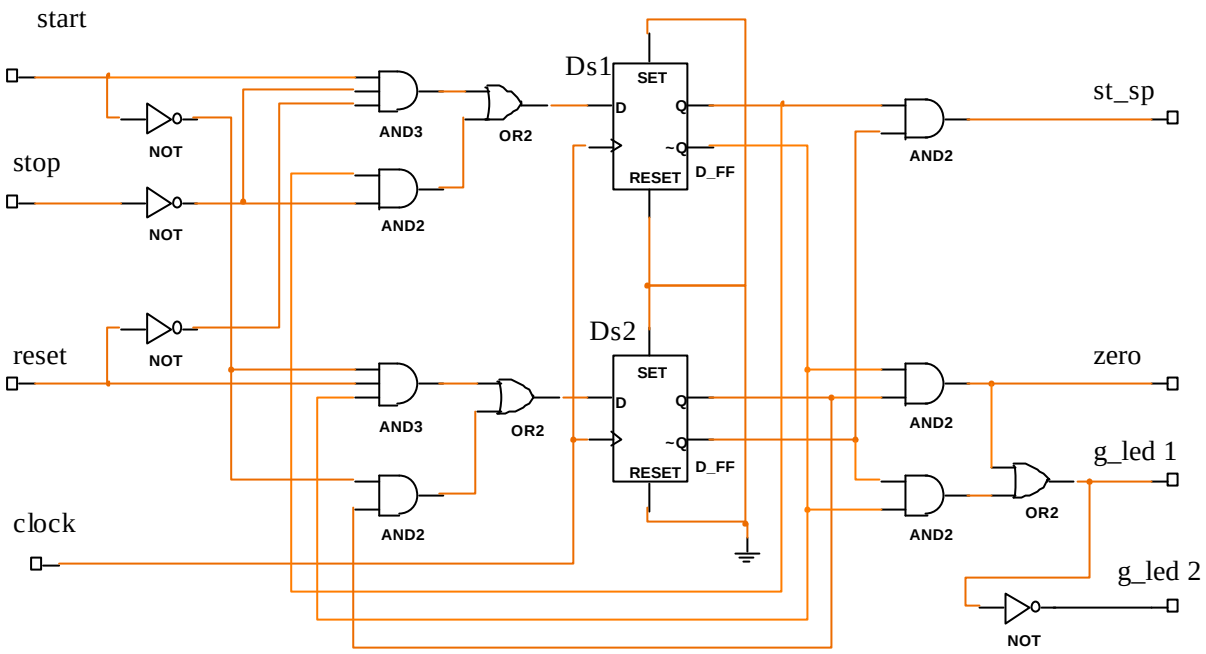


図 3.8　制御部 1の回路図

図 3.8 の出力の g_led1、g_led2 **は、図** 3.9 **に示すように、それぞれ** LED1、LED2 **に接続されており、プリント基板上の** start **ボタン、**reset **ボタンの上部に取り付け、次に押すべきボタンを知らせるものである。つまり現在** stop **が入力されてサイコロの表示（**1 **から** 6 **のいずれか）がなされているとすると（状態は** C0**）、次の入力は** start **か** reset **である。この時** g_led1 **が“** 1 **”となり、**start **ボタン上部の** LED1 **が光って** start **の入力を促す。**reset **ボタン上部に取り付けなかったのは** start **ボタンと混同するのを防ぐためである。そして** start **が入力されると、次の入力は** reset **であるから、**g_led2 **が“** 1 **”となり、**reset **ボタン上部にある** LED2 **が光る。**

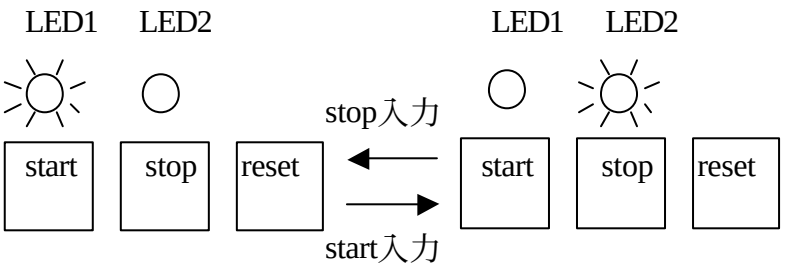


図3.9　入力によるLED1、2の動作

3.5.2 制御部２の回路

制御部２の真理値表を表 3.２に、カルノー図を図 3.10、回路図を図 3.11 に示す。
制御部２の状態数は s0 から s5 の６状態であり、それぞれ”000”、“001”、“011” 、“111”、 “110” 、“100”とし、”010”、”101”は不定とする。また、表 3.2 の「次の状態」は、D フリップフロップの入力である。

表 3.2 制御部 2の真理値表

現在の状態			入力	次の状態 (D_FFへの入力)		
Qc	Qb	Qa	st sp	Qc+ (Dc)	Qb+ (Db)	Qa+(Da)
0	0	0	0	0	0	0
			1	0	0	1
0	0	1	0	0	0	1
			1	0	1	1
0	1	1	0	0	1	1
			1	1	1	1
1	1	1	0	1	1	1
			1	1	1	0
1	1	0	0	1	1	0
			1	1	0	0
1	0	0	0	1	0	0
			1	0	0	0
0	1	0	×	×	×	×
1	0	1	×	×	×	×

Da について

Oc,Ob \ Oa,st_sp	00	01	11	10
00		1	1	1
01	×	×	1	1
11				1
10			×	×

$Da = \overline{Qc} \cdot st\_sp + Qa \cdot \overline{st\_sp}$

Db について

Oc,Ob \ Oa,st_sp	00	01	11	10
00			1	
01	×	×	1	1
11	1		1	1
10			×	×

$Db = Qb \cdot \overline{st\_sp} + Qa \cdot st\_sp$

Dc について

Oc,Ob \ Oa,st_sp	00	01	11	10
00				
01	×	×	1	
11	1	1	1	1
10	1		×	×

$Dc = Qb \cdot st\_sp + Qc \cdot \overline{st\_sp}$

図 3.10　表 3.2 のカルノー図

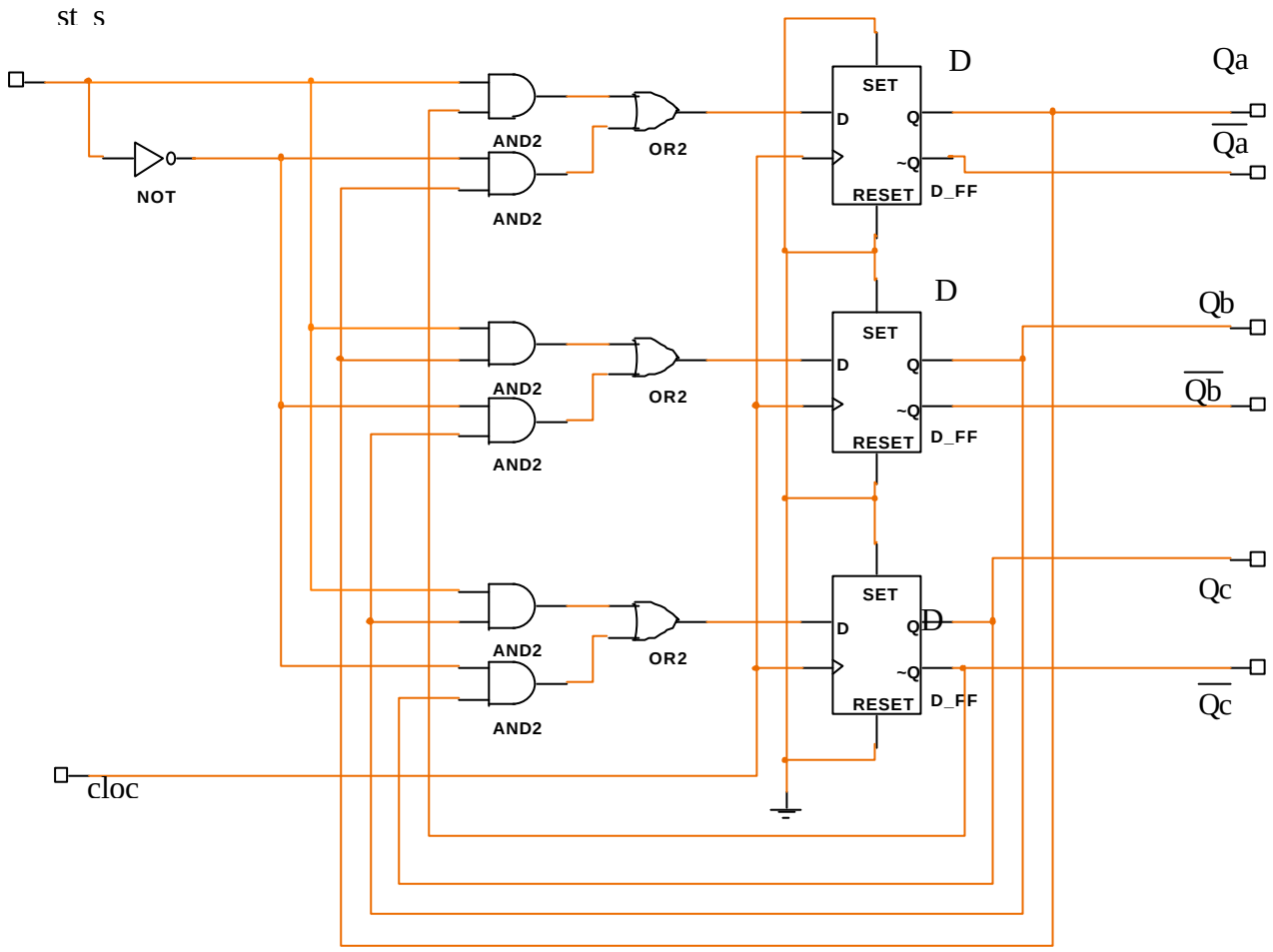


図 3.11　制御部 2の回路図

3.5.3　LED 駆動回路の回路

真理値表を表 3.3、カルノー図を図 3.12、回路図を図 3.13 に示す。
表３の入力は、制御部２のカウンタの出力である。表３の出力信号を見ると、a と f、b と e、c と d が同じであるのがわかる。よってこの３つの信号は LED を２つ直列に接続することになる。

表 3.3　LED 駆動回路

状態 （目の数）	入力			出力						
	Qc	Qb	Qa	a	b	c	d	e	f	g
1	0	0	0	0	0	0	0	0	0	1
2	0	0	1	0	0	1	1	0	0	0
3	0	1	1	0	0	1	1	0	0	1
4	1	1	1	1	0	1	1	0	1	0
5	1	1	0	1	0	1	1	0	1	1
6	1	0	0	1	1	1	1	1	1	0

出力 b,e について

Qc,Qb \ Q a	0	1
00		
01	×	
11		
10	1	×

$be = \overline{Qb} \cdot Qc$

出力 c,d について

Qc,Qb \ Q a	0	1
00		1
01	×	1
11	1	1
10	1	×

$cd = Qc + Qa$

出力 g について

Qc,Qb \ Q a	0	1
00	1	
01	×	1
11	1	
10		×

$g = \overline{Qc} \cdot \overline{Qc} \cdot + \overline{Qc} \cdot Qb + Qb \cdot \overline{Qa}$

図 3.12　表 3.3 のカルノー図

出力 a,f は、入力 Qc と同じであるため、af = Qc となる。

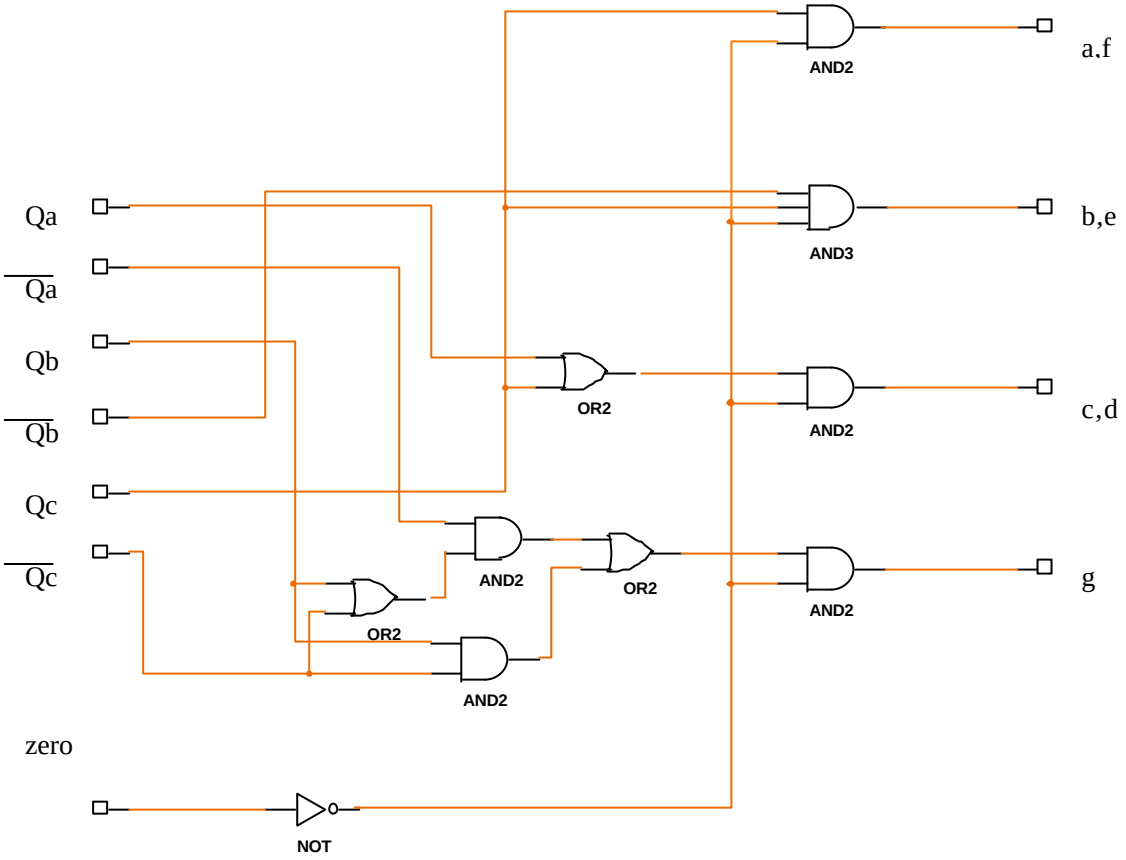


図 3.13　LED 駆動回路図

3.6　入力回路

外部入力は、start、stop、reset の３種類であるが、入力にスイッチを用いるとなると、入力時に発生するノイズ（チャタリング）によって誤動作を引き起こす可能性がある。通常では、入力時に SR フリップフロップを使うか、D フリップフロップを２、３段連ねたりするが、今回製作した回路では周波数が遅いので、チャタリングによる誤動作の可能性は高くないと考え、D フリップフロップを１段用いた。また、入力端子はオープンにしないよう気をつけた。
また回路図を図 3.14 に示す。

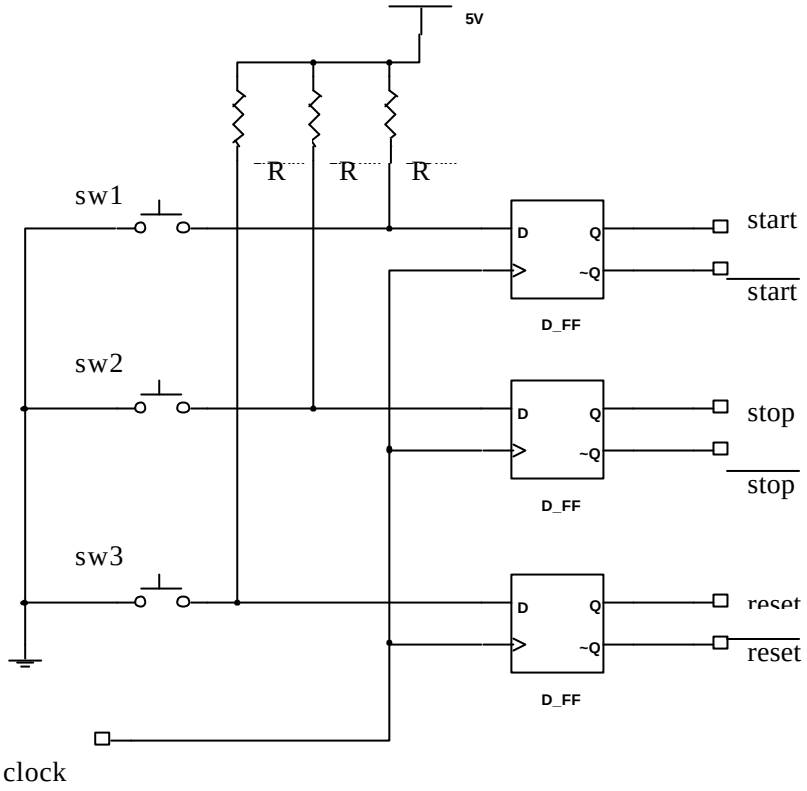


図 3.14　入力回路図

3.7　発振回路

フリップフロップを動作させるためのクロックが必要である。

CMOS **インバータを用いた** 20～25Hz **で発振する** CR **発振回路を作る。この発振周波数は、サイコロ表示の** LED **の点滅は人の目で確認できるが、表示される数字はわからない程度の速さである**

発振回路を図 3.15 **に示す。コンデンサ（**C**）と抵抗（**R1**）の** CR **時定数で発振周波数を基本的に決め、**R2 **は** inv3 **に電流が流れすぎないようにするための保護抵抗である。点** A**、**B **の波形を図** 3.16**、**3.17 **に示す。**

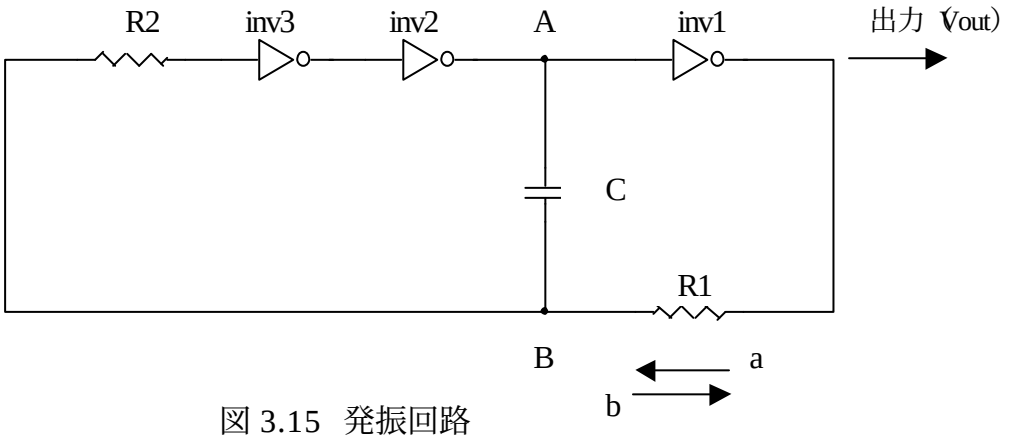


図 3.15　発振回路

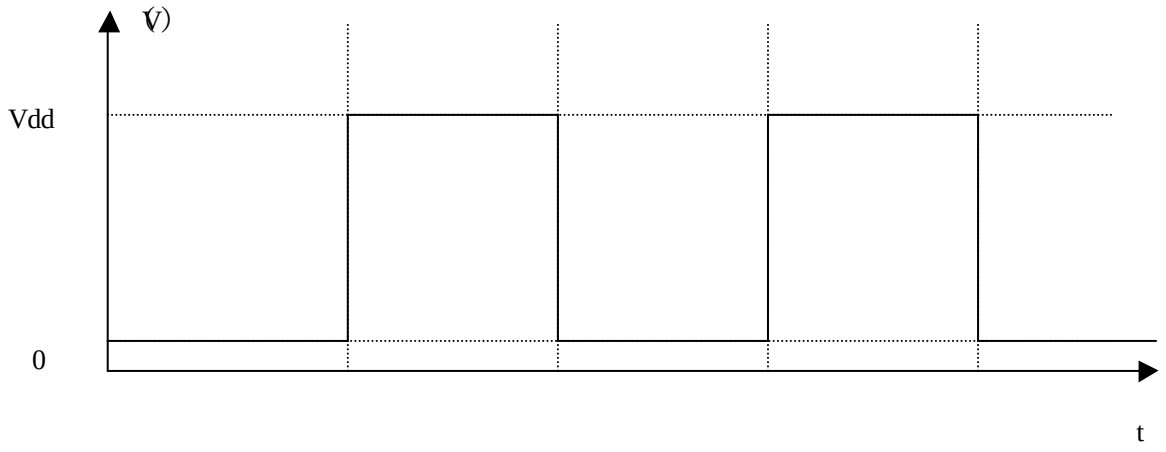


図3.16　点Aの波形

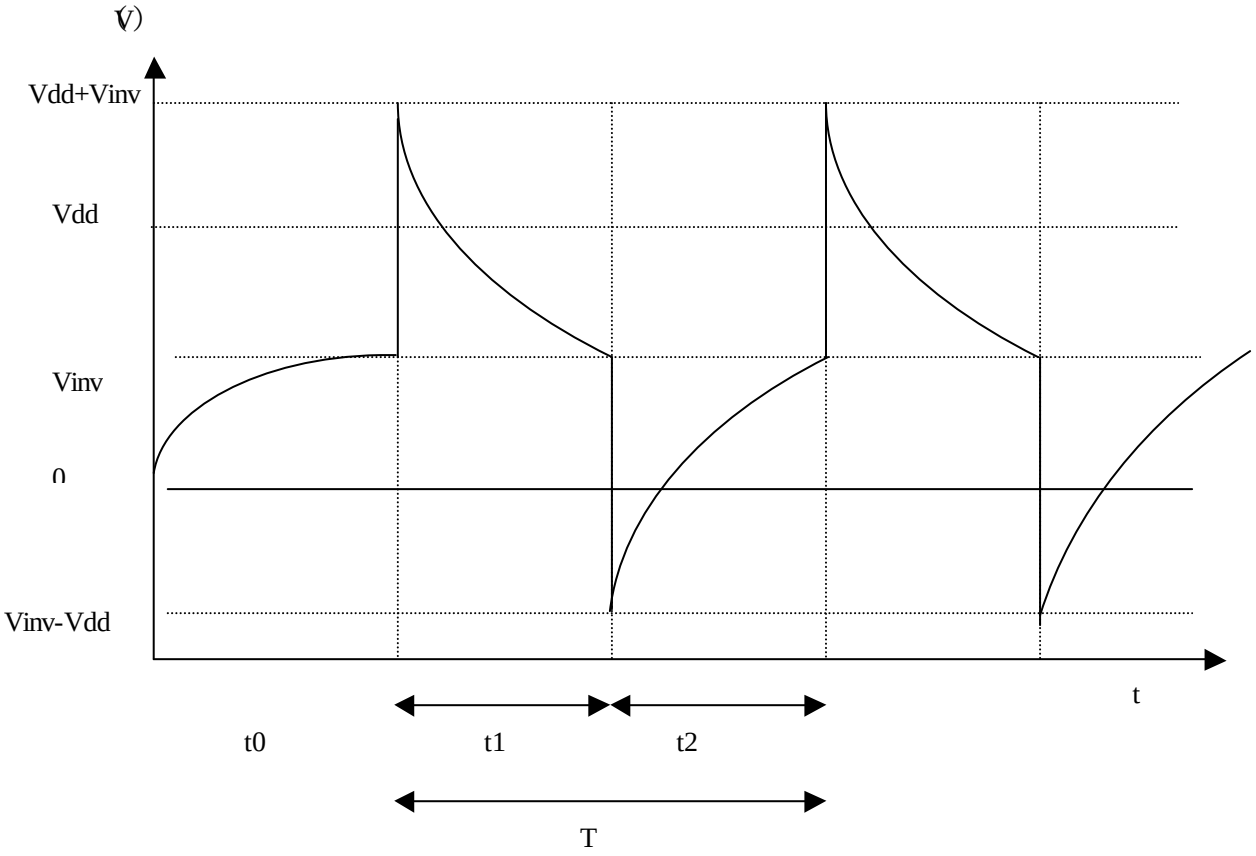


図3.17　点Bの波形

今、コンデンサ C に電荷が溜まってなく、点 A の電位が 0V で、inv1 の出力は 5V であるとする。R1 を流れる電流は a の向きであり、C の充電される。すると点 B の電位は上がり、論理反転電圧 Vinv を超えた瞬間に inv3 は反転し、次いで inv2 も反転して、点 A の電位は 5V に、inv1 の出力は 0V になる。この時 C の両端にかかる電圧を維持しようとするため、点 B の電位は Vinv+Vdd と

なる（図 3.17 の区間 t0）。今度は R1 に流れる電流は b の向きである。時間とともに点 B の電位は下がり、Vinv を超えた瞬間に、inv3 は反転し、点 A の電位は 0V となり、C は両端にかかる電圧は維持されるため、点 B は Vinv-Vdd となる（図 3.17 の区間 t1）。点 A の電位が 0V のため inv3 の出力は 5V であるので、R1 を流れる電流は a の方向である。そして点 B の電位は Vinv まで上がっていく（図 3.17 の区間 t2）。後は t1 と t2 の振動を繰り返すことによって発振する。

inv3 の PMOS、NMOS のインピーダンスを無視して t1、t2 を求める。

t1 について

$$Vinv = (Vdd + Vinv)e^{-t1/CR}$$

$$\ln(\frac{Vinv}{Vdd + Vinv}) = -\frac{t1}{CR}$$

$$t1 = — CR \ln(\frac{Vinv}{Vdd + Vinv})$$

t2 について

$$Vdd + (Vinv - 2Vdd)\, e^{-t2/CR} = Vinv$$

$$\frac{Vinv}{2Vdd - Vinv} = 1 - e^{-t2/CR}$$

$$-\frac{t2}{CR} = \ln(\frac{Vdd - Vinv}{2Vdd - Vinv})$$

$$t2 = - CR\ln(\frac{Vdd - Vinv}{2Vdd - Vinv})$$

T = t1 + t2 より、

$$T = - CR\ln(\frac{Vdd - Vinv}{2Vdd - Vinv}) - CR \ln(\frac{Vinv}{Vdd + Vinv})$$

$$T = - CR \ln[Vinv \frac{(Vdd - Vinv)}{(2Vdd - Vinv)(Vdd + Vinv)}]$$

ここで Vdd = 5V、Vinv = 1/2Vdd とすると、

$$T \quad 2.2CR$$

となる。

周波数 f = 20Hz とすると、T = 0.05s　であるから、
R = 22 kΩ、C = 1.0 μF とした。

3.8　動作テスト

multiSim でシミュレーションして動作確認後、プリント基板上に回路を作って、オシロスコープで波形を確認した。
図 3.19 に全体の回路を示す。

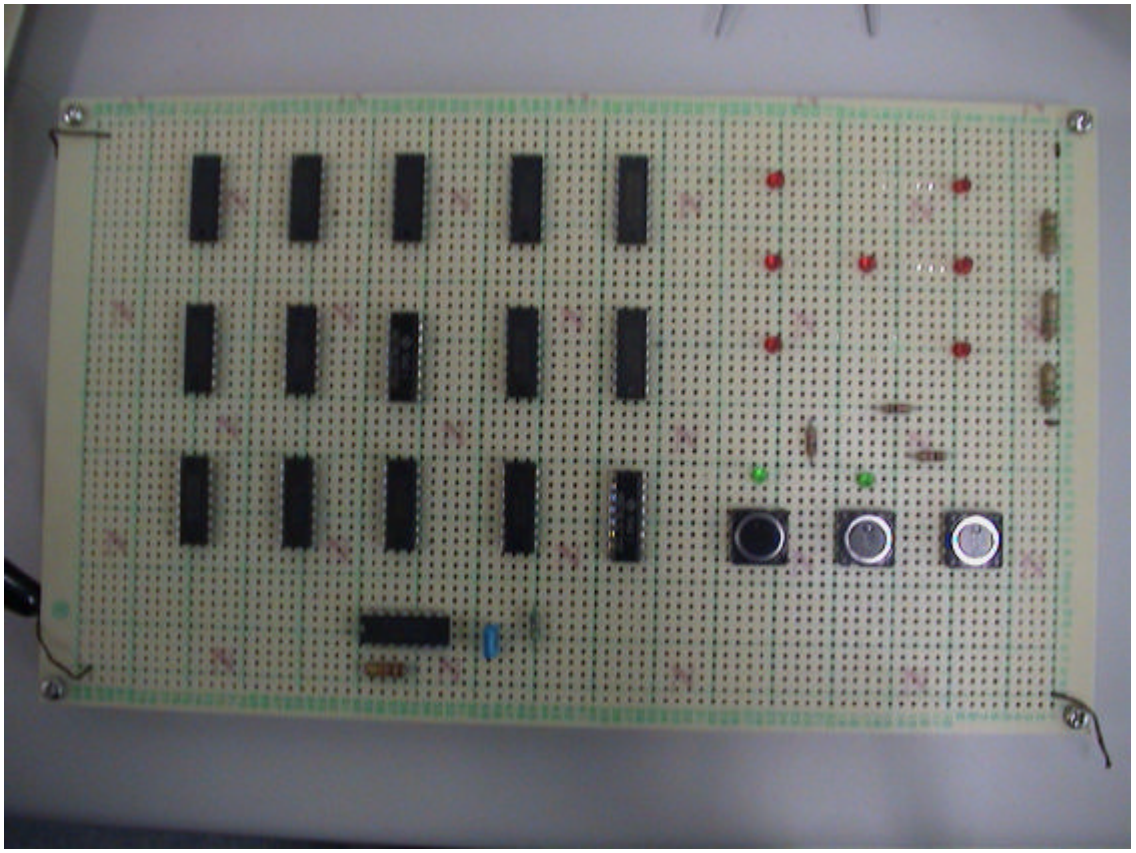


図3.19　サイコロ・マシン

観測は制御部 1、制御部 2、LED 駆動回路、発振器の波形について行った。制御部 1 は start、stop、reset の 3 つの入力があって状態遷移し、はじめて出力電圧の変化がある。また、その変化する数は多くない。そのため波形を確認するのではなく、入力による電圧の状態を確認した。図 3.20 に制御部 2 の観測波形を示す。表 3.2 の真理値表と見比べると、Qa、Qb、Qc が真理値表と同じ動作をしているのがわかる。

図 3.21 に LED 駆動回路の観測波形を示す。表 3.3 の真理値表と同様の動作をすることを確認した。

Qa

Qb

Qc

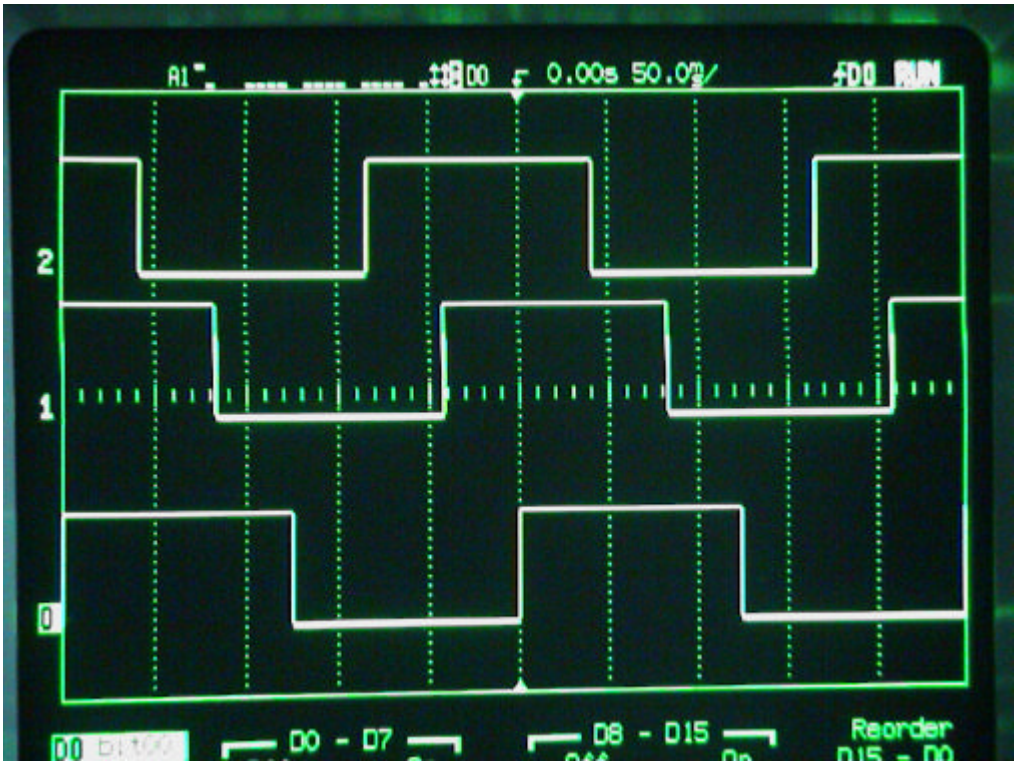


図3.20　制御部2の波形

cd

g

af

be

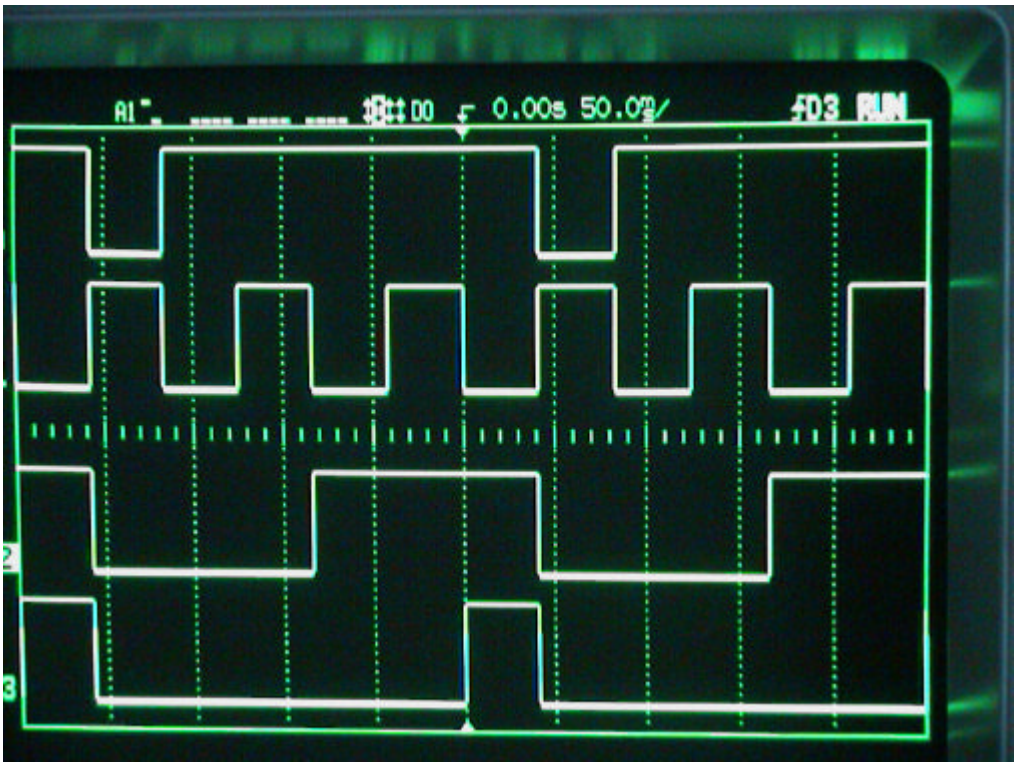


図3.21　LED駆動回路の波形

図 3.22 **は図** 3.15 **の発振回路における点** A **の波形で、図** 3.23 **は点** B **の波形である。**

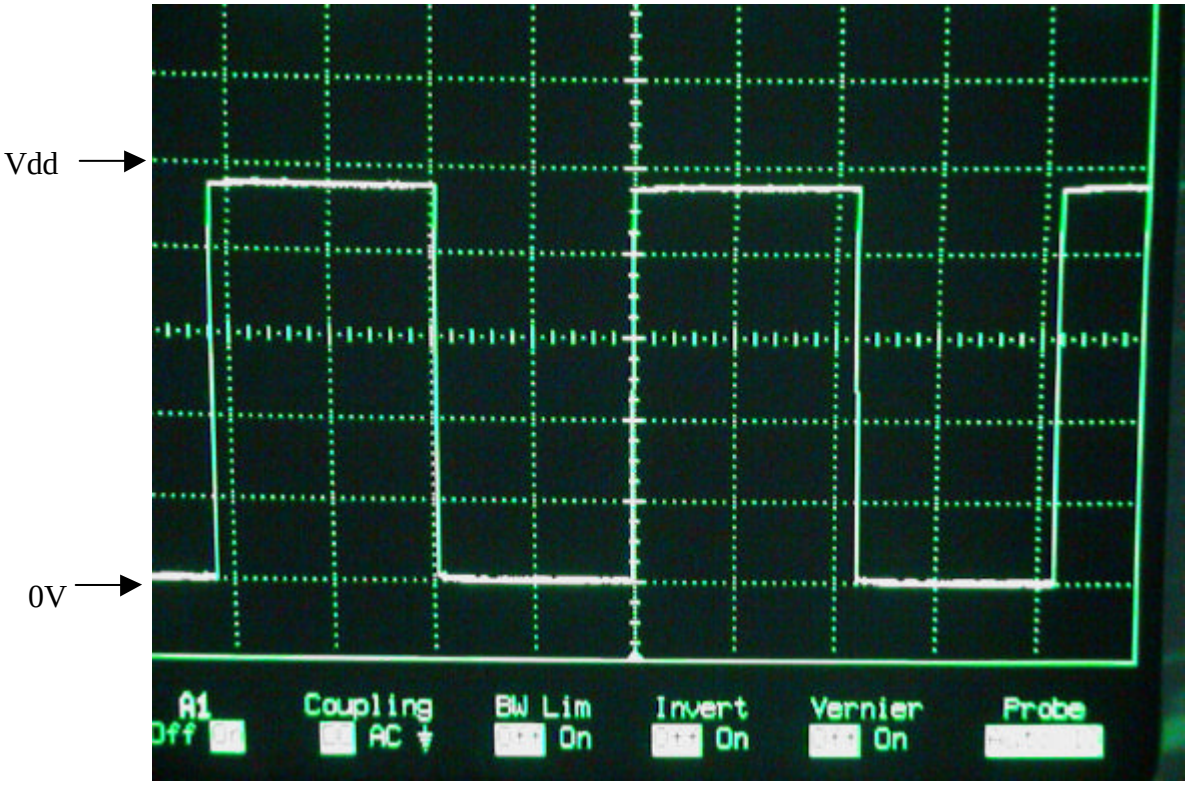


図3.22　発振回路の波形1

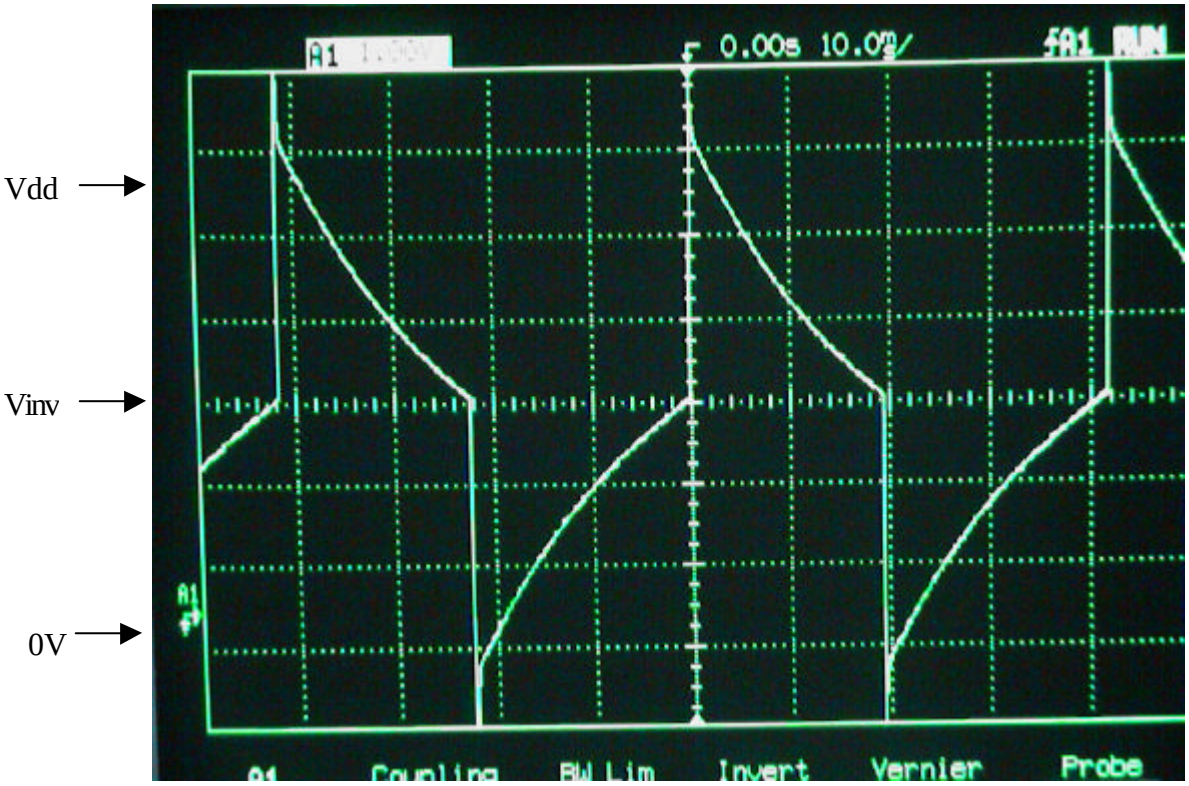


図3.23　発振回路の波形2

コンデンサ C の値を 1.0μF、抵抗 R1 を 22kΩとした時の波形である。計算値は周波数 f = 20.8Hz、論理反転電圧が 1/2Vdd とすると、デューティ比は 50%であったが､実測値では 24.5Hz から 25.0Hz､デューティ比が約 42%であった。

3.9　考察

・発振回路について

周波数とデューティ比が、予測値と実測値では異なっている。また実測した周波数はあまり精度が良くなかった。デューティ比が 50%にならなかったことについては、inv3 の PMOS、NMOS のインピーダンスの差によると考えられる。t1 が t2 より小さいことは、NMOS のインピーダンスが PMOS それより小さいことによるとも考えられ、通常の CMOS の構成からは理解できることである。周波数について実測値の方が高くなったことについてはまだ理解できていないが、実際の CMOS 回路の入力には保護素子が入っており、入力端が負の電圧 Vinv-Vdd（　2.5V）まで振り込まれるのは抑制されている。このため発振の測定値が小さくなり、発振周波数が予測値よりも高くなったとも考えられる。

・回路全体について

この回路は、当初から標準ロジック IC を用いて回路を作ることを前提としていた。回路をステートマシンで構成したが、IC のロジックとしての特性を生かして、組み合せて回路を作れば、より簡単な回路で、かつ早くできていたと思う。実際、ステートマシンを設計するのは注意していてもミスが多く発生し、muitiSim によるシミュレーションの段階で何度もやり直した。さらに IC をはんだ付けする際にも、配線が複雑になりすぎ、回路の動作がおかしい時にどこが悪いのか設計者すらわからなくなってしまう。回路を物理的にどう実現するかで、設計上回路の構成を考えるべきである。

今回はプリント基板に回路をすべて実装して終えてからオシロスコープで動作確認したのではなく、制御部 1 を作って動作確認、制御部 2 を作って動作確認・・というように各ブロックで進めたために、全体の動作がおかしくなってどうしようもなくなったという事態は避けられた。制御を分ける方法は良かったと思う。

4. 8bitCPU アーキテクチャの設計

4.1 設計フローの概要

今回設計した CPU は、データバス幅が 8bit、アドレスバス幅 8bit の CPU である。まず設計の大まかな流れを図 4.1 のフローチャートに示す。

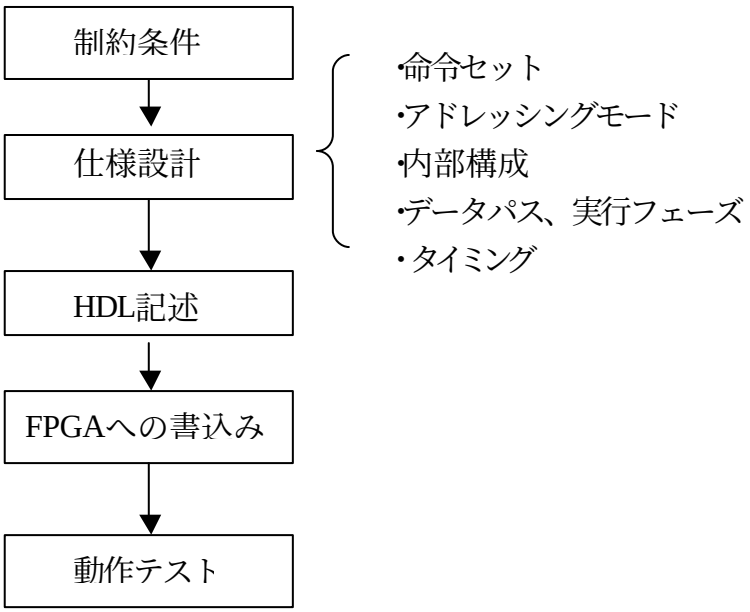


図 4.1 設計フロー

設計する上での条件、すなわち設計制約は何であるか、そしてそれをもとに仕様を考えていく。仕様では、CPU を動かすための命令はどのようなものがあるか（命令セット）、メモリにアクセスする時のアドレスを決める方法（アドレッシングモード）はどうするか、最上位ブロック下の機能ブロックをどうするか（内部構成）、そして CPU 内でのデータの流れ（データパス）を決め、命令の実行で、内部でどういった処理を行うか（実行フェーズ）、またその時のレジスタや制御線はどう動作するか（タイミング）を決定する。

仕様が決まれば、次に VHDL というハードウェア記述言語を用いて RTL レベルでの設計を行う。これを用いたのは、仕様通りに動作することを容易に確認するためであるのと、FPGA に書き込むためである。

4.2 仕様設計

アーキテクチャの仕様を決定していくにも、幾つかの物理的制約を考慮する必要がある。通常であれば、消費電力であるとか、動作速度や規模などを検討するのだが、今回は、とにかく矛盾なく完全に動作するものを作るというのが第一の条件とした。また第二に、設計をやり易くするため、メモリから CPU へ、あるいは CPU からメモリへのデータの転送は 1 クロックで行うことを前提する。第三はテストを容易にすることである。設計結果を FPGA に書き込んで、動作テストを行うには、単純なプログラムを走らせた後、内部が設計どおりに動作しているかどうか確認する必要がある。このためプログラム終了後、各レジスタの内容を見られるようにした。

4.2.1 命令セット

CPU を動かすのは命令である。命令が多ければプログラムは容易になるが、一方 CPU の設計が難しくなるため、命令の数はできるだけ抑え、必要な物のみとした。以下に用意した命令の種類（全 15 種）を示す。

・停止命令（HALT）
・データのロード、ストア（LD、ST）
・フラグのセット（SCF、RCF）
・データの移動（MOV）
・分岐命令（BR）
・ 演算命令（AND、OR、XOR、COMP、ADD、ADC、SUB、SUC）

停止命令は CPU の動作を停止させるもので、停止させた後は各レジスタの内容は現在の状態を保持し続ける。ここで外部からテスト入力を与えることによってレジスタの内容を確認する。したがってこの命令は、通常はプログラムの最後に持ってくることになる。データのロード、ストア命令は、メモリからデータを読み込む、あるいは書き込む命令である。フラグは、演算結果に応じて変化する信号で、演算結果が正か負か、あるいはゼロかなどの判断の結果や、

桁上げ、桁あふれなどの結果を示す。ここでは、フラグとしては、ネガティブフラグ（NF)、ゼロフラグ（ZF)、キャリーフラグ（CF)、オーバーフローフラグ（VF）を採用した。このフラグセット命令は、この中の一つであるキャリーフラグに関する命令である。おもに桁上げや借りのある加減算を行う場合、キャリーフラグを“1”にセットする命令（SCF）と、キャリーフラグをリセット（“0”にセット）する命令（RCF）の２種類の命令がある。

データ移動命令（MOV）は、あるレジスタのデータを、別のレジスタへコピーする命令である。分岐命令（BR）は、幾つかあるフラグの“0”か“1”を判断してプログラムをジャンプさせる命令である。例えば図 4.2 のように、メモリの 16 番地に分岐命令があった時、キャリーフラグが“0”であれば 10 番地へとジャンプし、“1”であれば次の 17 番地のデータを読み込む。

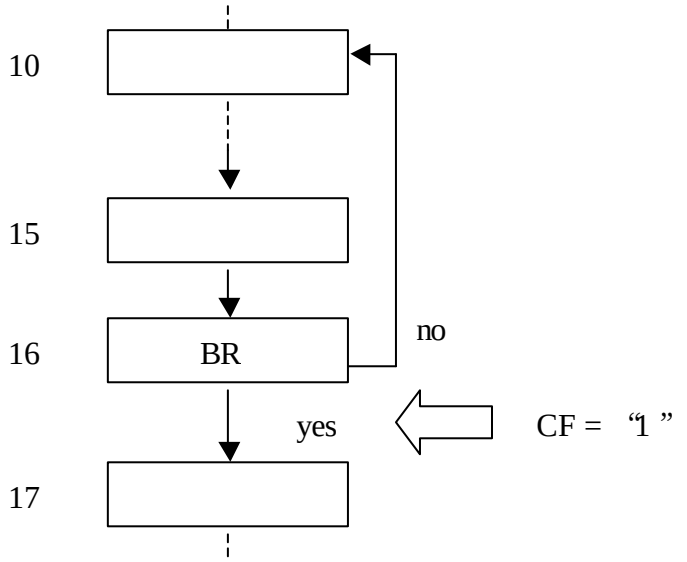


図4.2　分岐命令

最後に演算命令であるが、演算は論理演算（AND、OR、XOR、COMP）と、算術演算（ADD、ADC、SUB、SUC）の二つに分けられる。AND、OR、XOR は、ビットごとの論理積、論理和、排他的論理和である。COMP は比較で、データ A、データ B の減算を行う。ADD、ADC は加算と桁上げのある加算、SUB、SUC は減算と借りのある減算である。算術演算と COMP はデータ幅の 8bit と 1bit 付加した 9bit で演算を行う。付加した 1bit は演算結果の状態を示す。このビットの“0”と“1”でフラグの状態が変わる。例えば図 4.3 の様に、データ A、B の ADD の場合であれば、データ A とデータ B の 8bit 目の加算で桁上げが発生したとき、9bit 目が“1”となり、桁あふれ（オーバーフロー）が起こる。

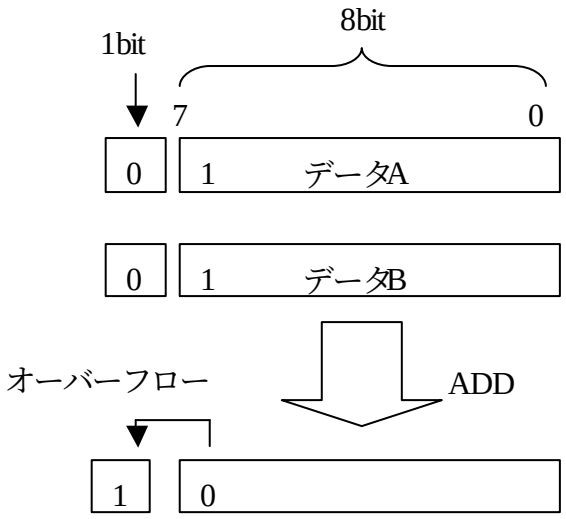


図4.3　桁あふれ（オーバーフロー）

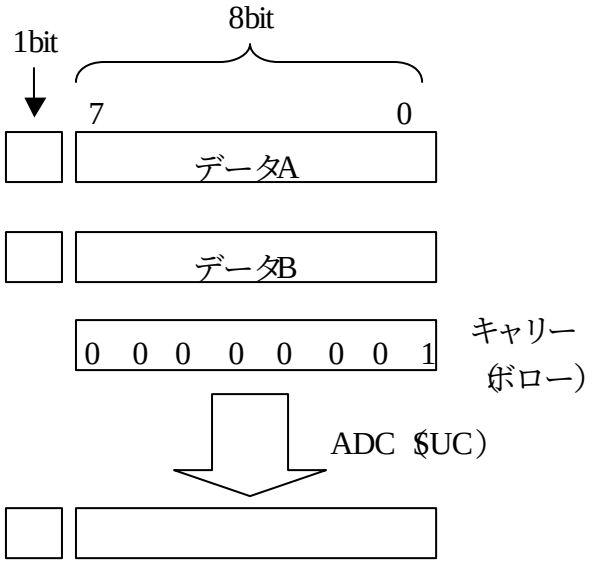


図4.4 キャリー付き加減算

ADC と SUC は、図 4.4 に示すように、もしキャリーフラグが“1”の場合、データ A、B と、“1”の演算となる。キャリーフラグが“0”の場合はデータ A、B と“0”の演算となる。また、ADC と SUC は、8bit 目の演算で桁上げ（借り）が起これば、オーバーフローではなくキャリー（ボロー）の発生となる。

COMP は、基本的には SUB と同じ減算であるが、図 4.5 に示すように、大小の比較であるので、結果としては A > B、A = B、A < B　の 3 通りがあり、それぞれの場合に応じてフラグを変化させる。A > B のときは演算結果が正であ

るので、ネガティブフラグを“ 0 ”とし、A = B のとき、すなわち演算結果が“ 0 ”であるときにはゼロフラグを“ 1 ”し、A < B のときは、結果が負になるので、ネガティブフラグを“ 1 ”とする。SUB の場合は、A < B のとき、ボローではなくオーバーフラグが“ 1 ”になること以外は COMP と同様である。

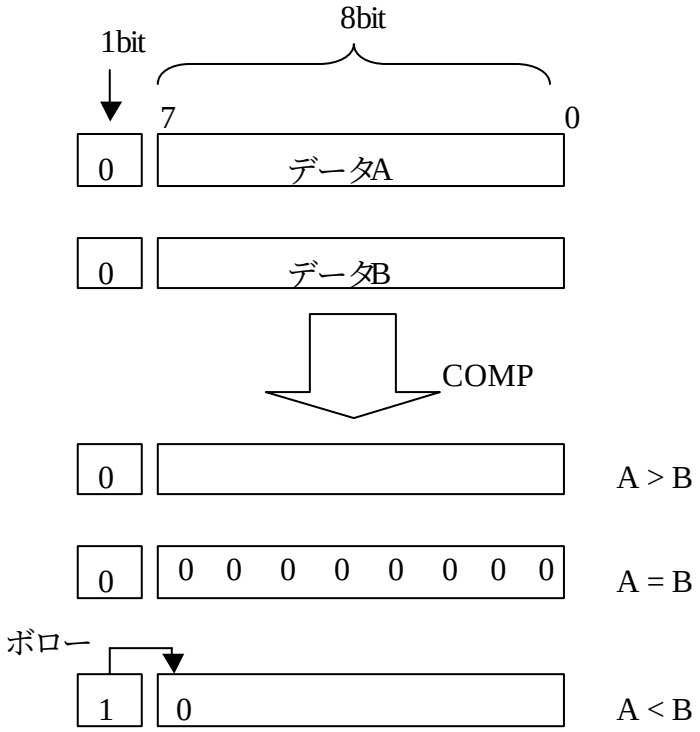


図4.5 比較

4.2.2 アドレッシングモード

命令が決まれば、その命令をメモリからどう読み込んでくるか、またメモリにあるデータをどう読み込むかを決める。命令あるいはデータを読み込むためにはメモリのアドレスを指定しなくてはならない。その方法を以下の 4 つとした。

- 即値アドレス
- 絶対アドレス

・ インデックス修飾アドレス
・ レジスタ間接アドレス

(1) 即値アドレス

即値アドレスで、演算対象となる命令のアドレス部を示すオペランド（operand）が、そのまま演算対象のデータとなる。（図 4.6）。

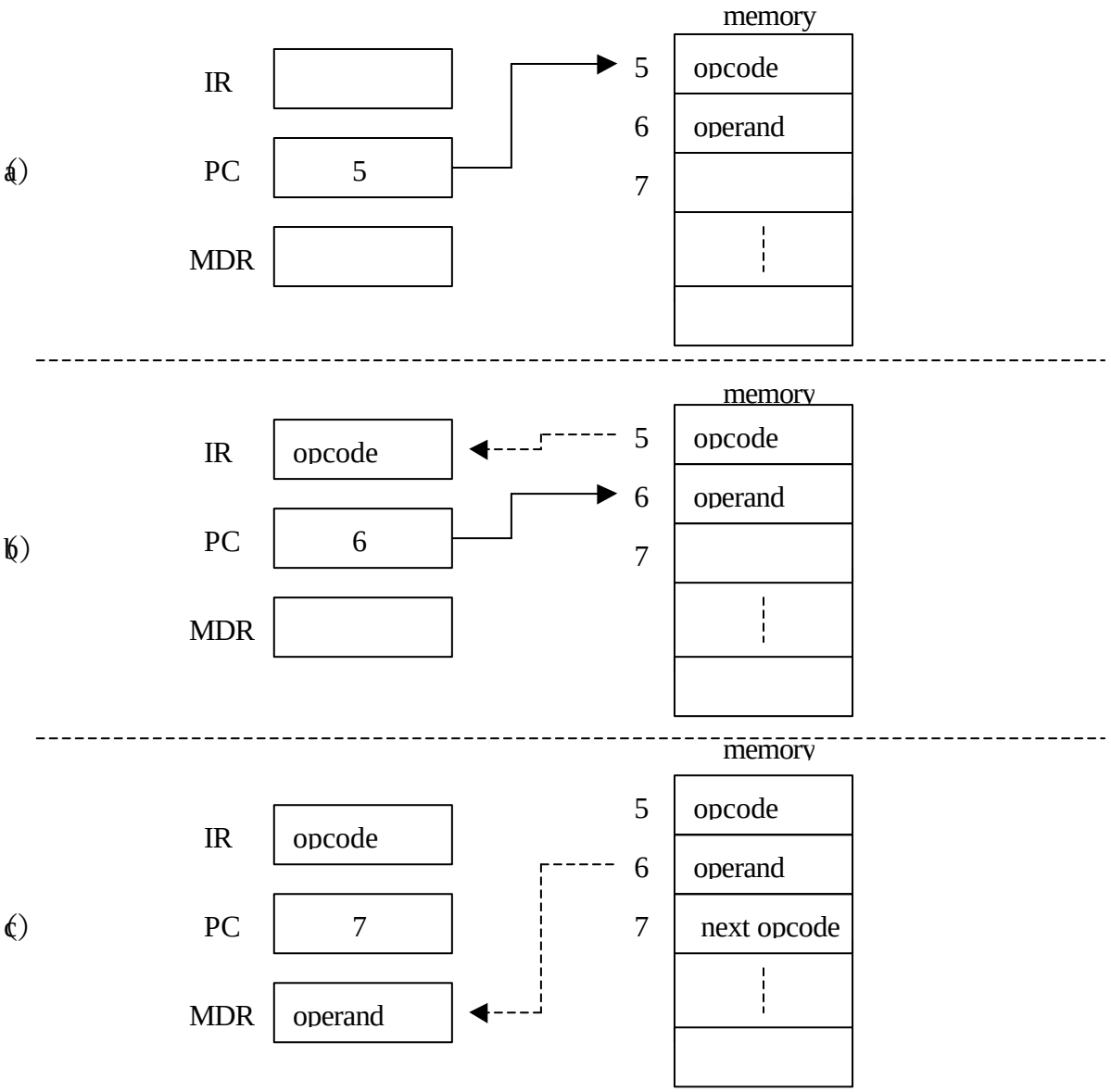


図 4.6　即値アドレス

図 4.6 の（a）から（c）は、即値アドレスで命令をメモリからレジスタへ格納する過程を示したものである。IR、PC、MDR はレジスタで、命令を保持するレジスタであるインストラクション・レジスタ（IR)、現在行っているプログラムの中の命令の番地を保持するプログラムカウンタ（PC)、メモリからのデータを保持するメモリデータ・レジスタ（MDR）である。

今、メモリの 5 番地に命令に 1 語目（オペコード、アドレッシングモードを含む）があり、6 番地に 2 語目（オペランド）があるとする。そして、PC はメモリの 5 番地を指している（a)。1 語目には命令やアドレッシングモードの指定がされており、クロックの立ち上がりで PC は 1 増加し、1 語目が IR に取り込まれ（b)、PC は 1 つ増加する。そして命令が何であるか、アドレッシングモードは何であるかを判断する。PC は 6 番地を指しているので、次のクロックでオペランドを MDR に取り込み、PC は次の命令を読むために 1 増加する（c)。MDR に取り込まれたオペランドは演算用データでとして使用される。

（2） 絶対アドレス

絶対アドレスはオペランドがメモリのアドレスとなるものである。絶対アドレスの手順を図 4.7 に示す。

図 4.7 のレジスタは、IR、PC、MDR とさらにメモリアドレスを保持するためのメモリアドレス・レジスタ（MAR）が加わっている。（a）はメモリの 10 番地から 1 語目を IR に読み込み、オペコードとアドレッシングモード、レジスタアドレスを判断し、PC を 1 増加させた状態である。ここでクロックが入ると、PC が指す 11 番地のオペランドを MAR に取り込み、PC を 1 増加する（b)。オペランドはデータが格納されているアドレスであるので、MAR に取り込まれたオペランドは、メモリのアドレスとなる（c)。仮にオペランドがメモリの 20 番地であった場合、 ロード命令や演算命令であればメモリの 20 番地からデータを読み込む動作となり、ストア命令であれば、レジスタアドレスで指定されたレジスタの内容が、20 番地に格納される。

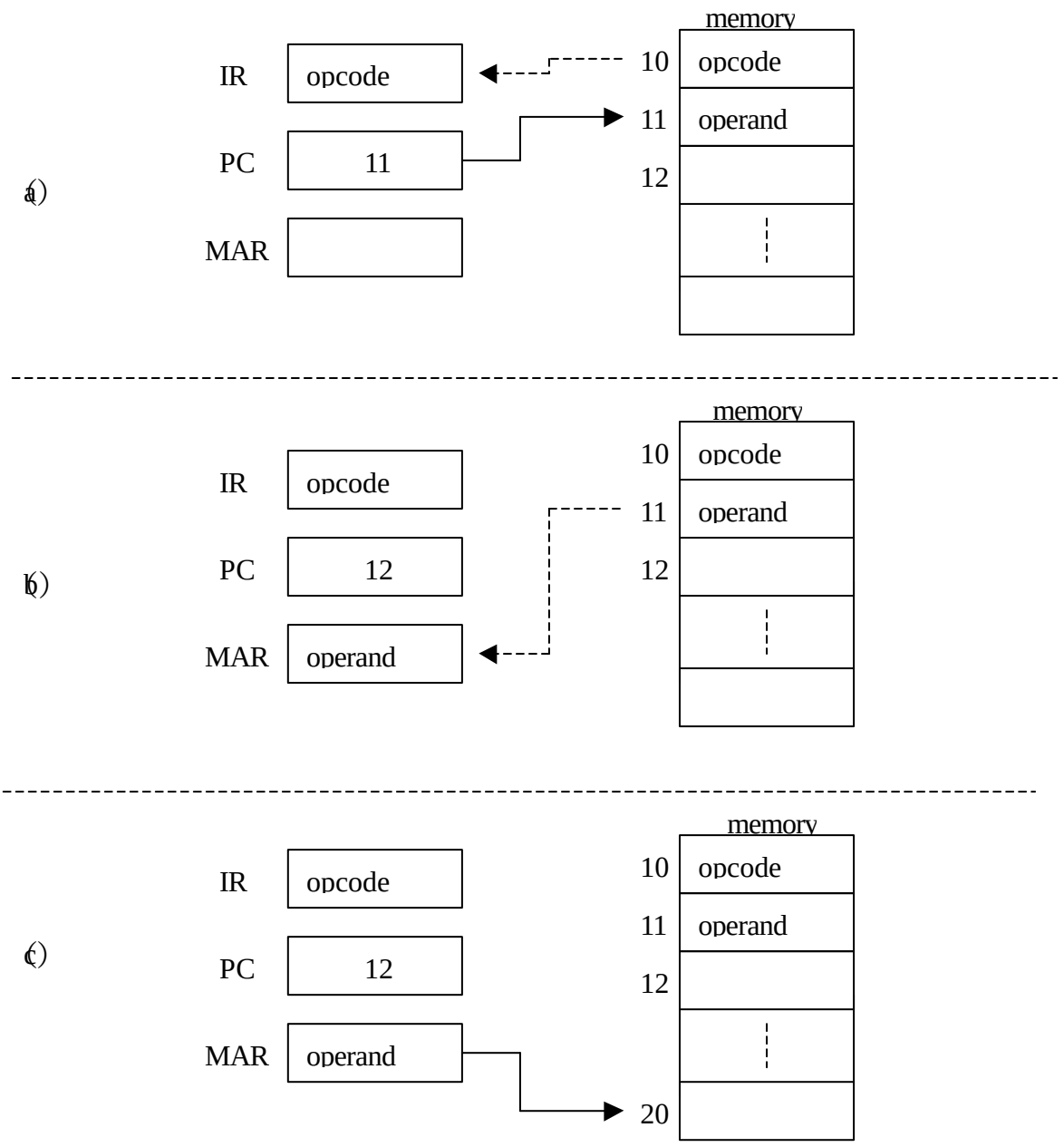


図 4.7　絶対アドレス

(3) インデックス修飾アドレス

インデックス修飾アドレスは、ディスプレースメント（変位量：displacement）と基底アドレスの加算でメモリアドレスを指定するものである。基底アドレスを保持するレジスタをインデックスレジスタ（IX）とする。図 4.8 にインデックス修飾アドレスの手順を示す。(a) は 25 番地から 1 語目

を IR に読み込み、PC を 1 増加させた状態である。クロックの入力で PC の増加とディスプレースメントを MDR に取り込み（b)、IX の内容と MDR の内容を加算し、メモリアドレスを指定する（c)。仮に IX に 50 番地を示す内容が保持されており、これが基底アドレスとなる。ディスプレースメントが 7 であるとすると、指定するアドレスは 57 番地となる。

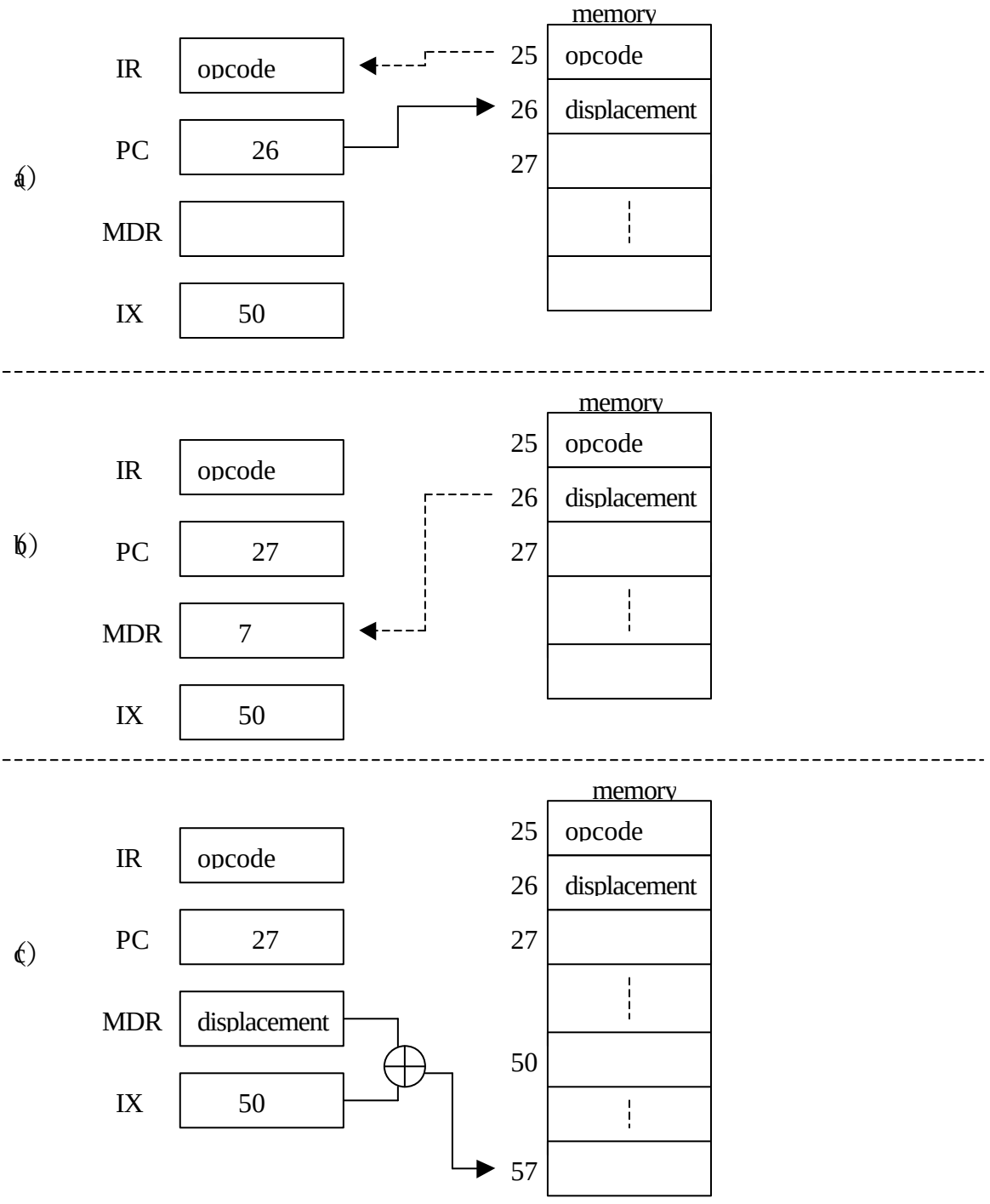


図 4.8　インデックス修飾アドレス

（4） レジスタ間接アドレス

レジスタ間接アドレスは、MAR **の内容をそのままアドレスとするものである。**

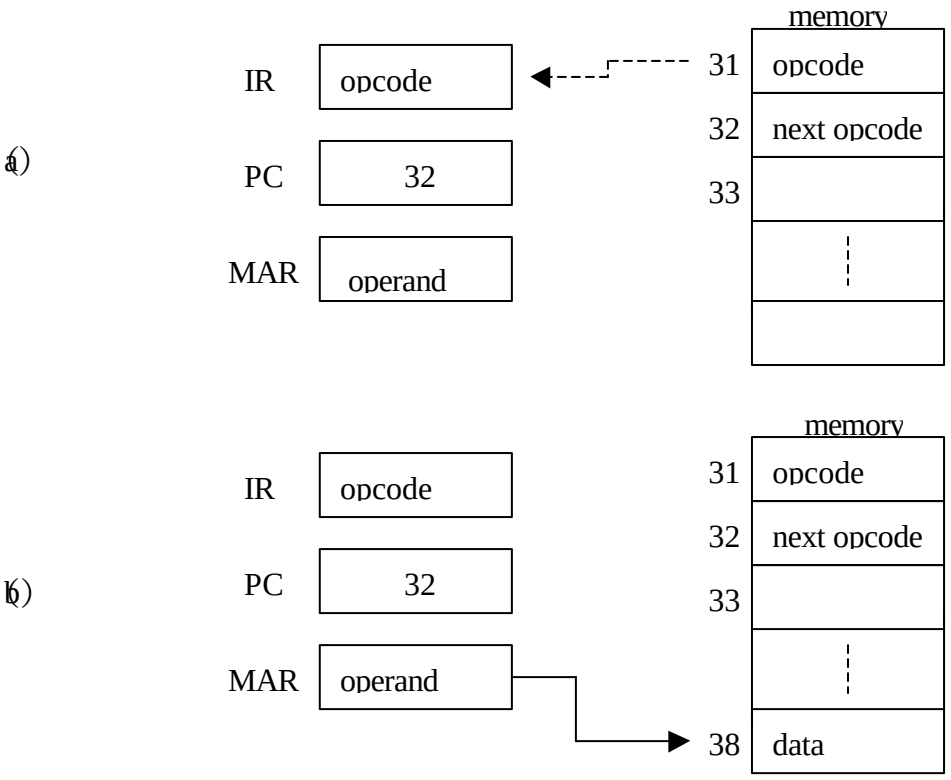


図 4.9　レジスタ間接アドレス

図 4.9 **にレジスタ間接アドレスの手順を示す。**（a）**は** 31 **番地から** 1 **語目を** IR **に読み込み、**PC **を** 1 **増加させた状態である。クロックの入力があると、**1 **語目のアドレッシングモードを判断して、**MAR **の内容をメモリアドレスとし、さらにクロックの入力で、そのアドレスにあるデータを読み込むかあるいはそのアドレスにデータを書き込む**（b）**。図では** MAR **にあらかじめオペランドとして** 38 **番地を示す内容が保持されており、レジスタ間接アドレスであることを判断した時点で** MAR **の内容をアドレスバスに載せる。メモリからオペランドを読み込む必要が無い。**

4.2.3 データフォーマット

命令、アドレッシングモードごとでデータフォーマットを考える。データフォーマットは、1 語命令であれば 8bit、2 語命令であれば 16bit であるが、そのデータ幅の中でオペコードやアドレッシングモードを指定する区域あるいはビット幅を規定したものである。またその区域をフィールドと呼ぶ。フィールドについては 4.2.5 で述べる。

命令は全 15 種類で、オペコードとして 4bit 幅を固定する。

図 4.10 は停止命令、フラグセット命令のフォーマットである。

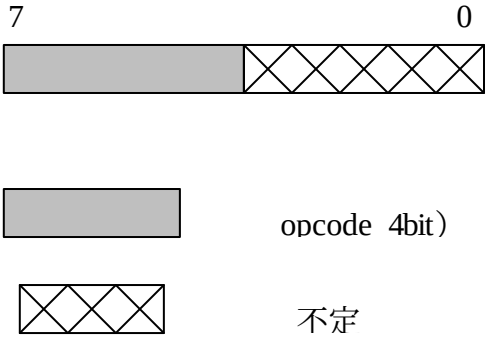


図4.10 停止、フラグセット命令のフォーマット

停止命令とフラグセット命令は 4bit のオペコードのみで、その他は必要でないので残り 4bit は不定である

図 4.11 は分岐命令のフォーマットである。オペコードの 4bit、分岐条件が 4bit、オペランドが 8bit である。分岐条件は 5 つで、フラグの状態で判断させる。無条件（always）と、CF =“1”(on carry)、NF =“1”(on negative)、ZF =“1”(on zero)、VF =“1”(on over flow）である。2 語目のオペランドは条件が成立しなければ読み込まれる。条件成立ではオペランドは読み込まれず、次の命令を読み込む。またオペランドを読み込むときのアドレッシングモードは即値アドレスのみとし、PC に保持される。

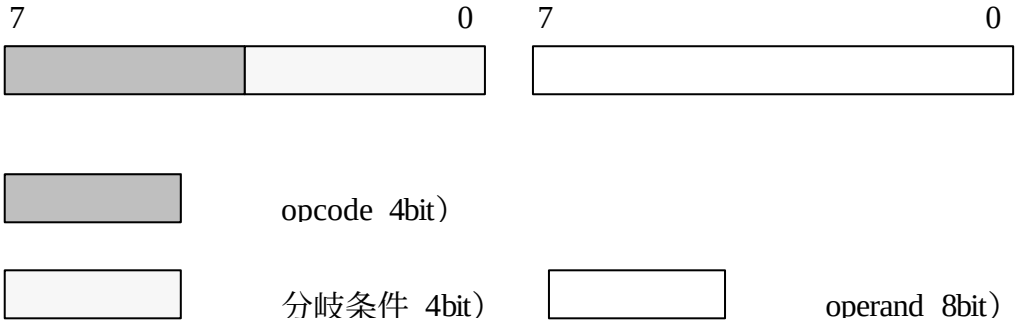


図4.11 分岐命令のフォーマット

図 4.12 は移動命令のフォーマットである。

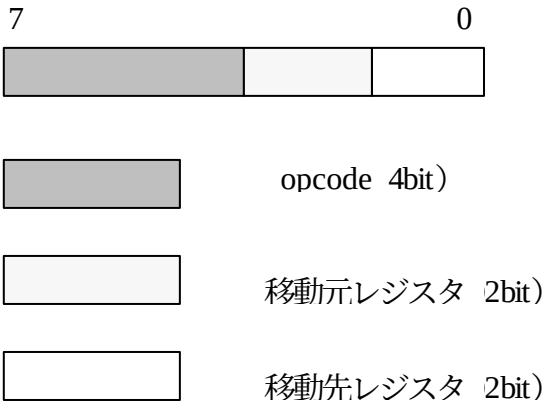


図4.12　移動命令のフォーマット

オペコードが 4bit**、移動元レジスタが** 2bit**、移動先レジスタ** 2bit **である。移動元レジスタとはコピーの対象となるデータが保持されているレジスタで、移動先レジスタは移動元レジスタのデータを保持するためのレジスタである。移動後は移動元レジスタの内容は消えない。**

図 4.13 **はロード命令、ストア命令、演算命令で使用するフォーマットである。**

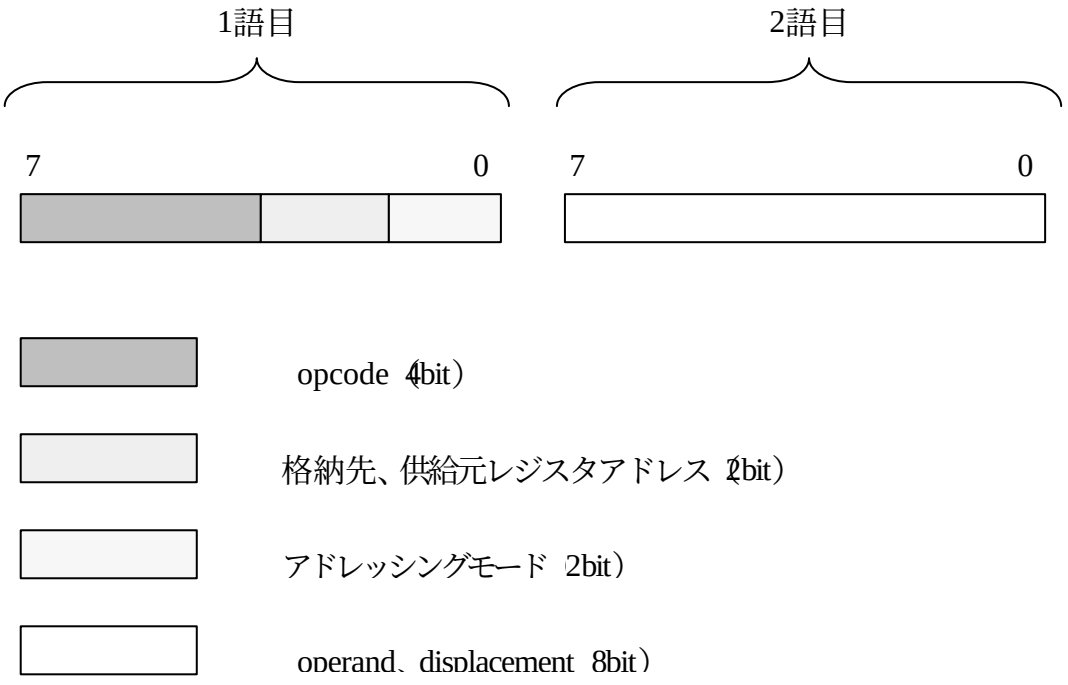


図4.13　ロード、ストア、演算命令のフォーマット

1 語目はオペコード、格納先あるいは供給元レジスタ、アドレッシングモードの指定、2 語目はオペランド、ディスプレースメントである。オペコードは 4bit、アドレッシングモードが 4 種であるので 2bit、あとの 2bit でレジスタを指定する。ロード命令の時は格納先レジスタで、ストア命令と演算命令の時は供給元レジスタの指定となる。演算の場合、メモリのデータと供給元のレジスタのデータで演算を行うことになるわけだが、その演算結果をどこに格納するかが 1 語目には指定されていない。よって演算の場合は結果を格納するレジスタを暗黙的に決め、格納用レジスタを用意することになる。

4.2.4　レジスタセット

内部構成やデータパス構築のためにレジスタセットをまとめる。

まず以下のレジスタが必要である。

- IR ---- 命令を保持する
- IX ---- 主にアドレス修飾に用いる
- MDR ----メモリからのデータを保持する
- PC ---- プログラムのアドレスを保持する
- MAR ----主にメモリアドレスを保持する

他に、演算で使用するための演算用レジスタが必要であるが、汎用性のある演算用レジスタをアキュムレータ 1（ACC1）とし、結果を格納するレジスタをアキュムレータ 2（ACC2）とする。あとフラグの状態を保持するフラグレジスタ（Flag）も必要である。他のレジスタは 8bit 幅であるが、Flag は CF、ZF、NF、VF の 4 種類であるため 4bit 幅とした。

- ACC1 ---- 汎用性のある演算用レジスタ
- ACC2 ---- 演算結果格納用レジスタ
- Flag ---- フラグの状態を保持する

4.2.5 フィールド

命令フィールド、アドレッシングモード・フィールド、レジスタフィールド、分岐条件フィールドがある。

（1） 命令フィールド

表 4.1 **に命令フィールドを示す。**

表4.1　命令フィールド

番号	命令
0000	HALT
0001	LD
0010	ST
0011	SCF
0100	RCF
0101	MOV
0110	BR
0111	AND
1000	OR
1001	XOR
1010	COMP
1011	ADD
1100	ADC
1101	SUB
1110	SUC

（2） **レジスタフィールド**

表 4.2 にレジスタフィールドを示す。表 4.2 のレジスタは、演算に使用できるレジスタで、他のレジスタ（IR、PC、MDR）は用途が限られているため、番号は割り振られていない。しかし表 4.4 のテスト用レジスタフィールドは、すべてのレジスタについて番号を割り振っている。テストは各レジスタの内容を見て動作を確認するためである。テスト入力は外部から与える。

表 4.2 レジスタフィールド

番号	レジスタ
00	ACC1
01	ACC2
10	IX
11	MAR

表4.3 テスト用レジスタフィールド

番号	レジスタ
000	ACC1
001	ACC2
010	IX
011	MAR
100	IR
101	PC
110	MDR

（3） アドレッシングモード・フィールド

表 4.4 にアドレッシングモード・フィールドを示す。

表 4.4　アドレッシングモード･フィールド

番号	アドレッシングモード
00	即値アドレス
01	絶対アドレス
10	インデックス修飾アドレス
11	レジスタ間接アドレス

（4） 分岐条件フィールド

表 4.5 に分岐条件フィールドを示す。

表 4.5　分岐条件フィールド

番号	条件
0000	Always
1000	on carry
0100	on zero
0010	on overflow
0001	on negative

4.2.6 内部構成

図 4.14 は、CPU の最上位ブロックであり、その入出力を示したものである。入力はクロック（ck）とリセット（reset)、テスト用（test)、スタート（start）の 4 つで、出力はメモリからデータを読み込むためのメモリリード信号（ memr ）と、メモリへデータを書き込むためのメモリライト信号（ memw ）と 8bit のアドレスバス（address bus）で、メモリリード信号とメモリライト信号は“ 0 ”の時有効である。8bit の外部データバス(external data bus)は、データをメモリから読み込む時とメモリへ書き込む時があるため双方向である。スタート入力は、プログラムを開始するためのもので、通常リセットを入力し内部を初期状態に戻したあと入力する。

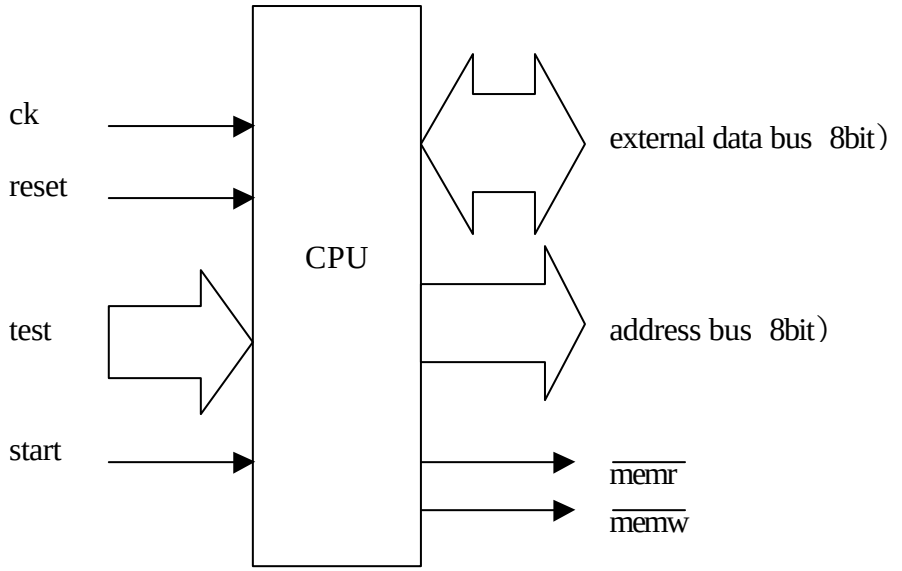


図4.14　最上位ブロック

図 4.14 のブロックを機能別に分割する。図 4.15 は図 4.17 を大きく 4 つのブロックに分けたものである。その 4 つとは、データを保持するレジスタ群（register)、演算部（ALU)、アドレス生成部（address)、それらを制御する制御部（control）である。

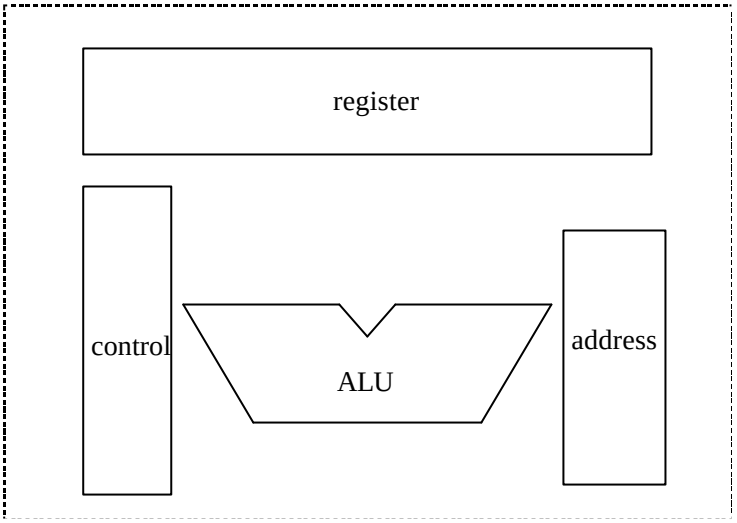


図4.15　ブロック図1

さらに図 4.16 **に示したように、レジスタ群は機能別に** 8 **つのブロックに分けられる。**

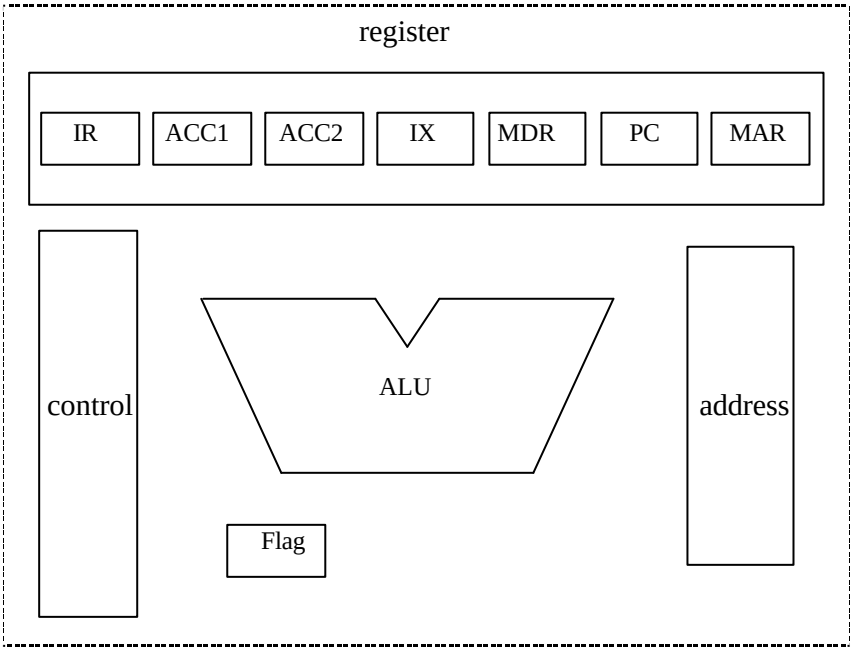


図4.16　ブロック図2

図 4.17 は大まかなデータパスを示したものである。レジスタの入力には、ALU の出力（ALU bus）と内部データバスの信号（internal databus）の 2 種類あり、レジスタの出力は、制御部・ALU・アドレス生成部の入力となる。内部データバスと ALU バスはデータバスバッファ（ databus buffer）を介して外部データバスと信号を送受信する。アドレスバス（address bus）は、アドレス生成部からの出力である。

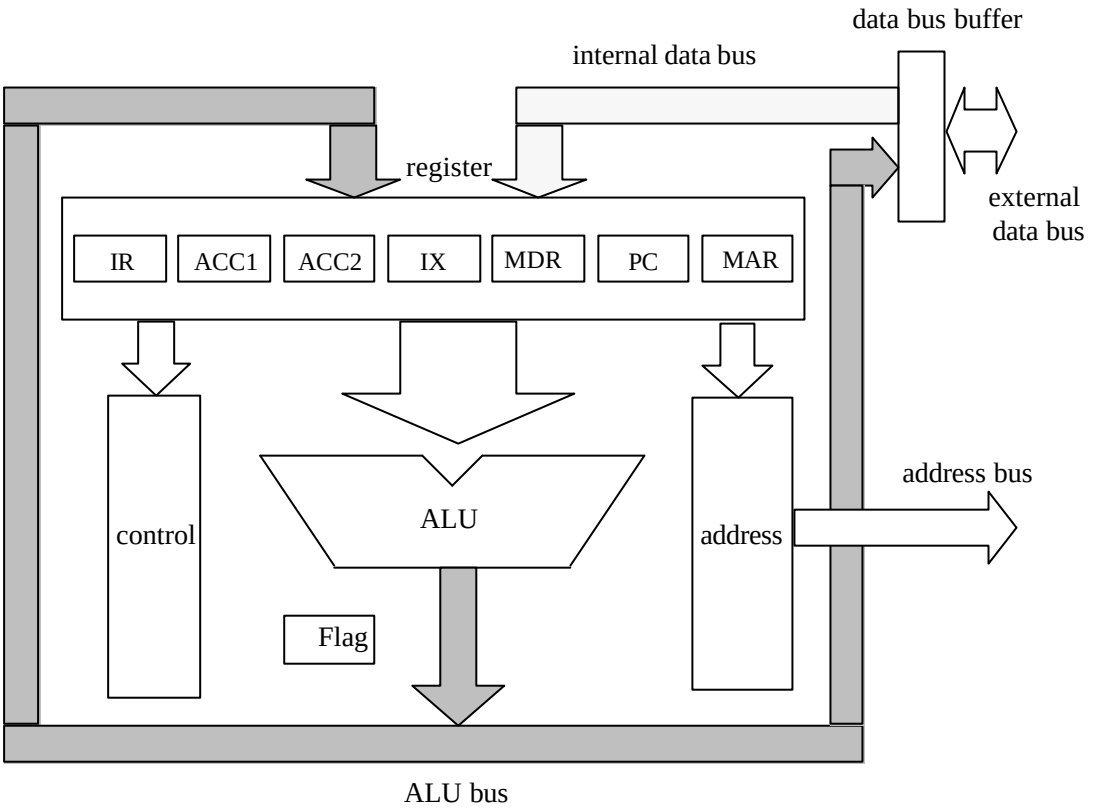


図4.17　ブロック図3

図 4.18 はテスト入力 1、2（test signal 1、2）と ALU マルチプレクサ（ALU multiplexer）を加えたものである。ALU マルチプレクサは 2 つの目的があり、1 つは ALU の入力を決めるためで、フラグレジスタを除く全レジスタの出力が入力され、ALU に入力する信号を選択する。もう 1 つはテスト出力を決めるためである。テスト用の入力 2（test signal 2）で選択された各レジスタの内容は、テスト出力（test output）として ALU マルチプレクサから外部データバスへ出力される。テスト入力 1（test signal 1）は、ALU の出力とテスト出力（test output）の選択回路（multiplexer）でテスト出力を選択し、データバスバッファで常にテスト出力を外部バスへ出力する。

以後テスト入力は省略する。

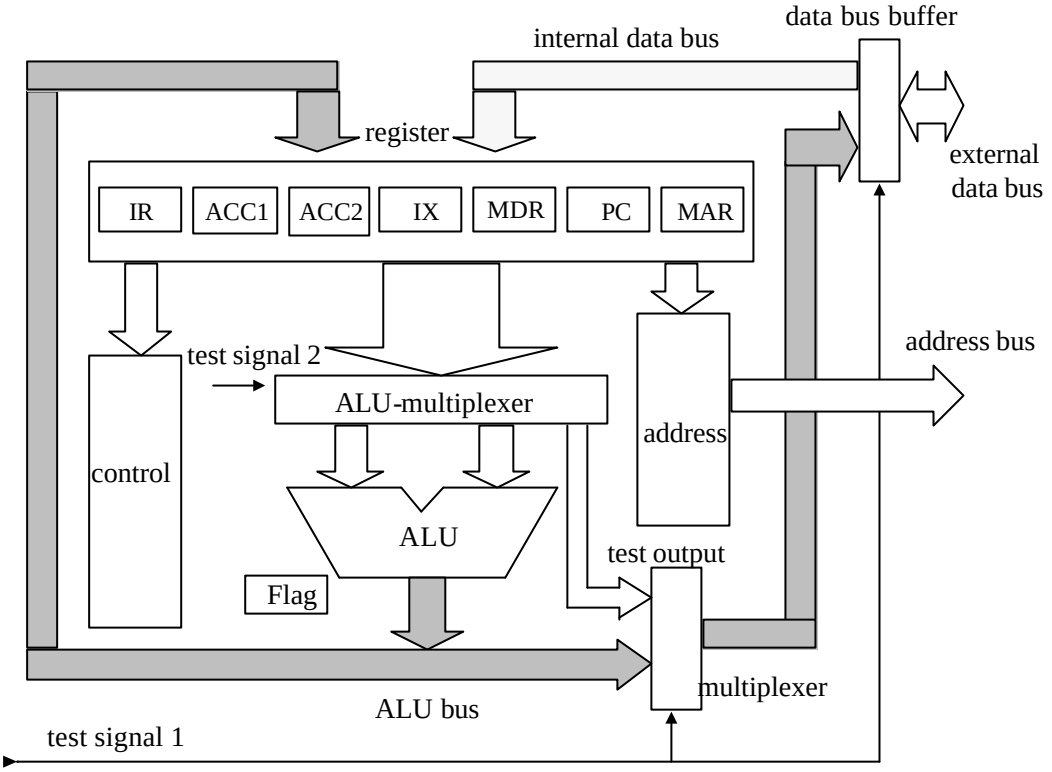


図4.18　ブロック図4

図 4.19 はレジスタ群の入出力を示したものである。ALU の出力は ACC1、ACC2、IX、MAR のどれかに保持され、メモリからのデータは IR、MDR、PC、MAR のどれかに保持される。PC は、プログラムを 1 ずつ増加させる加算器（inc）と、メモリからのデータを選択する回路（mux1）の出力を保持する。MAR は ALU の出力とメモリからのデータを選択する回路（mux2）の出力を保持する。

図 4.20 は図 4.19 を全体のデータパスに加えたものである。ALU マルチプレクサの入力に、レジスタの出力とグランドに接続された 0V の入力がある。この 0V はデータをそのまま ALU バスを通じてレジスタへ入力するときに使用する。この ALU にはデータをそのまま通過させる機能を持たせていないため、“0”とデータの OR をとることでデータをレジスタへ入力する。

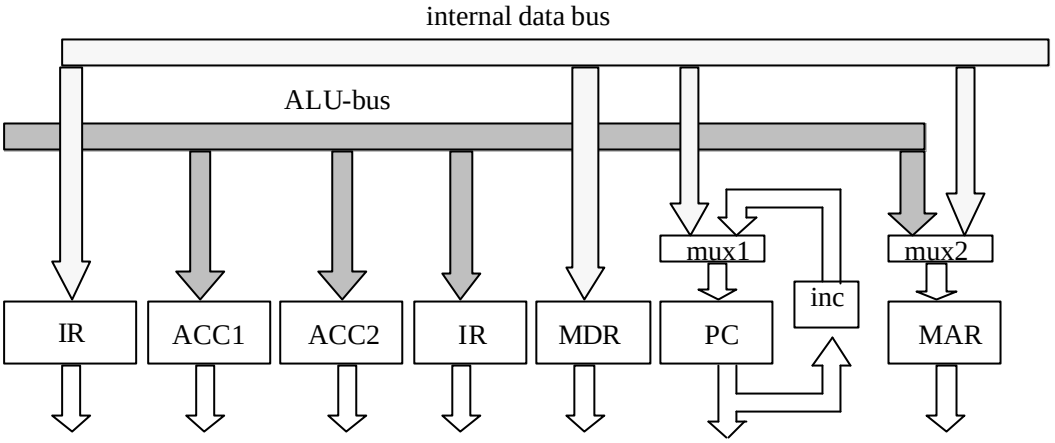


図4.19　ブロック図5

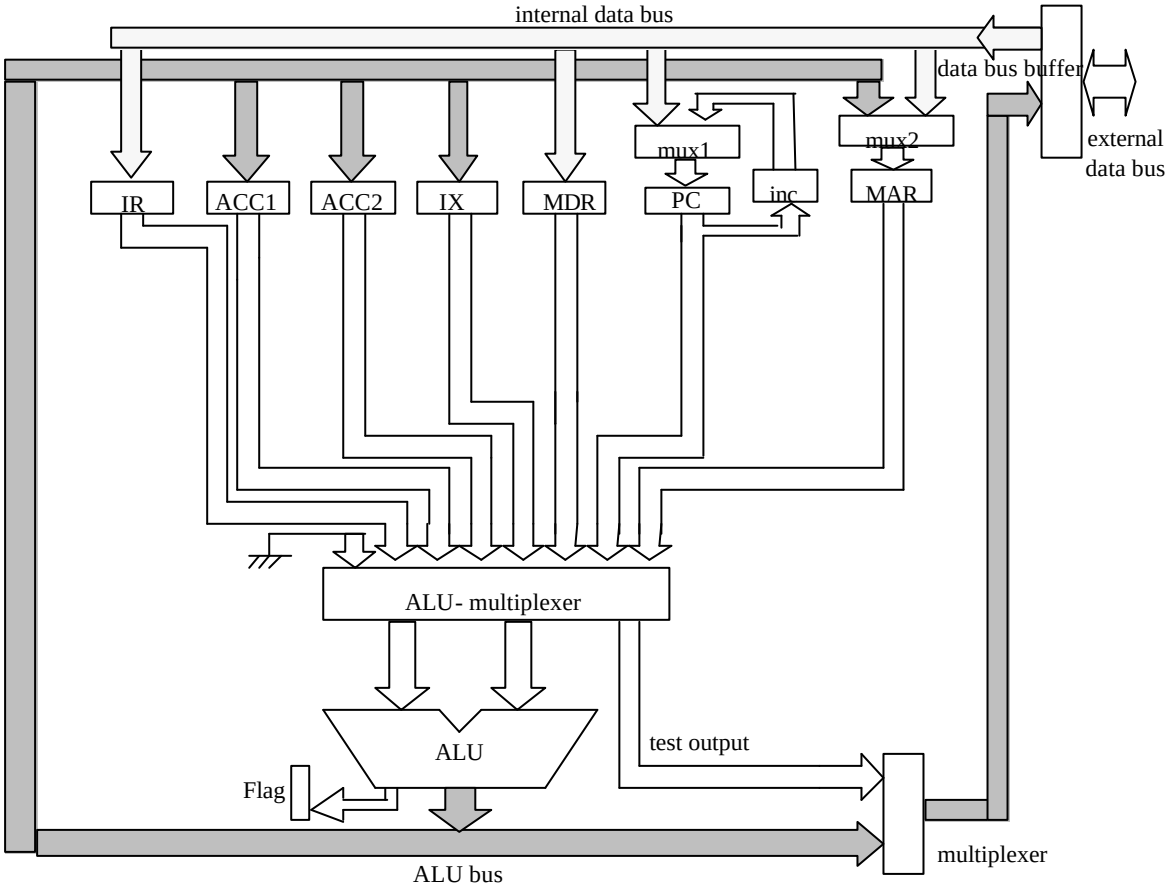


図4.20　ブロック図6

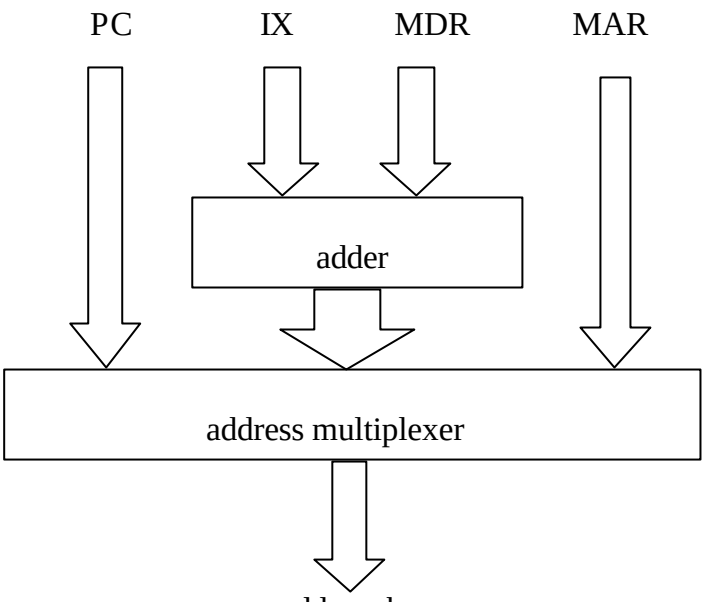


図4.21　ブロック図7

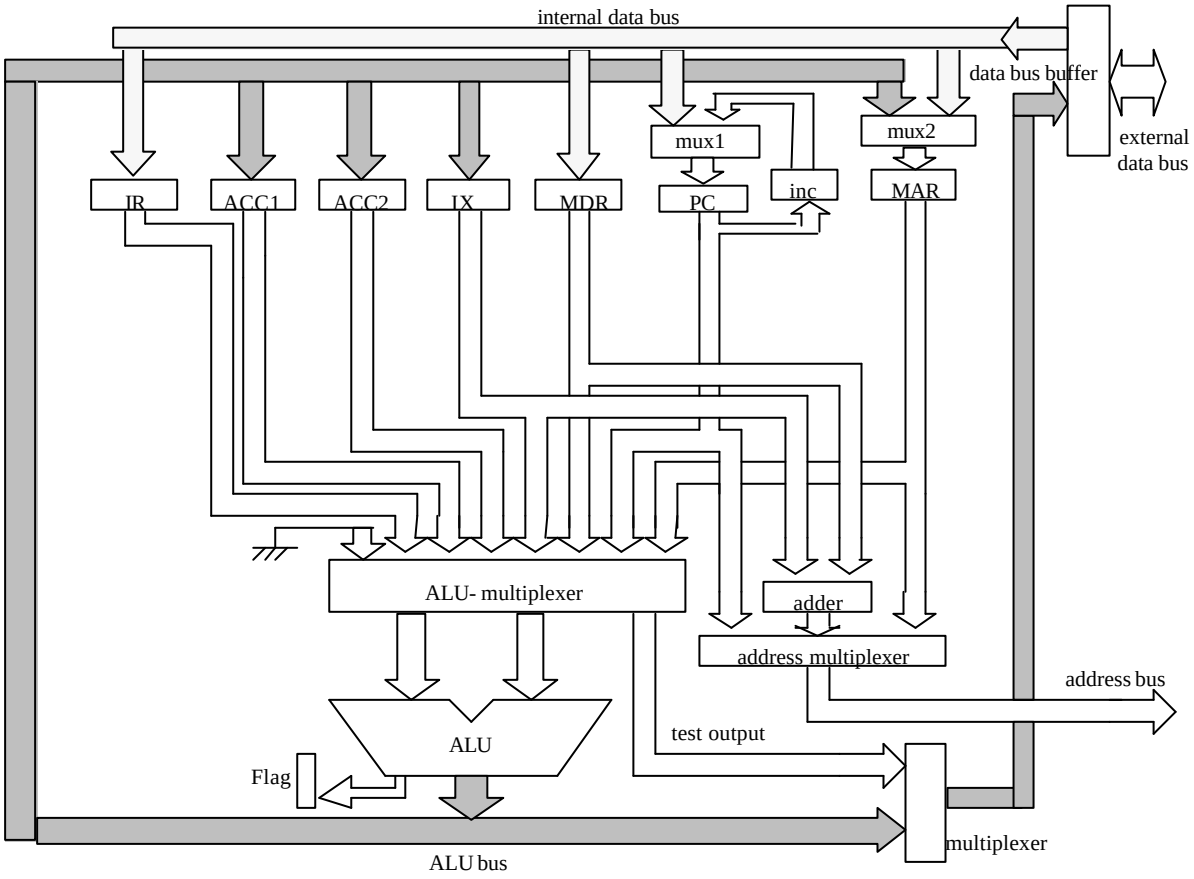


図4.22　ブロック図8

図 4.21 はアドレス生成部の入出力である。アドレスはアドレッシングモードにもとづいて決定されるが、通常どおりプログラムを順番に読む場合、即値アドレスの場合は PC、インデックス修飾アドレスは IX と MDR の加算値、絶対アドレス、レジスタ間接アドレスの場合は MAR がアドレス選択回路（address mutiplexer）によって選択された信号がアドレスバスに載る。

図 4.22 は図 4.21 を全体のデータパスに加えたものである。

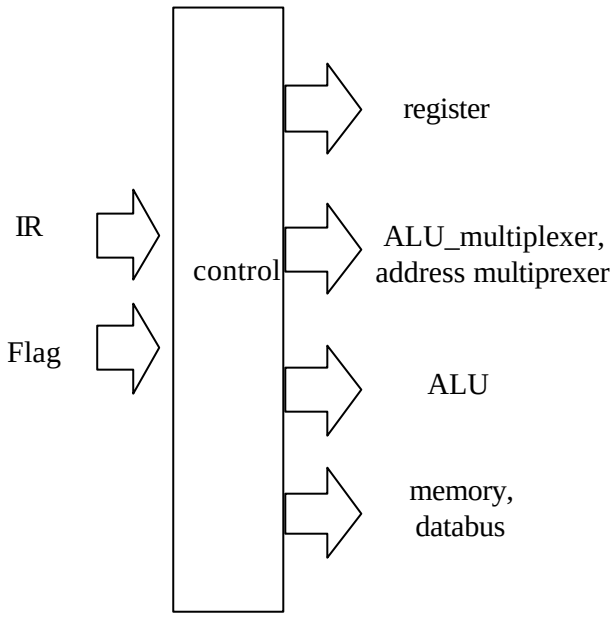


図4.23 ブロック図9

図 4.23 は制御部の入出力である。制御部はステートマシンで構成され、状態を決めるのは IR の内容と Flag の状態である。出力は各レジスタの制御、ALU マルチプレクサ、アドレスマルチプレクサの制御、演算の種類を指定するための ALU の制御、メモリの読み込みか書き込みかの制御、データバスバッファの制御などの線がある。

図 4.24 は図 4.23 を全体のデータパスに加えたものである。他より太い制御線は複数のビット線である。

この図が全体の内部構成である。

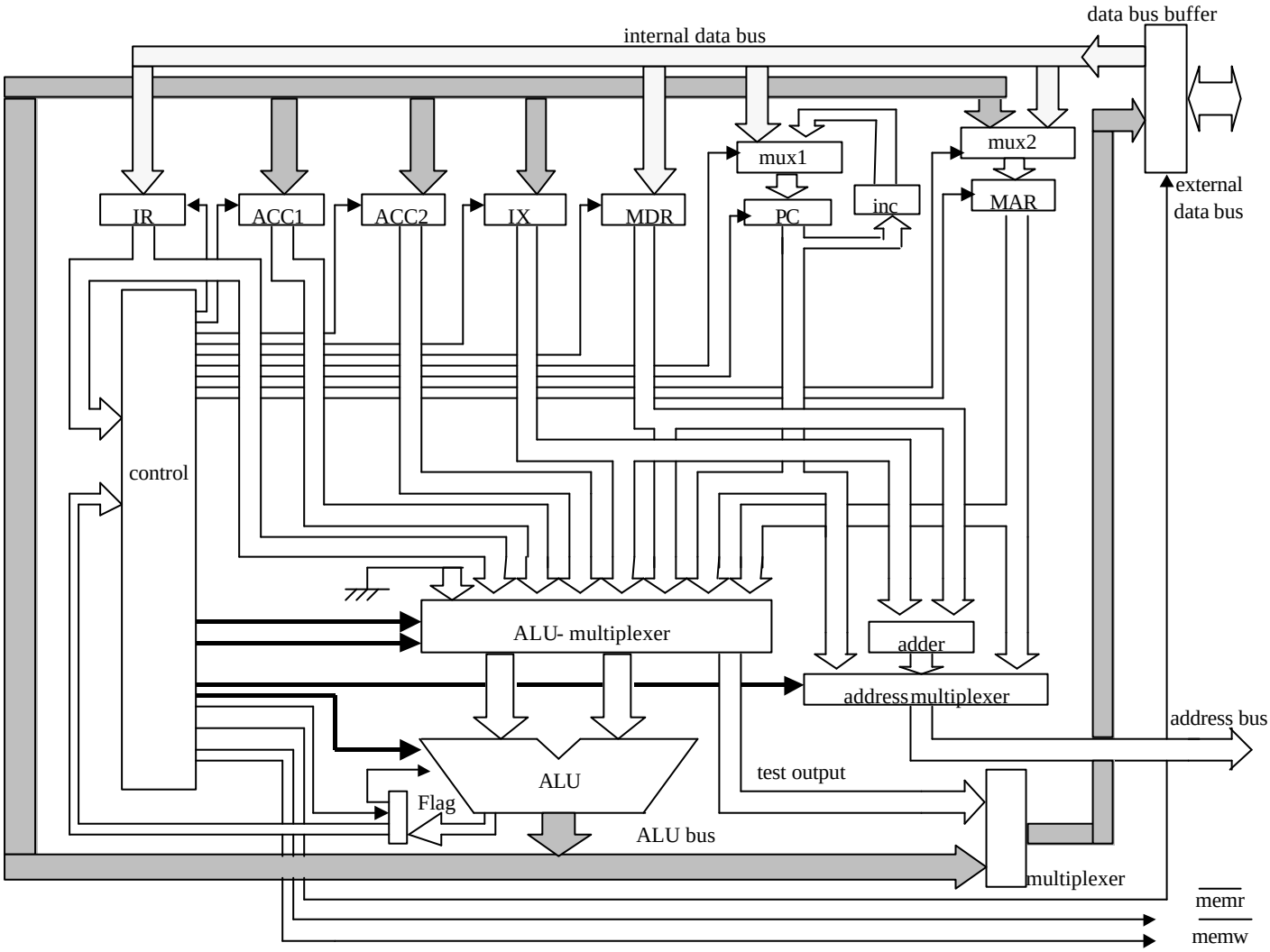


図4.24 ブロック図10

4.2.7 タイミング

各命令、及びアドレッシングモードにより制御信号やレジスタがどうなるかを決める。

表 4.6 はタイミング図で使用する主な制御線とレジスタである。

表4.6　タイミング図の制御信号とレジスタ

$\overline{\text{memr}}$	メモリ・リード。LOWレベルで有効
$\overline{\text{memw}}$	メモリ・ライト。LOWレベルで有効
data	外部データバスに載るデータ
address	アドレスバスに載るデータ
PC	プログラムカウンタの内容
MAR	メモリアドレス・レジスタの内容
IR	インストラクション・レジスタの内容
register	レジスタの内容（ACC1、ACC2、MAR、IX）
a_mux	アドレス選択回路
dbb	データバスバッファ。LOWレベルで外部データバスからデータを取り込み、HIレベルで外部データバスへデータを載せる。
MDR	メモリデータ・レジスタの内容
Flag	フラグレジスタの内容

a_mux : address multiprexer

dbb : data bussbuffer

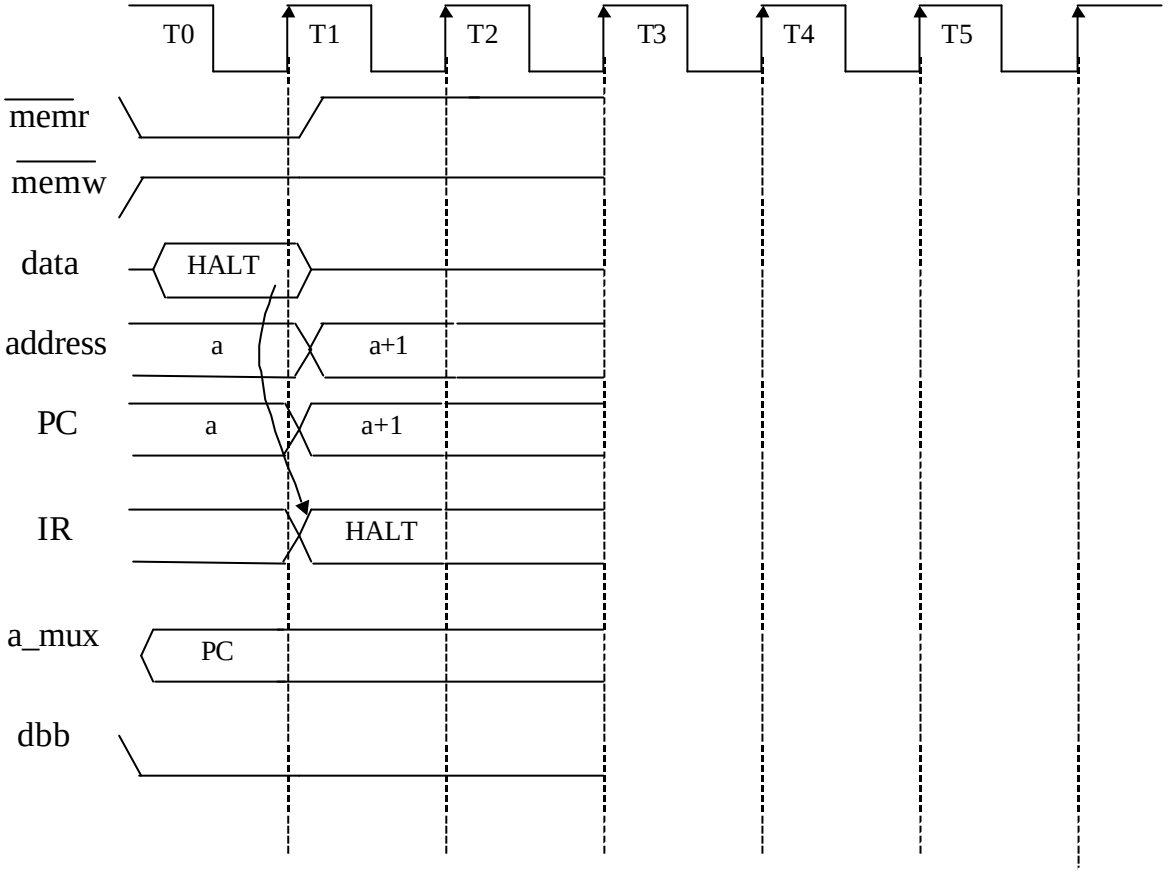


図4.25　停止命令のタイミング

図 4.25 は停止命令のタイミングである。メモリアドレスは PC にあるので、a_mux は PC を選択している。そしてアドレスが a を指していたとして、データバスに停止命令を示すデータが載っており、T1 の立ち上がりで IR に取り込まれる。このときメモリからデータを取り込むので dbb は“ 0 ”でなくてはならない。また次の命令を読むために PC を 1 増加させる。

T2 の立ち上がりで IR 内のデータが制御部に取り込まれ、停止命令が実行される。

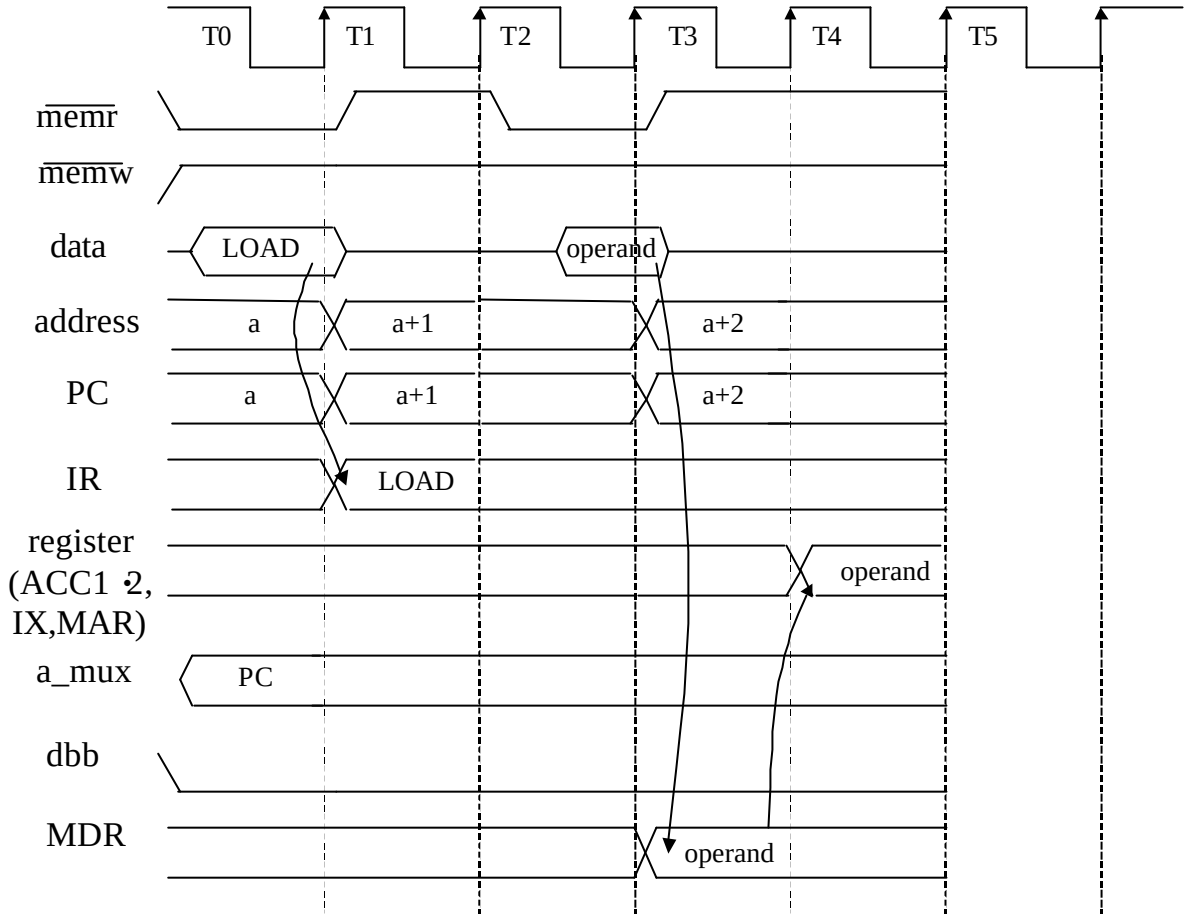


図4.26　ロード命令のタイミング
即値アドレス）

図 4.26 **は、ロード命令（即値アドレス）である。ロード命令が** a **というメモリアドレスにあるとして、**a **は** PC **にあるので** a_mux **は** PC **を選択している。**T1 **の立ち上がりで** IR **にロード命令を取り込み、**T2 **のとき制御部で命令とアドレッシングモードを判断し、オペランドを読みに行く。このとき** a_mux **は** PC **を選択しているので、アドレスは** a+1 **である。**T3 **の立ち上がりで、オペランドは** MDR **に取り込まれる。データは** T4 **でレジスタ（**ACC1、ACC2、IX、MAR **のいずれか）に保持される。**

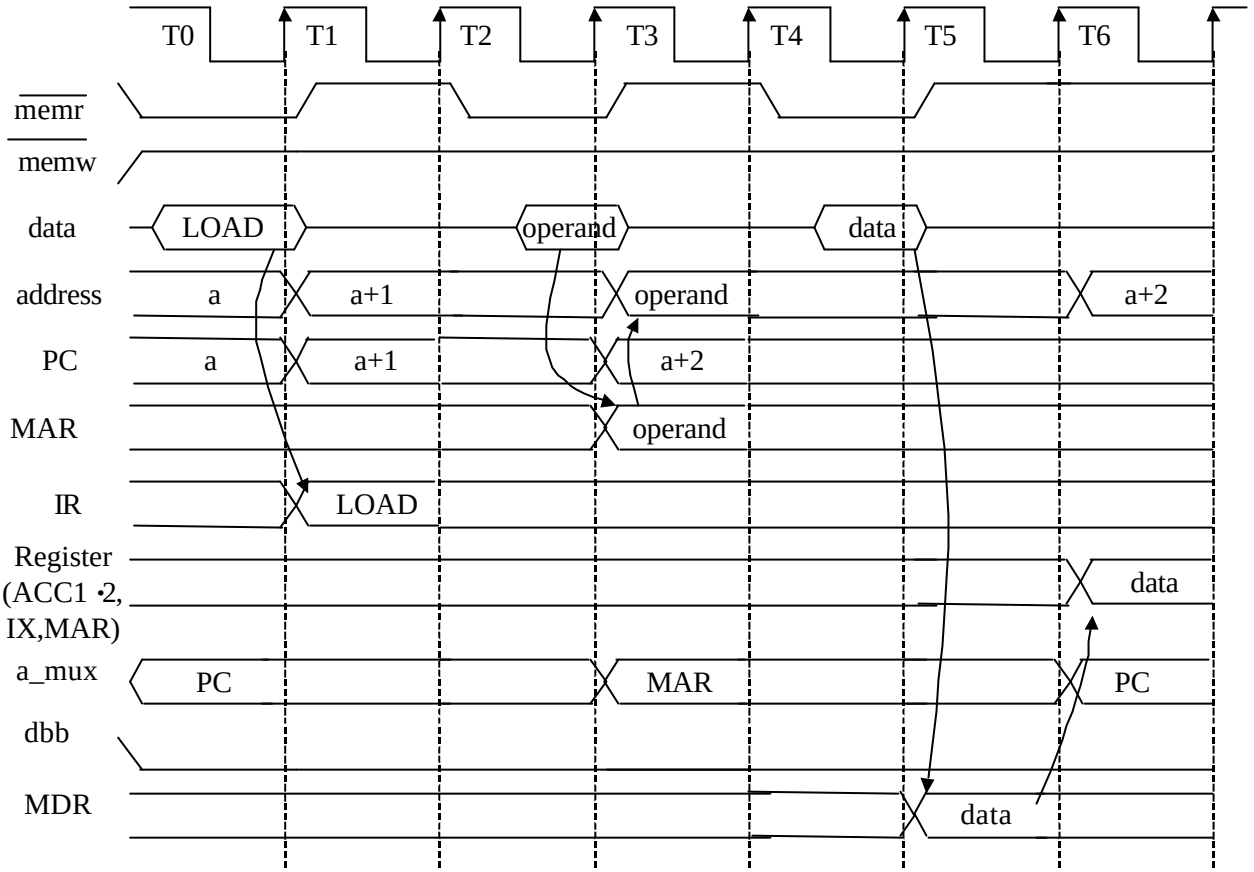


図4.27　ロード命令のタイミング
絶対アドレス）

図 4.27 **はロード命令（絶対アドレス）である。ロード命令が** a **というメモリアドレスにあるとして、**a **は** PC **にあるので** a_mux **は** PC **を選択している。**T1 **の立ち上がりで** IR **にロード命令を取り込み、**T2 **のとき制御部で命令とアドレッシングモードを判断し、オ ペランドを読みに行く。このとき** a_mux **は** PC **を選択しているので、アドレスは** a+1 **である。**T3 **の立ち上がりで、オペランドは** MAR **に取り込まれる。**a_mux **は** MAR **を選択し、**MAR **の内容がアドレスとなり、**T5 **で** MDR **にデータが取り込まれ、データは** T6 **でレジスタ（**ACC1、ACC2、IX、MAR **のいずれか）に保持される。**

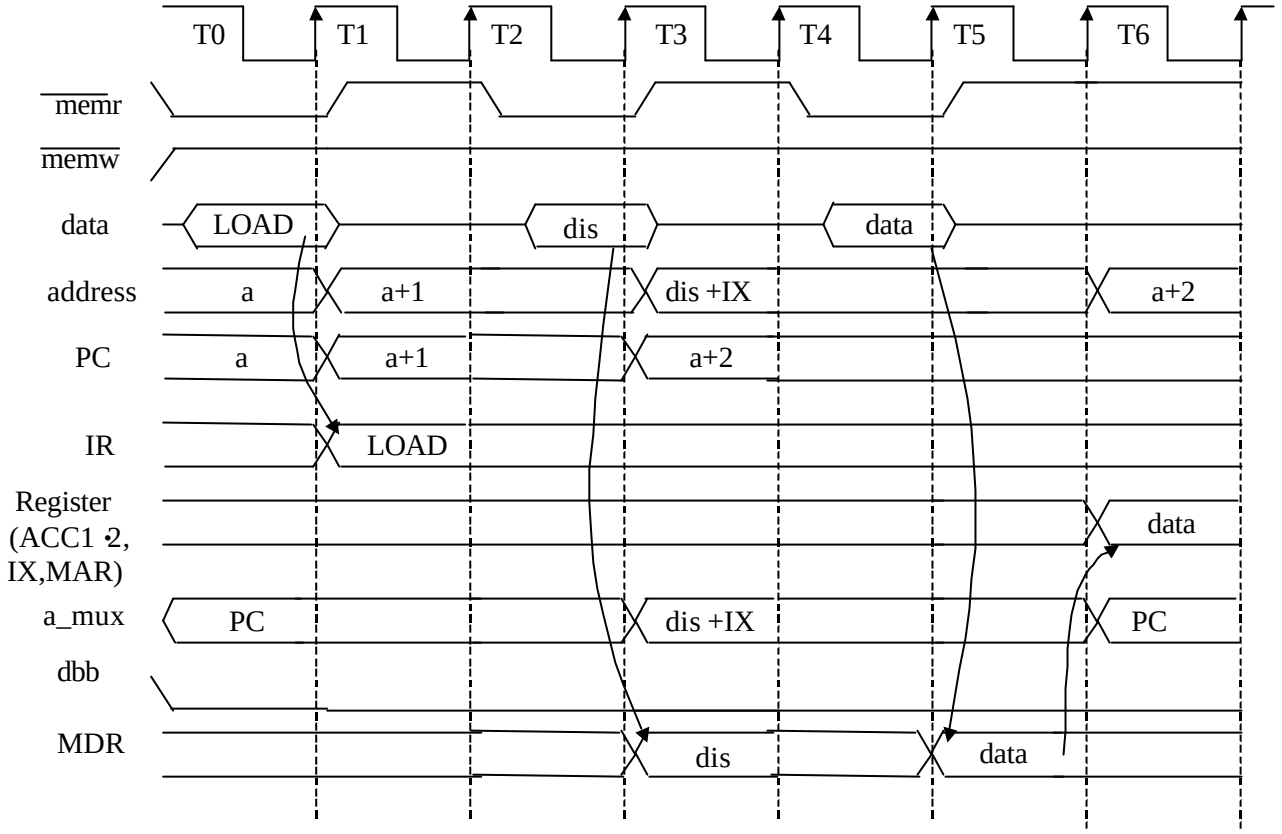


図4.28　ロード命令のタイミング
（インデックス修飾アドレス）

図 4.28 **はロード命令（インデックス修飾アドレス）である。ロード命令が** a **というメモリアドレスにあるとして、**a **は** PC **にあるので** a_mux **は** PC **を選択している。**T1 **の立ち上がりで** IR **にロード命令を取り込み、**T2 **のとき制御部で命令とアドレッシングモードを判断し、ディスプレースメント（**dis**）を読みに行く。このとき** a_mux **は** PC **を選択しているので、アドレスは** a+1 **である。**T3 **の立ち上がりで、ディスプレースメントは** MDR **に取り込まれる。**a_mux **はディスプレースメントとインデックスレジスタの内容の加算したものを選択し、それがアドレスとなりなる。そして** T5 **で** MDR **にデータが取り込まれ、データは** T6 **でレジスタ（**ACC1**、**ACC2**、**IX**、**MAR **のいずれか）に保持される。**

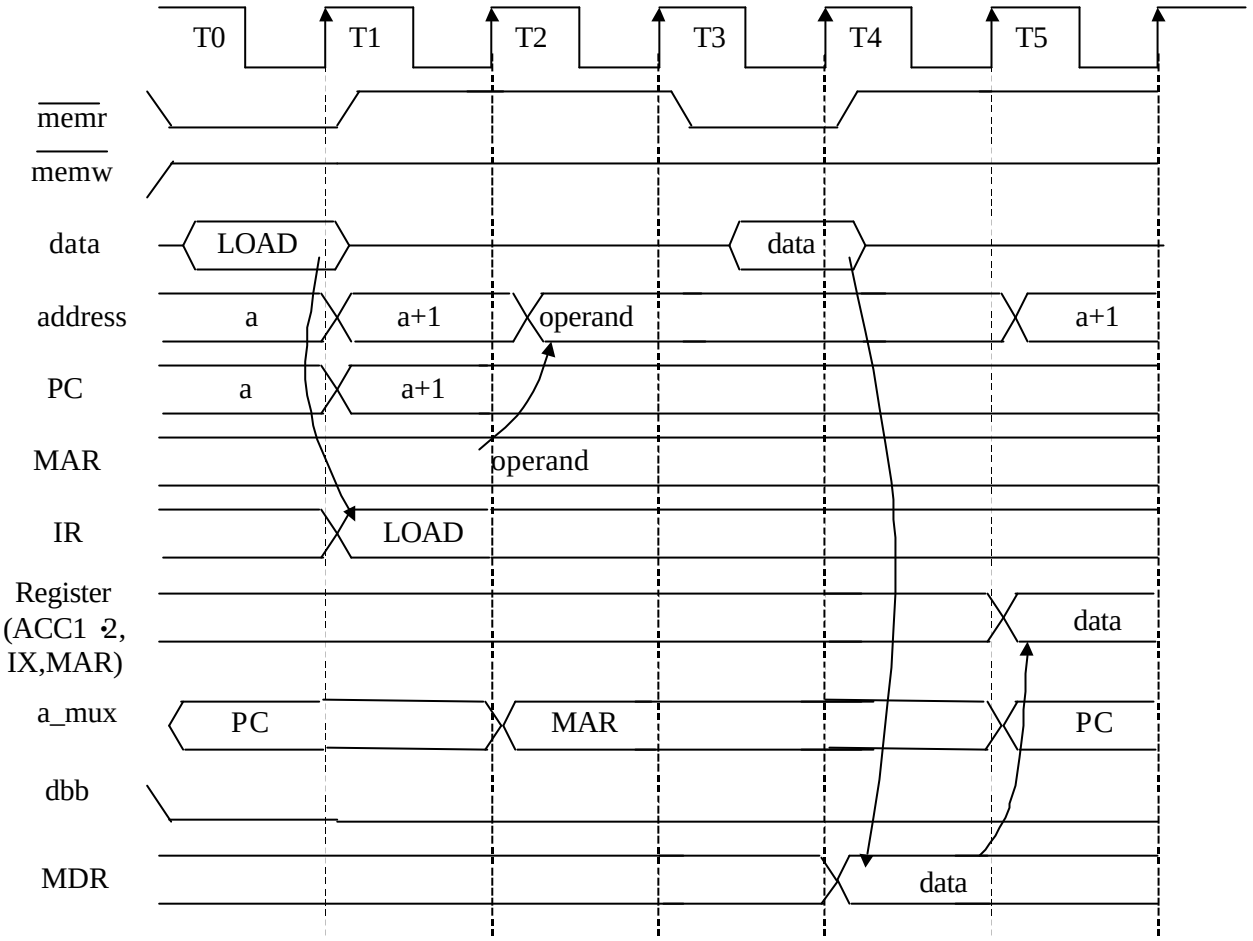


図4.29　ロード命令のタイミング
（レジスタ間接アドレス）

図 4.29 はロード命令（レジスタ間接アドレス）である。ロード命令が a というメモリアドレスにあるとして、a は PC にあるので a_mux は PC を選択している。T1 の立ち上がりで IR にロード命令を取り込み、T2 のとき制御部で命令とアドレッシングモードを判断し、オペランドを読みに行く。このとき a_mux は MAR を選択しているので、アドレスは MAR の内容となる。T4 の立ち上がりで、データが MDR に取り込まれる。データは T5 でレジスタ（ACC1、ACC2、IX、MAR のいずれか）に保持される。

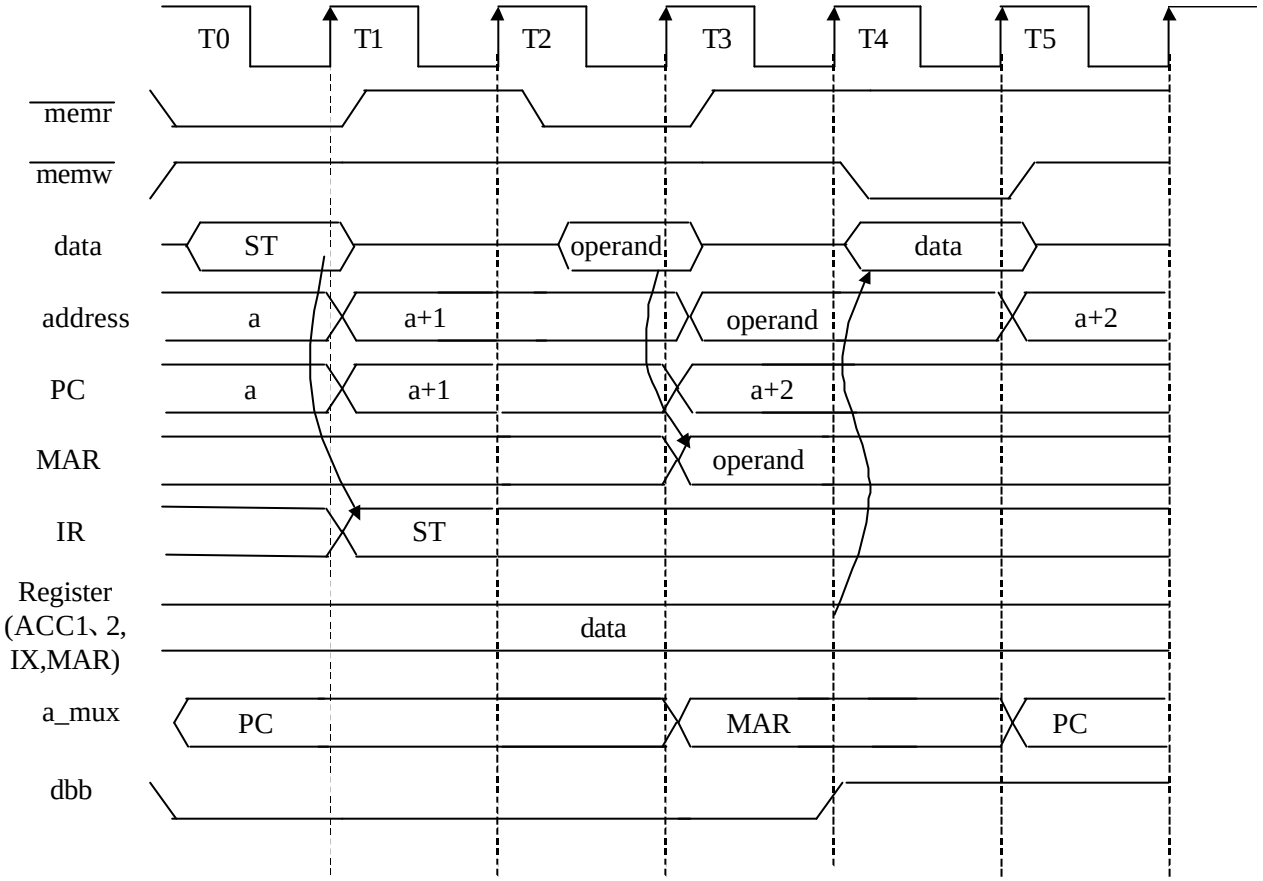


図4.30　ストア命令のタイミング
絶対アドレス）

図 4.30 **はストア命令（絶対アドレス）のタイミングである。ストア命令が** a **というメモリアドレスにあるとして、**a **は** PC **にあるので** a_mux **は** PC **を選択している。**T1 **の立ち上がりで** IR **にストア命令を取り込み、**T2 **のとき制御部で命令とアドレッシングモードを判断し、オペランドを読みに行く。**T3 **の立ち上がりで、データが** MAR **に取り込まれる。このとき** a_mux **は** MAR **を選択しているので、アドレスは** MAR **の内容となる。**T4 **でレジスタ（**ACC1、ACC2、IX、MAR **のいずれか）のデータが外部データバスに載り、**T5 **の立ち上がりでメモリに保存される。このときメモリライト信号は“**0**”、**dbb **は“**1**”でなくてはならない。**

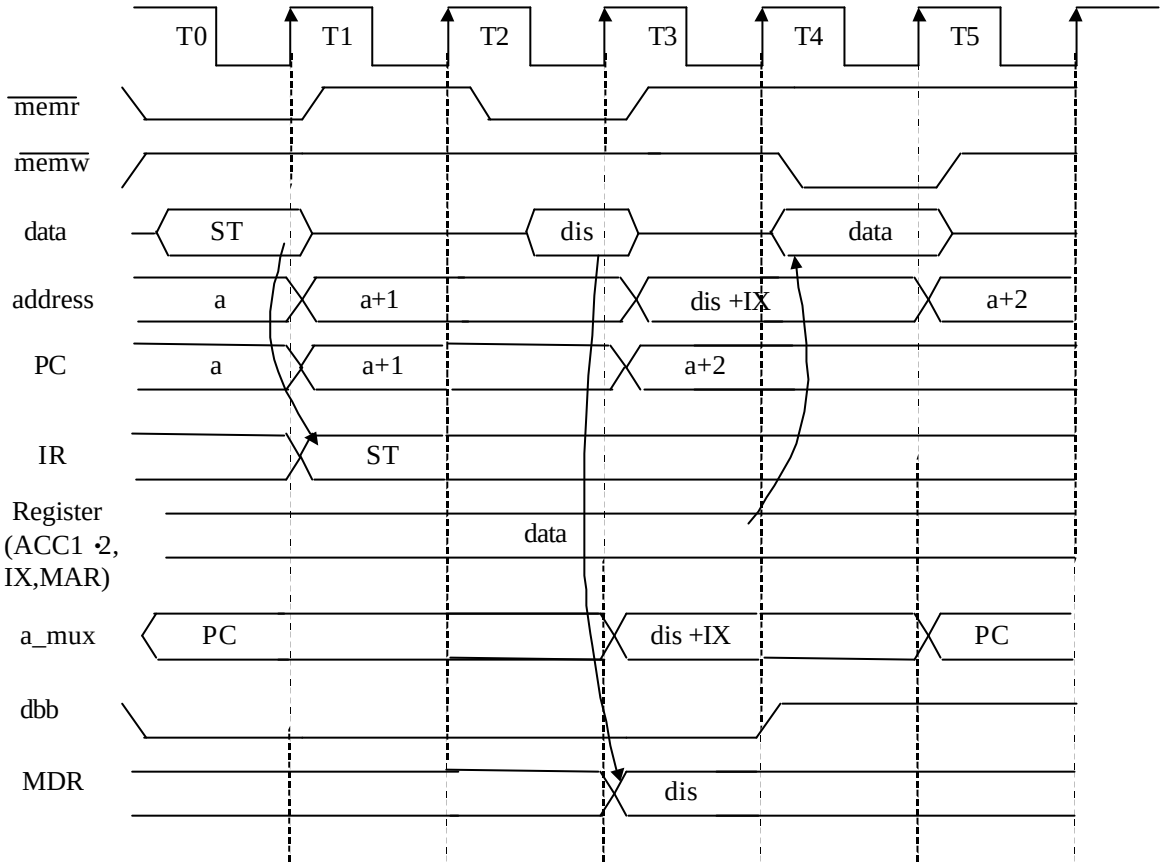


図4.31　ストア命令のタイミング
（インデックス修飾アドレス）

図 4.31 はストア命令（インデックス修飾アドレス）のタイミングである。ストア命令が a というメモリアドレスにあるとして、a は PC にあるので a_mux は PC を選択している。T1 の立ち上がりで IR にストア命令を取り込み、T2 のとき制御部で命令とアドレッシングモードを判断し、ディスプレースメントを読みに行く。T3 の立ち上がりで、ディスプレースメントが MDR に取り込まれる。このとき a_mux はディスプレースメントとインデックスレジスタの加算結果を選択し、アドレスとなる。T4 でレジスタ（ACC1、ACC2、IX、MAR のいずれか）のデータが外部データバスに載り、T5 の立ち上がりでメモリに保存される。このときメモリライト信号は“0”、dbb は“1”でなくてはならない。

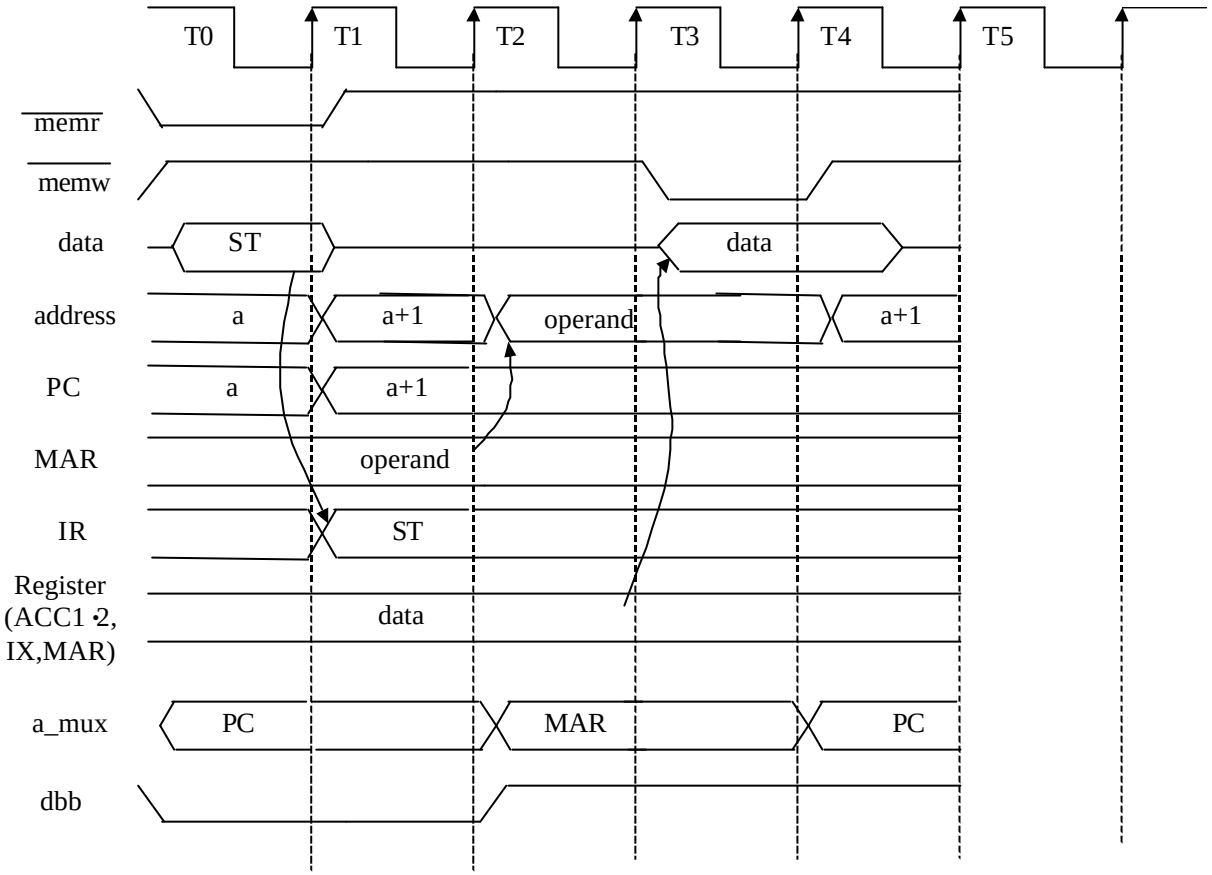


図4.32　ストア命令のタイミング
（レジスタ間接アドレス）

図 4.32 はストア命令（レジスタ間接アドレス）のタイミングである。ストア命令が a というメモリアドレスにあるとして、a は PC にあるので a_mux は PC を選択している。T1 の立ち上がりで IR にストア命令を取り込み、T2 のとき制御部で命令とアドレッシングモードを判断し、a_mux は MAR を選択する。アドレスは MAR 内のデータで、外部データバスにはレジスタ（ACC1、ACC2、IX、MAR のいずれか）のデータが載る。そして T4 の立ち上がりでメモリに保存される。このときメモリライト信号は“ 0 ”、dbb は“ 1 ”でなくてはならない。

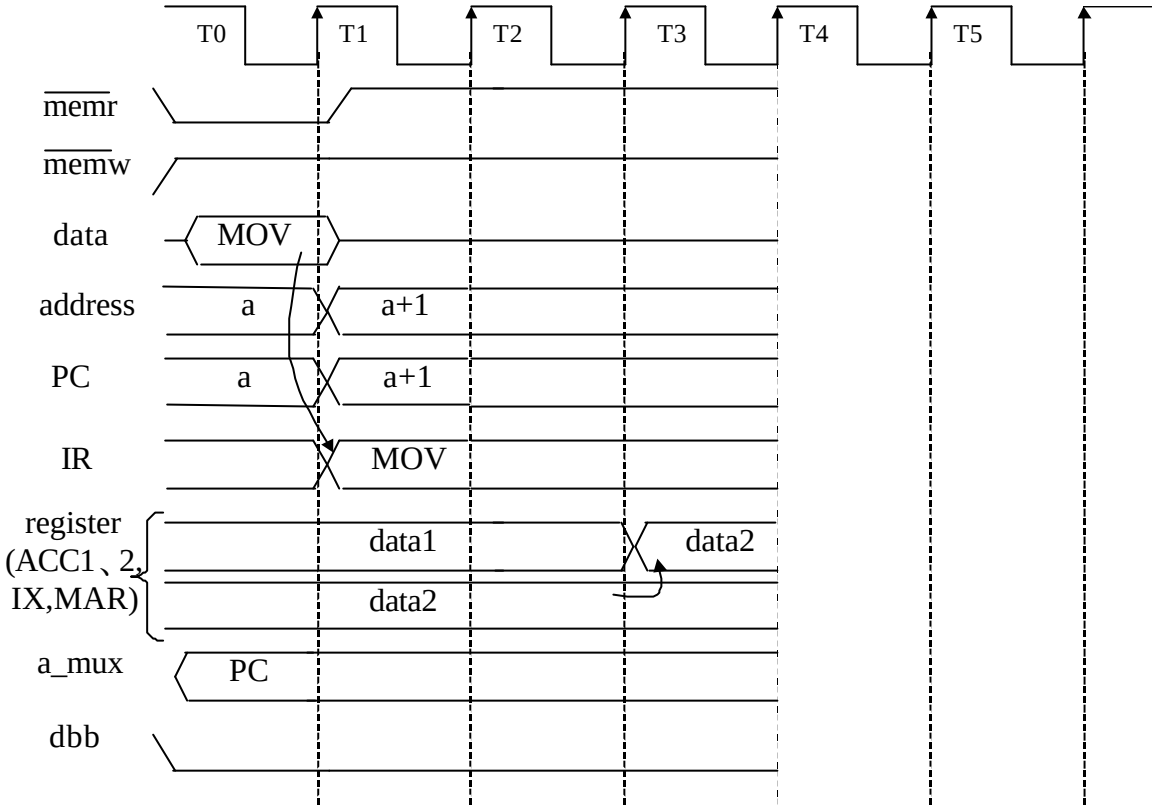


図4.33　データ移動命令のタイミング

図 4.33 はデータ移動命令のタイミングである。命令が a というメモリアドレスにあるとして、a は PC にあるので a_mux は PC を選択している。T1 の立ち上がりで IR に移動命令を取り込み、T2 のとき制御部で命令を判断する。このときデータを出すレジスタとデータを受け取るレジスタを制御し、ALU バスに載ったデータ（data2）は T3 でデータをレジスタに保持する。

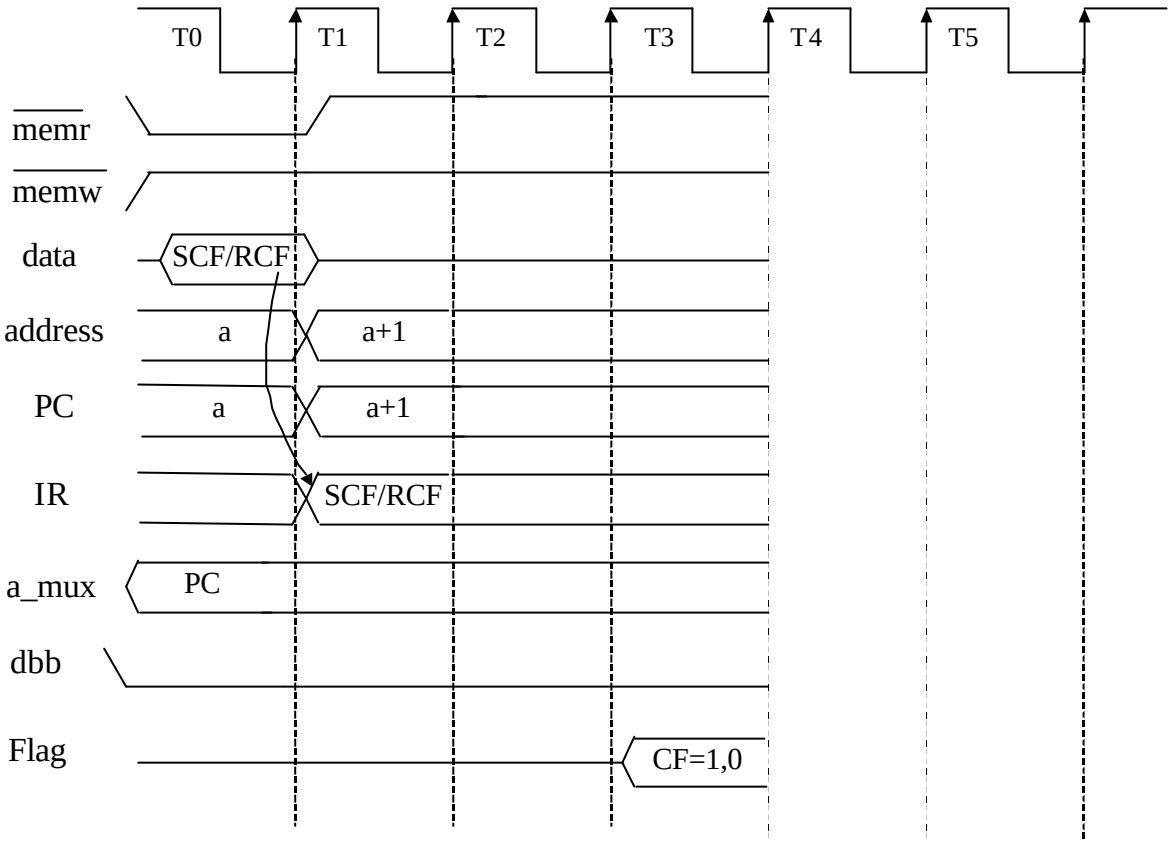


図4.34　フラグセット命令のタイミング

図 4.34 **はフラグセット命令のタイミングである。命令が** a **というメモリアドレスにあるとして、**a **は** PC **にあるので** a_mux **は** PC **を選択している。**T1 **の立ち上がりで** IR **にフラグセット命令を取り込み、**T2 **のとき制御部で命令を判断する。セット命令であれば“** 1 **”、リセット命令であれば“** 0 **”をフラグレジスタに入力し、**T3 **の立ち上がりで確定する。**

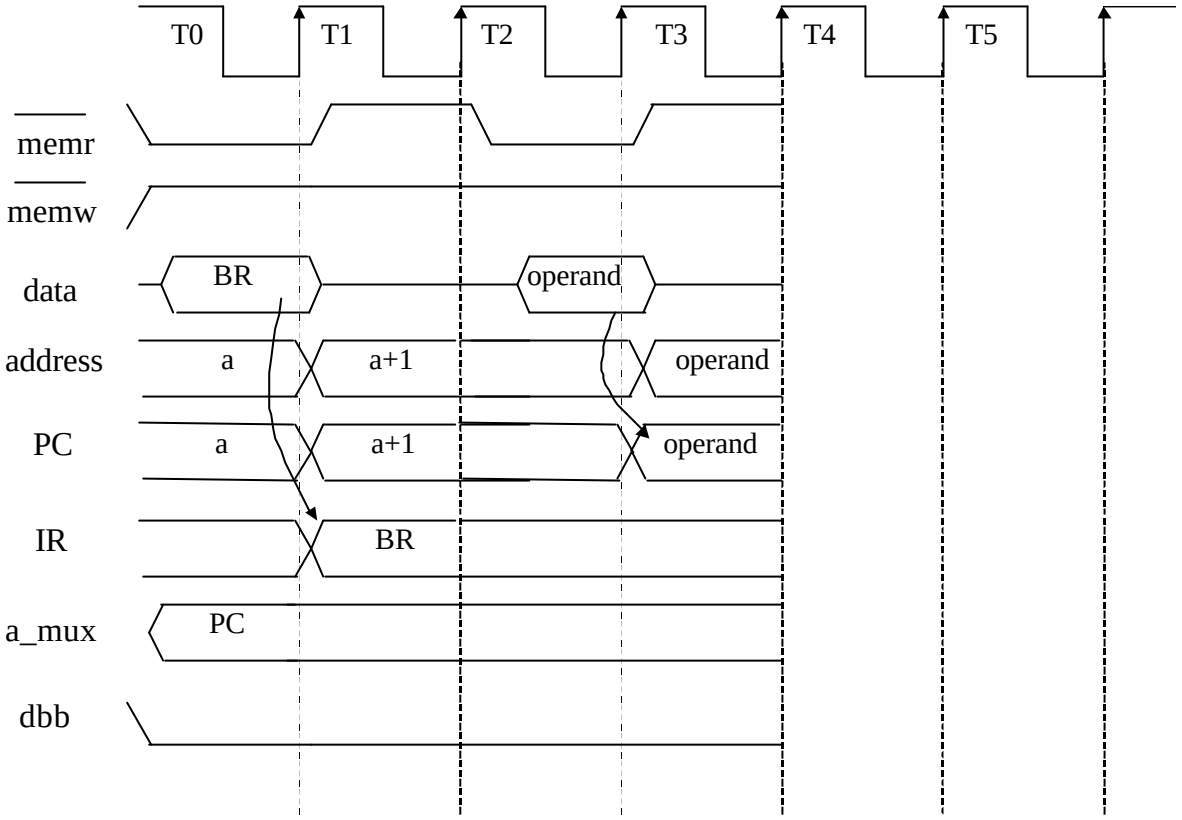


図4.35　分岐命令
条件不成立の場合）

図 4.35 は条件不成立の場合の分岐命令のタイミングである。命令が a というメモリアドレスにあるとして、a は PC にあるので a_mux は PC を選択している。T1 の立ち上がりで IR に分岐命令を取り込み、T2 のとき制御部で命令を判断する。このとき制御部はフラグの状態をみて、条件が不成立の場合は次の命令を読むためにオペランドを T3 の立ち上がりで PC に取り込む。そして条件が成立するまで繰り返す。

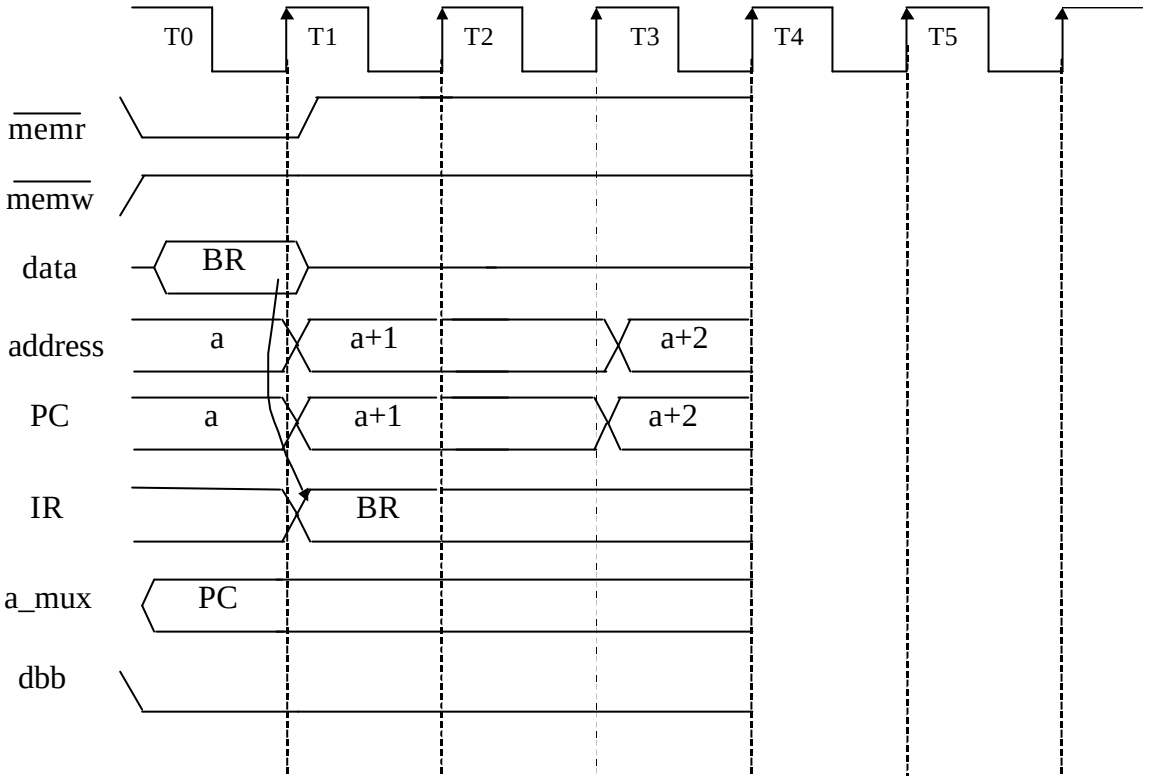


図4.36　分岐命令
条件成立の場合）

図 4.36 は条件不成立の場合の分岐命令のタイミングである。命令が a というメモリアドレスにあるとして、a は PC にあるので a_mux は PC を選択している。T1 の立ち上がりで IR に分岐命令を取り込み、T2 のとき制御部で命令を判断する。このとき制御部はフラグの状態をみて、条件が成立の場合は次の命令を読むために PC 内のデータをアドレスとしてバスに載せる。

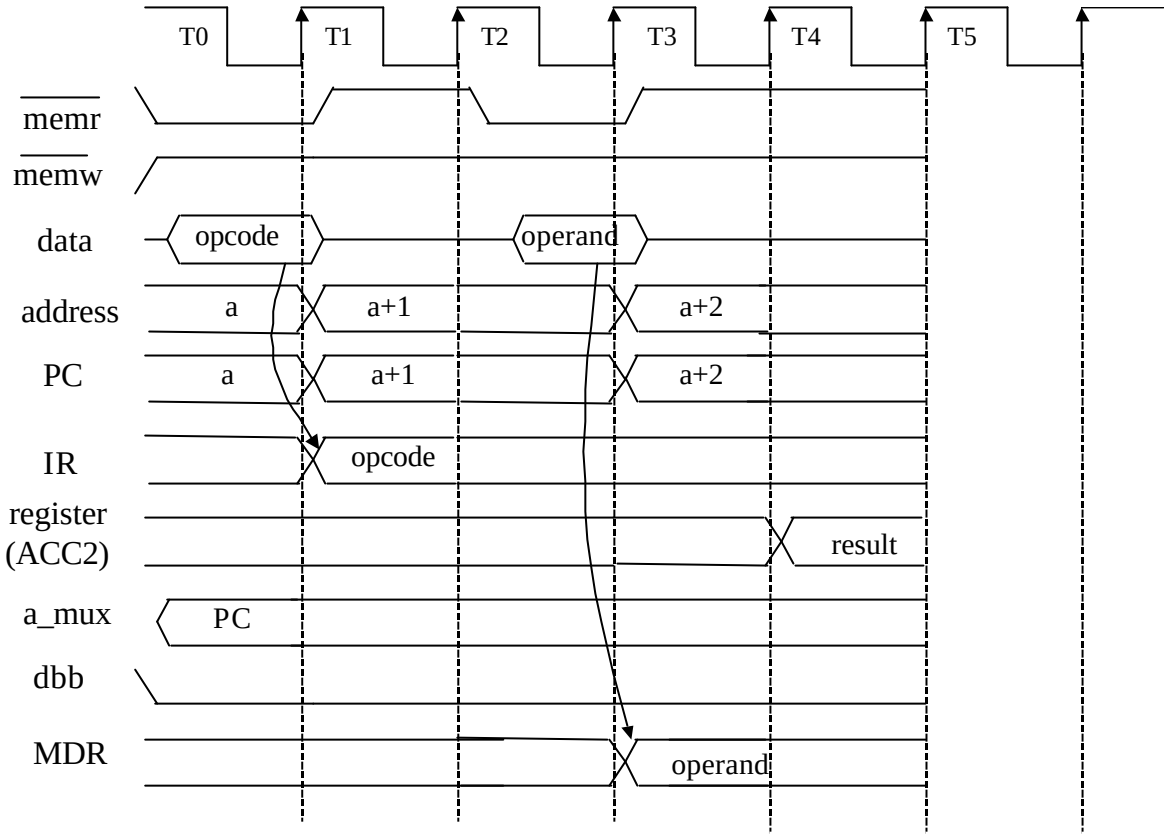


図4.37　演算命令のタイミング
即値アドレス）

図 4.37 は演算命令（即値アドレス）のタイミングである。命令が a というメモリアドレスにあるとして、a は PC にあるので a_mux は PC を選択している。T1 の立ち上がりで IR に演算命令を取り込み、T2 のとき制御部で命令とアドレッシングモードを判断する。この場合、オペランドのアドレスは PC 内のデータ（a+1）であり、T3 の立ち上がりで MDR にオペランドが取り込まれる。そしてオペランドがそのまま演算対象となるため、T3 の区間内で演算を行い、T4 の立ち上がりで演算結果（result）が ACC2 に保持される。

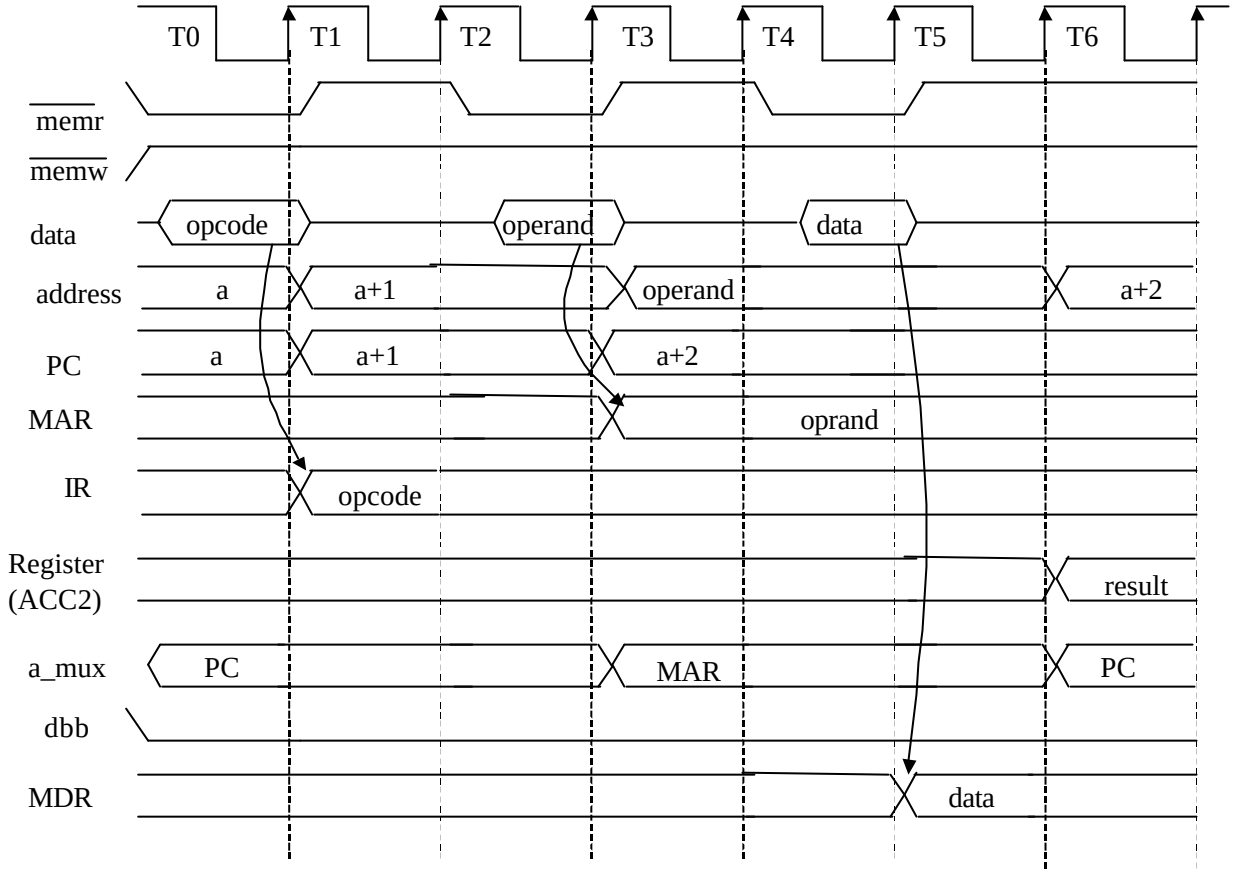


図4.38　演算命令のタイミング
絶対アドレス)

図 4.38 は演算命令（絶対アドレス）のタイミングである。命令が a というメモリアドレスにあるとして、a は PC にあるので a_mux は PC を選択している。T1 の立ち上がりで IR に演算命令を取り込み、T2 のとき制御部で命令とアドレッシングモードを判断する。この場合、オペランドのアドレスは PC 内のデータ（a+1）であり、T3 の立ち上がりで MAR にオペランドが取り込まれる。そして a_mux は MAR を選択し、MAR 内のデータがアドレスとなる。T5 の立ち上がりでメモリからのデータを MDR に取り込み、T5 の区間内で演算を行い、T6 の立ち上がりで演算結果（result）が ACC2 に保持される。

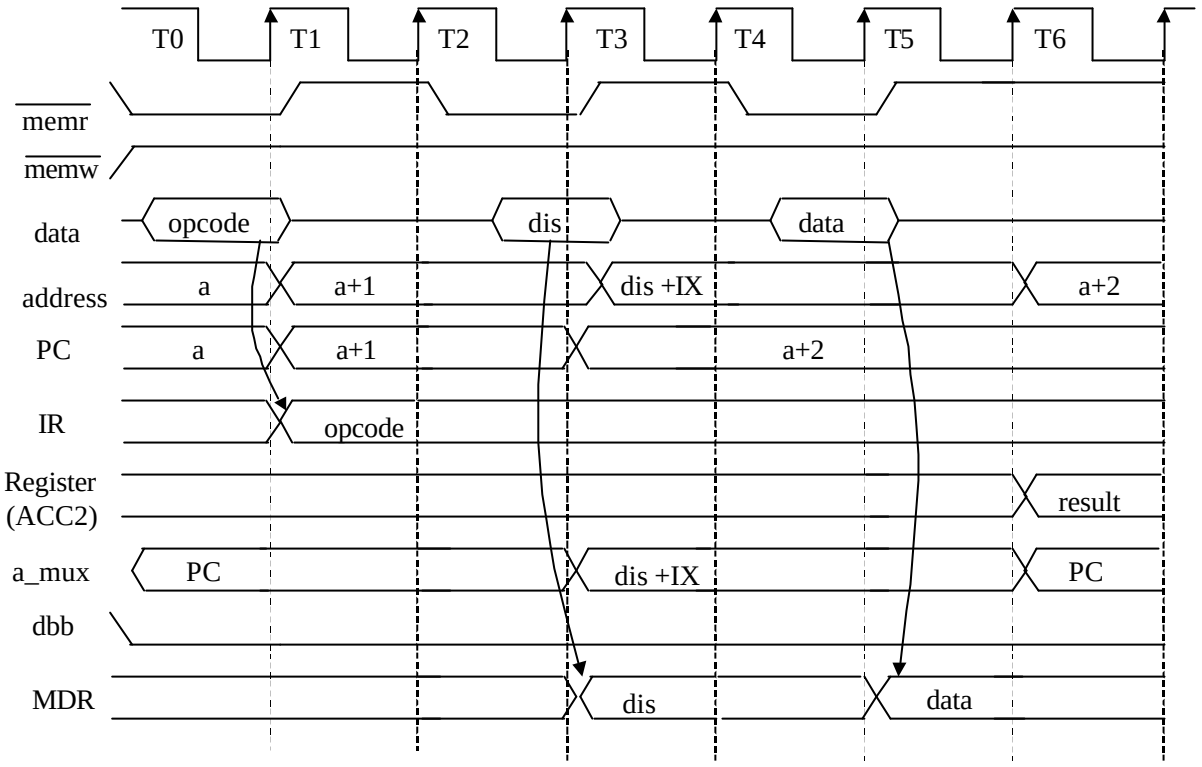


図4.39　演算命令のタイミング
（インデックス修飾アドレス）

図 4.39 は演算命令（インデックス修飾アドレス）のタイミングである。命令が a というメモリアドレスにあるとして、a は PC にあるので a_mux は PC を選択している。T1 の立ち上がりで IR に演算命令を取り込み、T2 のとき制御部で命令とアドレッシングモードを判断する。この場合、ディスプレースメント（dis）のアドレスは PC 内のデータ（a+1）であり、T3 の立ち上がりで MDR にディスプレースメントが取り込まれる。そして a_mux はディスプレースメントと IX の加算結果を選択し、そのデータがアドレスとなる。T5 の立ち上がりでメモリからのデータを MDR に取り込み、T5 の区間内で演算を行い、T6 の立ち上がりで演算結果（result）が ACC2 に保持される。

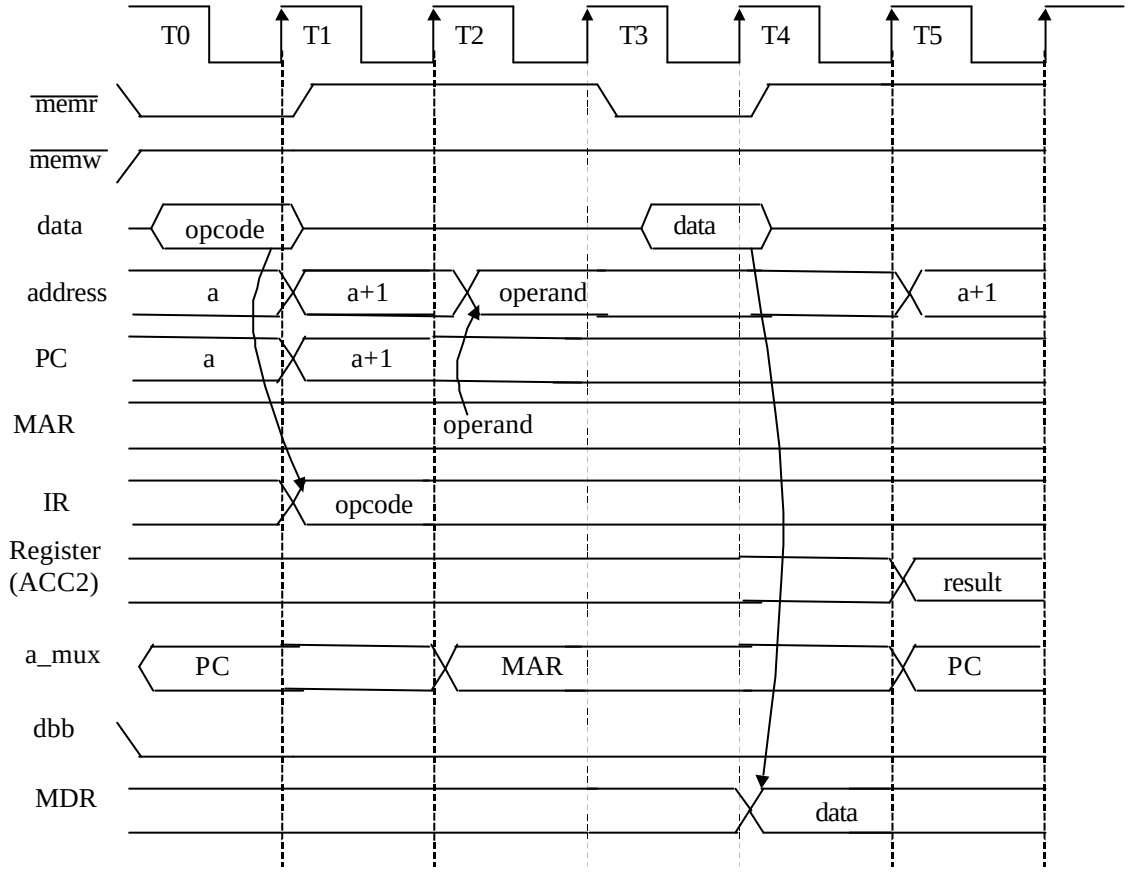


図4.40　演算命令のタイミング
（レジスタ間接アドレス）

図 4.40 は演算命令（レジスタ間接アドレス）のタイミングである。命令が a というメモリアドレスにあるとして、a は PC にあるので a_mux は PC を選択している。T1 の立ち上がりで IR に演算命令を取り込み、T2 のとき制御部で命令とアドレッシングモードを判断する。この場合、オペランドのアドレスは MAR 内のデータであり、a_mux は MAR を選択し、そのデータがアドレスとなる。そして T4 の立ち上がりで MDR にメモリからのデータが取り込まれ、T4 の区間内で演算を行い、T5 の立ち上がりで演算結果（result）が ACC2 に保持される。

4.2.8 実行フェーズ

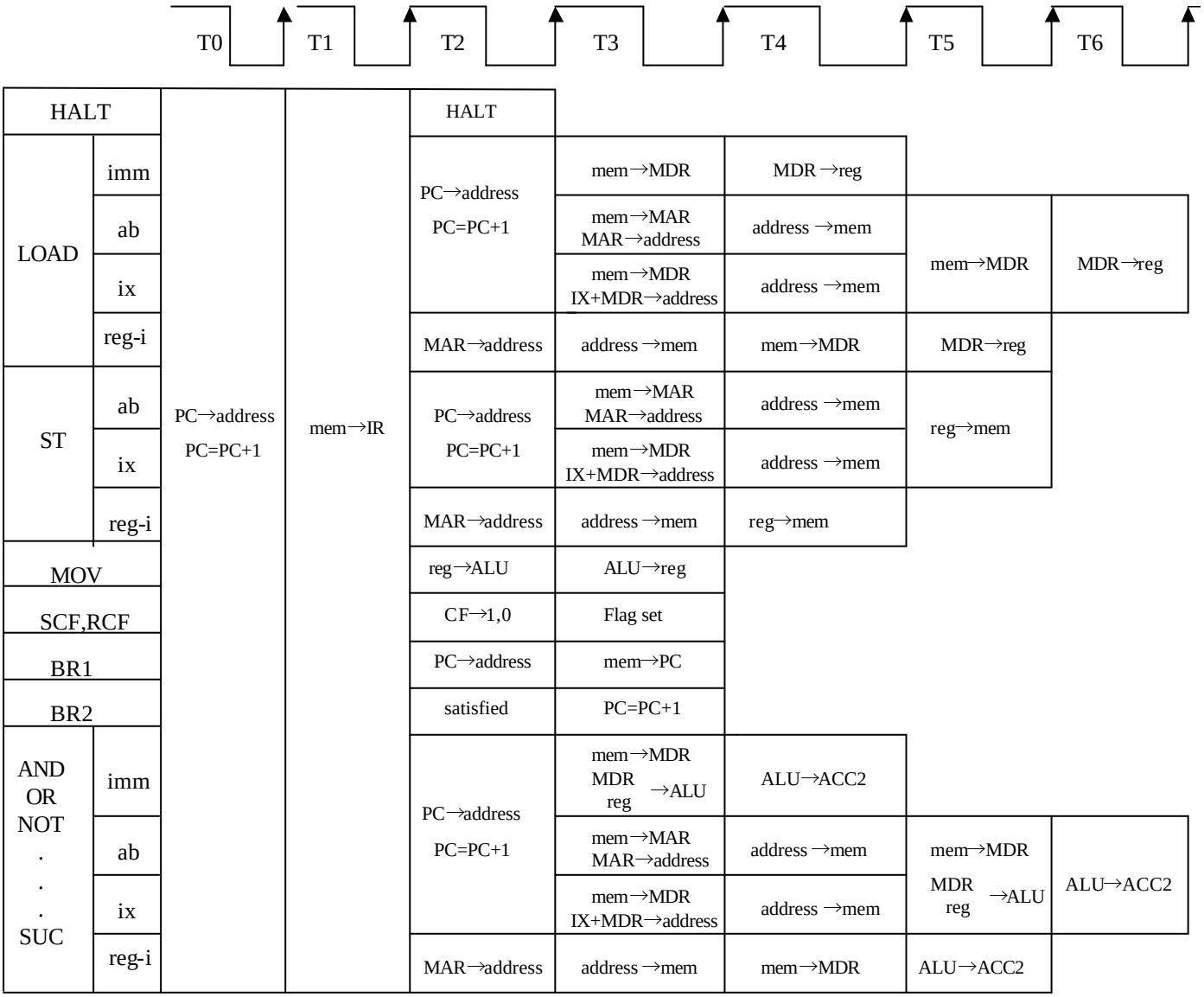
4.2.7 のタイミング図をもとに、各命令の動作を実行フェーズとしてまとめる。表 4.7 は CPU 内の動作を簡潔に表すための対応表である。また実行フェーズを図 4.41 に示す。

表4.7 対応表

PC=PC+1	PCを1増加させる。
PC→address	PCのデータをアドレスバスに載せる。
MAR→address	MARのデータをアドレスバスに載せる。
IX+MDR→address	IXとMDRの加算結果をアドレスバスに載せる。
mem→IR	メモリからのデータをIRに取り込む。
mem→MDR	メモリからのデータをIRに取り込む。
mem→MAR	メモリからのデータをMARに取り込む。
mem→PC	メモリからのデータをPCに取り込む。
reg→mem	レジスタのデータをメモリへ書き込む。
MDR →reg	MDRのデータをレジスタへ移す。
ALU→ACC2	ALUの出力をACC2に取り込む。
MDR →ALU reg	MDRとレジスタの演算結果がALUから出力される。
ALU→reg	ALUの出力をレジスタに取り込む。
reg→ALU	レジスタのデータをALUに出力させる。
CF→1,0	CFに “1 ”または “0 ”を入力し、クロックの立ち上がりでセット。
Flag set	CFに “1 ”または “0 ”をセットする。
satisfied	条件成立。
HALT	停止状態。

対応表と実行フェーズを見ると、動作の種類はさほど多くないことがわかる。

また、命令を読み込む過程は全ての命令において共通であり、そしてロード、ストア、演算命令の動作は、最終的にレジスタへデータを格納する、メモリにデータを格納する、演算するという違いはあるものの、それに到るまでの過程は似ている。この 3 つの命令については、アドレッシングモードによる動作の違いのほうが大きい。



imm - 即値アドレス　　ab - 絶対アドレス
ix - インデックス修飾アドレス　reg_i - レジスタ間接アドレス

図4.41　実行フェーズ

4.2.9 状態遷移図

実行フェーズをもとにCPUの動作を状態遷移図で表す。各状態から制御信号を定義すれば、この図から制御部を設計することができる。

図4.42は実行フェーズを状態遷移図で表したものである。

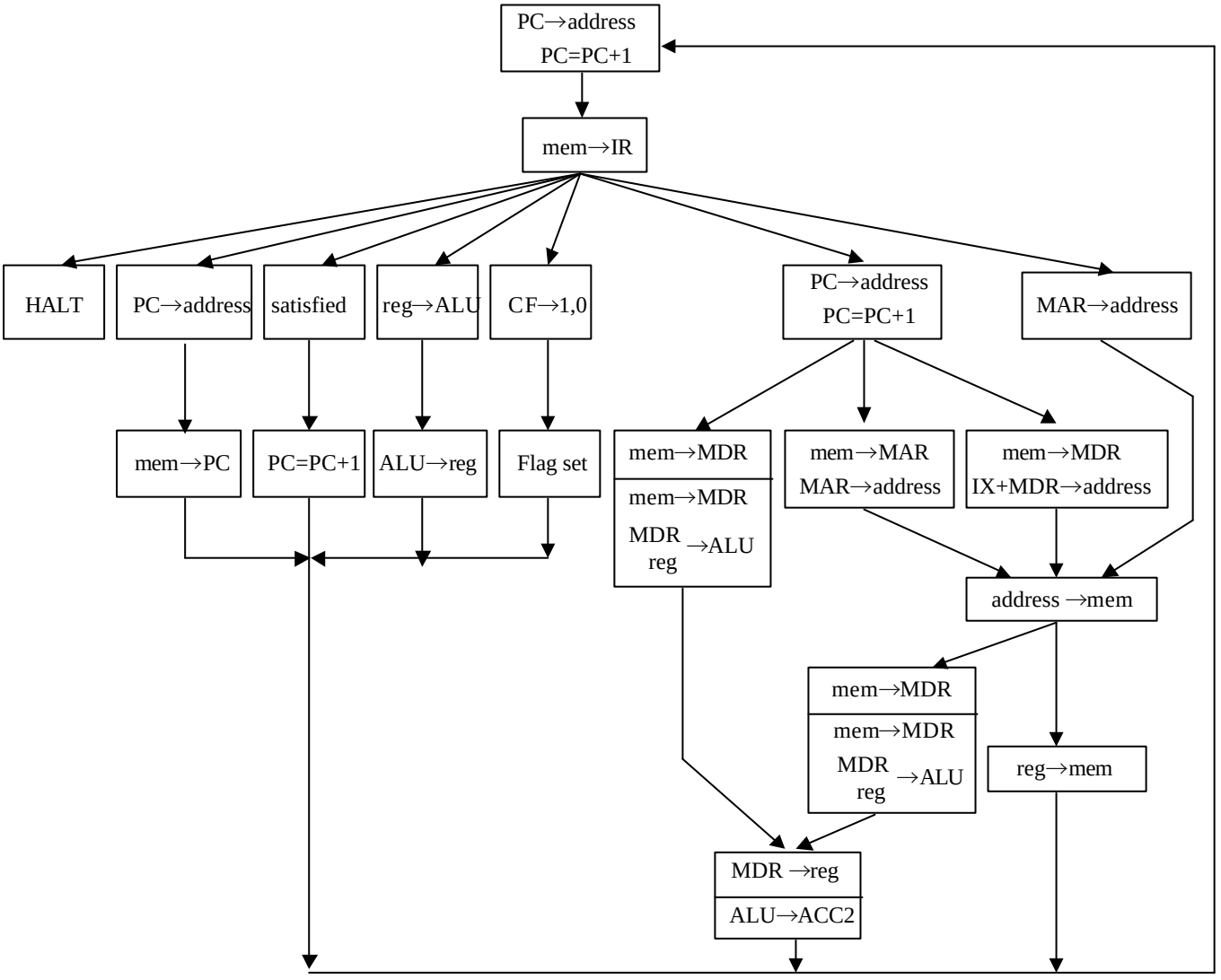


図4.42　状態遷移図

状態遷移はIRに取り込まれた命令あるいはアドレッシングモードによって上図のいずれかの経路をたどる。

HDLで記述するため各状態に状態名を割り当てる。その図を図4.43に示す。

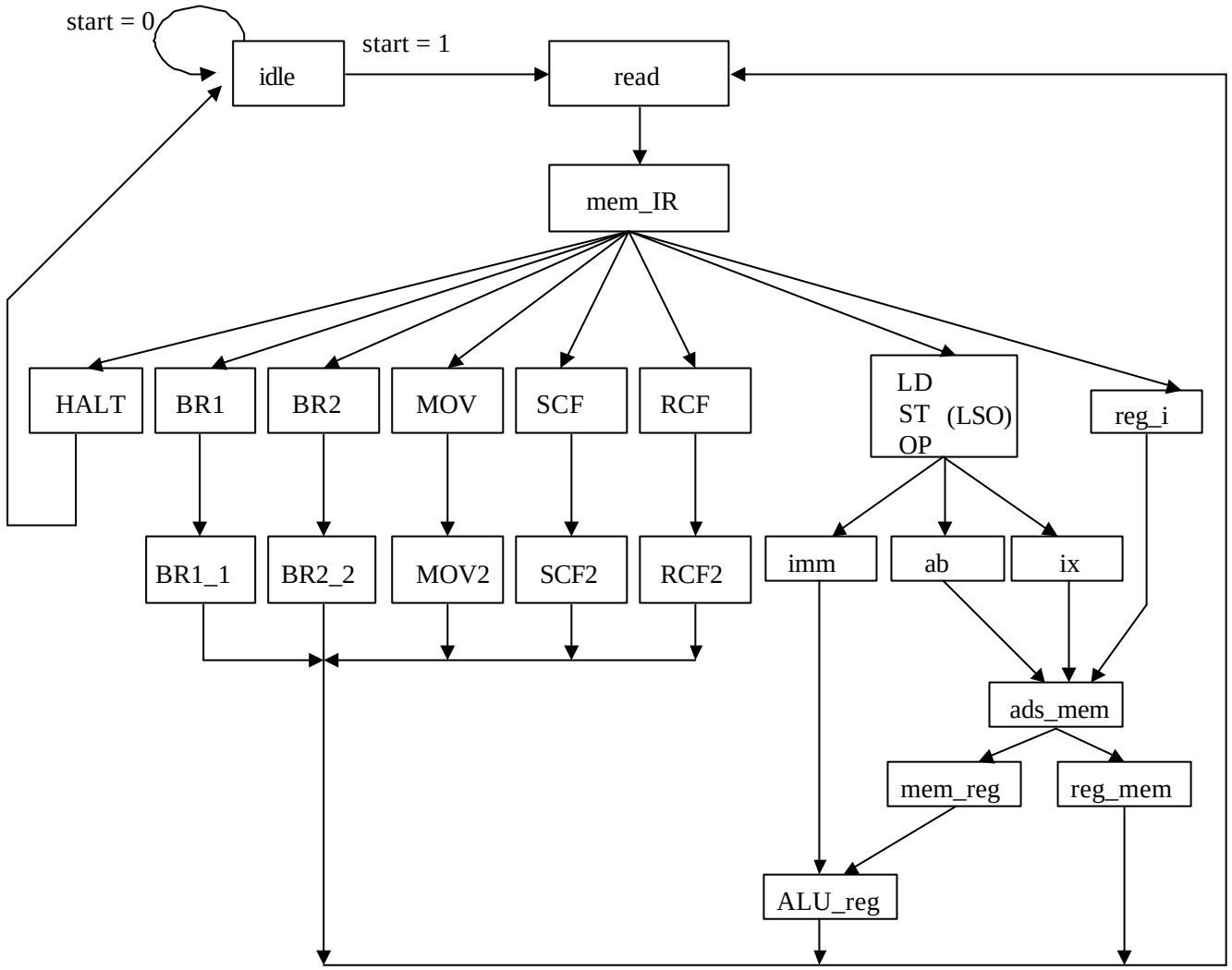


図4.43　状態の割り当て

プログラム終了後テストを行うが、プログラムを停止するには、停止命令を使う。その後テストしたい状態を保持しておかなくてはならない。そこで図 4.43 **の** HALT **の状態（停止した状態）の次に、アイドル状態（**idle**）を加える。この状態は、外部からの入力（**start**）がない限り現在の状態を保持する。**

4.3　HDL 記述

4.3.1　各ブロックの信号の定義

仕様をもとに HDL で記述し、シミュレーションする。HDL は各ブロック（ALU、ALU マルチプレクサ、制御部、アドレス生成部、各種レジスタ）で記述した後、ブロックを統合する。

（1）　ALU マルチプレクサ

ALU マルチプレクサの信号線の定義を図 4.44、表 4.8、表 4.9 に示す。図の矢印の太さはビット幅を表し、表 4.7 は図中の信号の簡単な説明であり、表 4.9 はレジスタ選択信号（cont1、2、test）のビットの割り当てである。

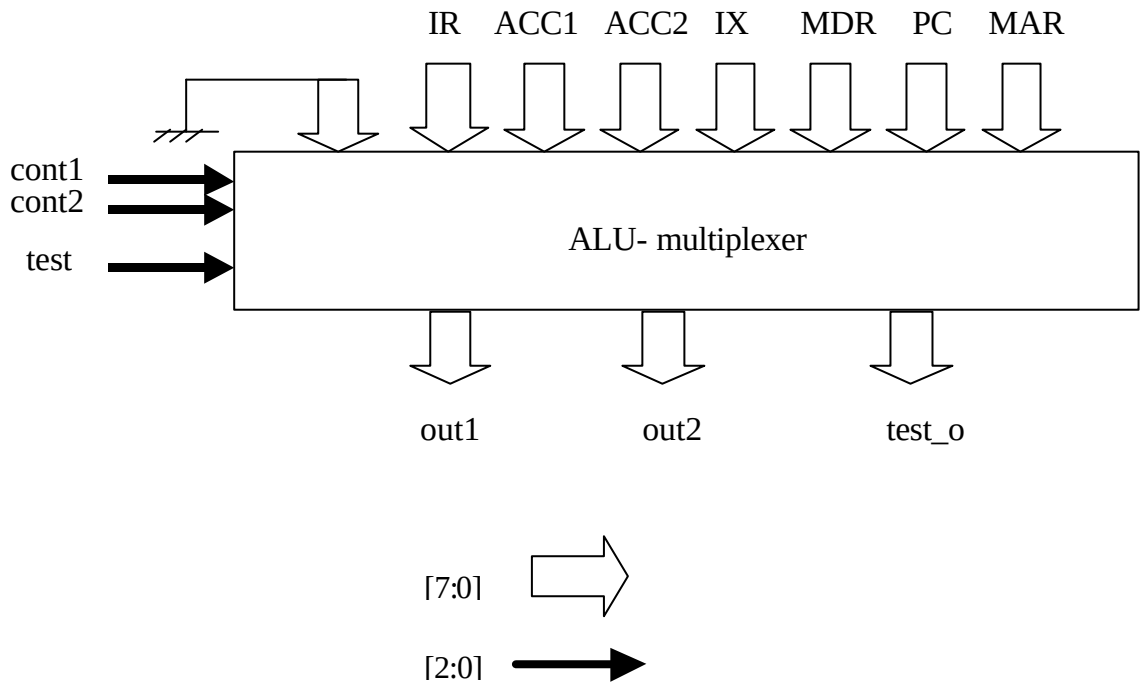


図4.44　ALUマルチプレクサの入出力信号

表4.8　ALUマルチプレクサの信号の定義

ir	IRからの出力信号
acc1	ACC1からの出力信号
acc2	ACC2からの出力信号
ix	IXからの出力信号
mdr	MDRからの出力信号
pc	PCからの出力信号
mar	MARからの出力信号
cont1	out1へ出力する制御部からの選択信号
cont2	out2へ出力する制御部からの選択信号
test	test_o　へ出力する制御部からの選択信号
out1	ALUへ入力される信号
out2	ALUへ入力される信号
test_o	テスト出力信号

表4.9　レジスタ選択信号の割り当て

cont1,2

ACC1	000
ACC2	001
IX	010
MAR	011
MDR	100
GND	101

test

IR	000
ACC1	001
ACC2	010
IX	011
MDR	100
PC	101
MAR	110

（2）　ALU

ALU の信号線の定義を図 4.45、表 4.10、表 4.11 に示す。図の矢印の太さはビット幅を表し、表 4.9 は図中の信号の簡単な説明であり、表 4.11 は演算制御信号（alu_cont）のビットの割り当てである。

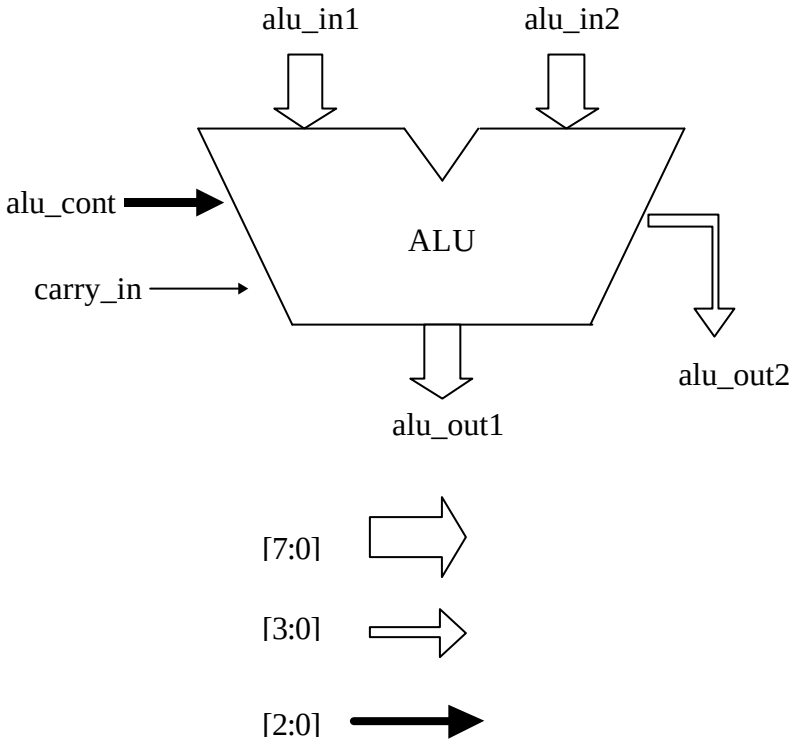


図4.45　ALUの入出力信号

表4.10　ALUの信号の定義

alu_in1	ALUへの入力信号
alu_in2	ALUへの入力信号
alu_cont	演算の種類を指定する制御信号
carry_in	キャリー入力
alu_out1	ALUの出力
alu_out2	演算結果の状態（フラグ）の出力

表4.11　演算制御信号の割り当て

	alu_cont
AND	000
OR	001
XOR	010
COMP	011
ADD	100
ADC	101
SUB	110
SUC	111

（3） **制御部**

制御部の信号線の定義を図 4.46**、表** 4.12**、表** 4.13 **に示す。図の矢印の太さはビット幅を表し、表** 4.12 **は図中の信号の簡単な説明であり、表** 4.13 **はフラグとアドレス選択回路の制御信号（**flag_s**、**a_mux**）と、**PC**、**MAR **の選択回路（**mux1**、**mux2**）の制御信号（**pc_s**、**mar_s**）のビットの割り当てである。**

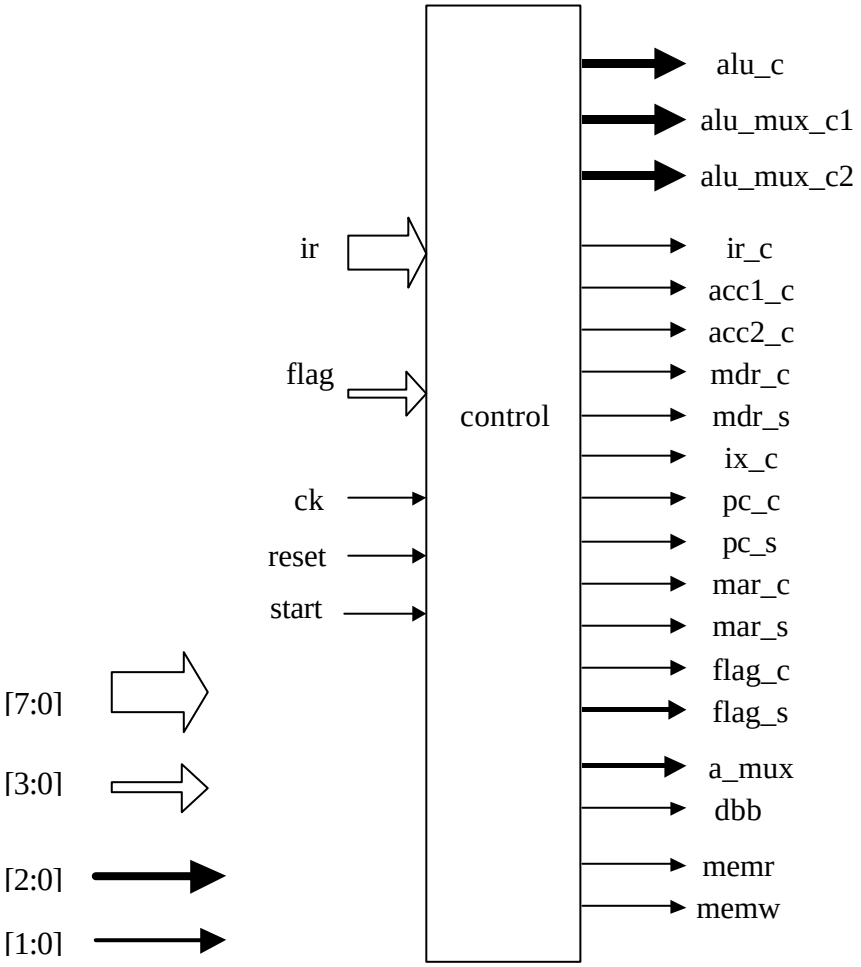


図4.46 制御部の入出力信号

表4.12　ALUの信号の定義

ir	IRからの信号
flag	Flagからの信号
ck	クロック
reset	リセット
start	スタート
alu_c	ALUへの演算制御信号
alu_mux_c1	ALUマルチプレクサの制御信号
alu_mux_c2	ALUマルチプレクサの制御信号
ir_c	IRへの制御信号、LOWレベルで取り込み
acc1_c	ACC1への制御信号、LOWレベルで取り込み
acc2_c	ACC2への制御信号、LOWレベルで取り込み
mdr_c	MDRへの制御信号、LOWレベルで取り込み
ix_c	IXへの制御信号、LOWレベルで取り込み
pc_c	PCへの制御信号、LOWレベルで取り込み
pc_s	PCへの制御信号、メモリデータとPCのデータを選択する
mar_c	MARへの制御信号、LOWレベルで取り込み
mar_s	MARへの制御信号、ALUの出力とメモリデータを選択
flag_c	Flagへの制御信号、LOWレベルで取り込み
flag_s	Flagへの制御信号、主にキャリーセットの際使用
a_mux	アドレス選択回路の制御信号
dbb	データバスバッファの制御信号
memr	メモリ・リード
memw	メモリ・ライト

表4.13　フラグ、アドレス選択回路と、PC、MARの制御信号の割り当て

flag_s

通常	00
SCF	01
RCF	10
--	11

a_mux

即値アドレス（PC）	00
絶対アドレス、レジスタ間接アドレス（MAR）	01
インデックス修飾アドレス（MDR+IX）	10
--	11

pc_s

data bus	0
PC	1

mar_s

ALU bus	0
data bus	1

pc_s の信号は、“ 0 ”のとき、内部データバスの信号を選択し、“ 1 ”のとき 1 増加した PC のデータを選択する。選択された信号は PC へ入力される。mar_s **は“ 0 ”のとき ALU バスの信号を選択し、“ 1 ”のとき内部データバスの信号を選択する。選択された信号は MAR に入力される。**

（4） アドレス生成部

アドレス生成部の信号線の定義を図 4.47、表 4.14 に示す。図の矢印の太さはビット幅を表し、表 4.14 は図中の信号の簡単な説明である。

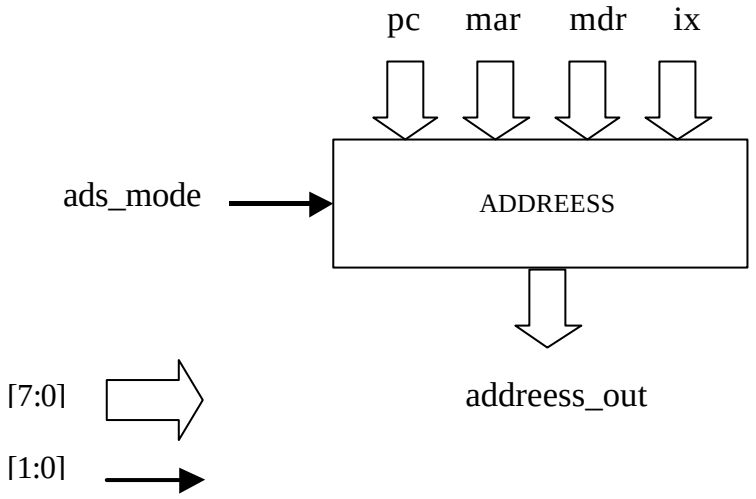


図4.47 アドレス生成部の入出力信号

表4.14 アドレス生成部の信号の定義

pc	PCの出力
mar	MARの出力
mdr	MDRの出力
ix	IXの出力
ads_mode	アドレスバスへの出力を選択
address_out	アドレスバスへの出力

（5） プログラムカウンタ（PC）

PC の信号線の定義を図 4.48、表 4.15 に示す。図の矢印の太さはビット幅を表し、表 4.15 は図中の信号の簡単な説明である。

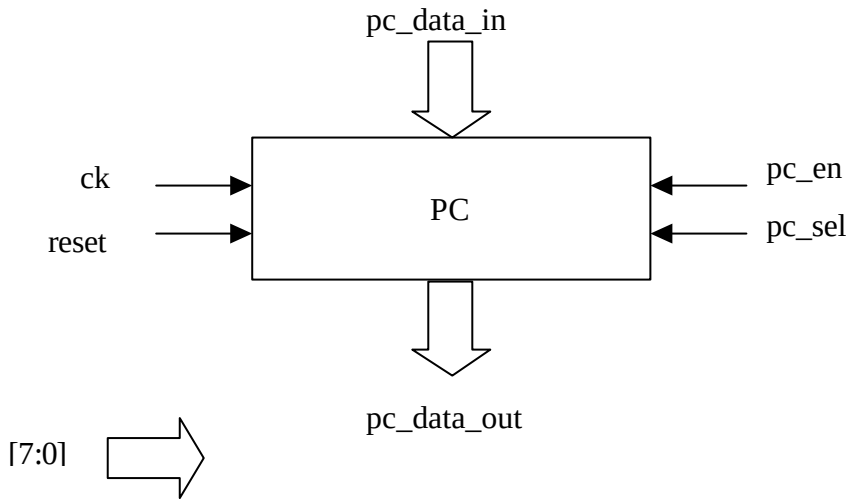


図4.48 PCの入出力信号

表4.15 PCの信号の定義

pc_data_in	内部データバスからのデータ
pc_en	データを取り込むためのイネーブル信号
pc_sel	内部データバスとインクリメントされたPCのデータの選択信号
pc_data_out	PCの出力信号

（6） メモリアドレスレジスタ（MAR）

MAR の信号線の定義を図 4.49、表 4.16 に示す。図の矢印の太さはビット幅を表し、表 4.16 は図中の信号の簡単な説明である。

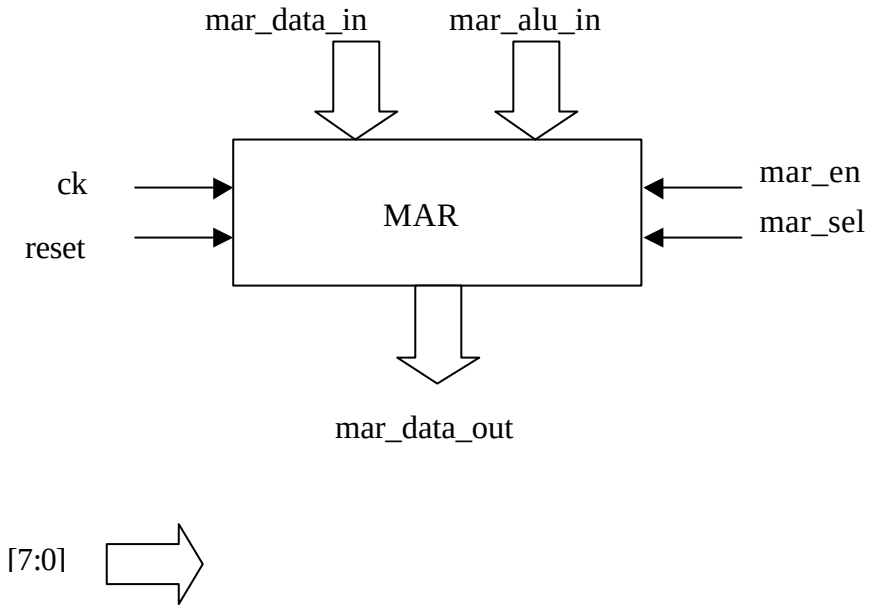


図4.49 MARの入出力信号

表4.16 MARの信号の定義

mar_alu_in	ALUバスからのデータ
mar_data_in	内部データバスからのデータ
mar_en	データを取り込むためのイネーブル信号
mar_sel	内部データバスとALUバスの選択信号
mar_data_out	MARの出力信号

（7） フラグレジスタ（Flag）

Flag の信号線の定義を図 4.50、表 4.17 に示す。図の矢印の太さはビット幅を表し、表 4.17 は図中の信号の簡単な説明である。

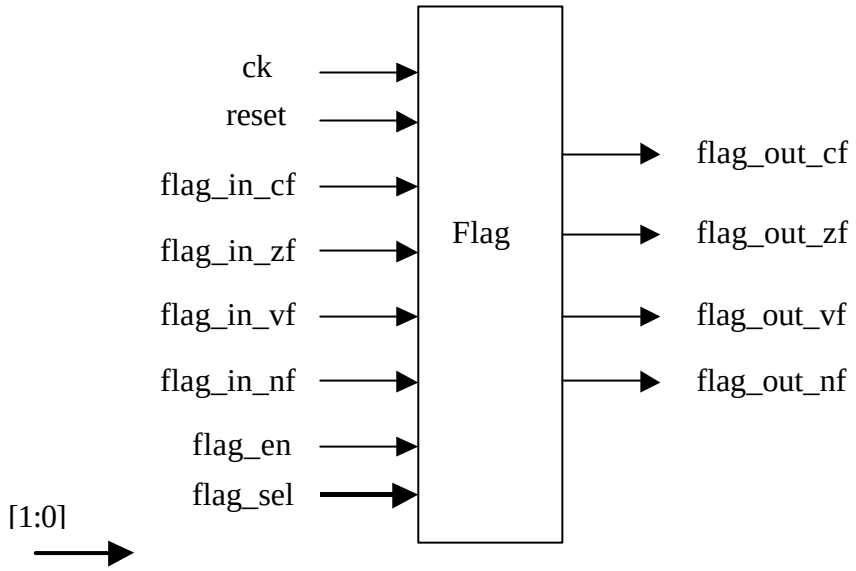


図4.50 Flag の入出力信号

表4.17 Flag の信号の定義

flag_en	データを取り込むイネーブル信号
flag_sel	ALUからの信号とキャリーフラグのセット（リセット）を選択
flag_in_cf	キャリーフラグの入力
flag_in_zf	ゼロフラグの入力
flag_in_vf	オーバーフローフラグの入力
flag_in_nf	ネガティブフラグの入力
flag_out_cf	キャリーフラグの出力
flag_out_zf	ゼロフラグの出力
flag_out_vf	オーバーフラグフラグの出力
flag_out_nf	ネガティブフラグの出力

表4.18　flag_sel の制御信号の割り当て

	flag_sel
ALUからの状態を選択	00
CFをセットする場合	01
CFをリセットする場合	10
― ―	11

（8） **レジスタ群**（IR、ACC1、ACC2、MDR）

IR の信号線の定義を図 4.51**、表** 4.19 **に示す。図の矢印の太さはビット幅を表し、表** 4.18 **は図中の信号の簡単な説明である。**ACC1、ACC2、MDR **についても、信号名が違うだけで動作及び** VHDL **の記述は同じである。**

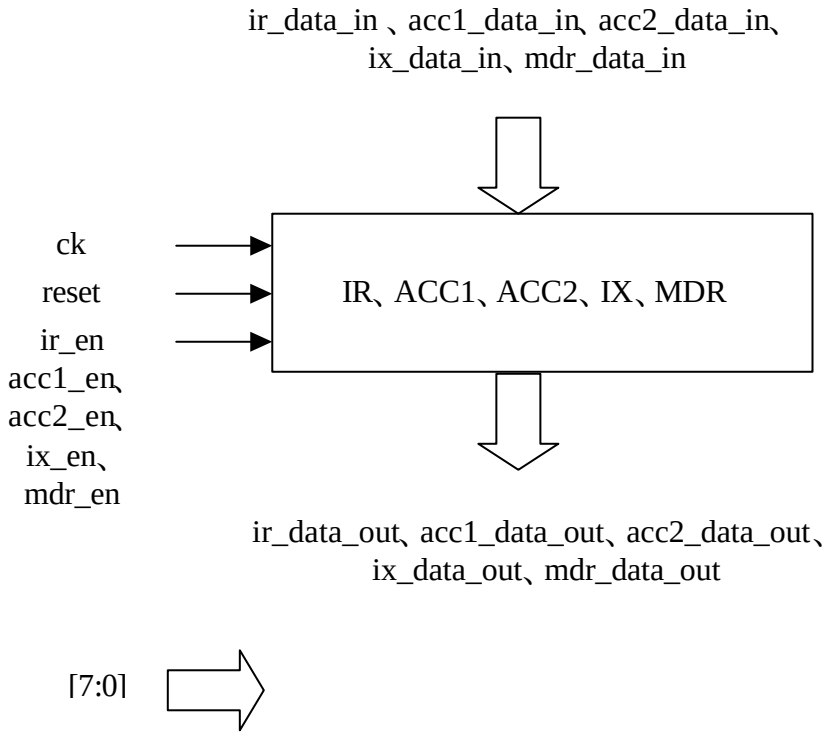


図4.51　レジスタ群の入出力信号

表4.19　レジスタ群の信号の定義

ir_en 、acc1_en、acc2_en、 ix_en、mdr_en	データを取り込むイネーブル信号
ir_data_in、acc1_data_in、 acc2_data_in、ix_data_in、 mdr_data_in	ALUバスからのデータ
ir_data_out、acc1_data_out、 acc2_data_out、ix_data_out、 mdr_data_out	出力信号

4.3.2 VHDL ソースコード

リスト 1 **に** ALU **マルチプレクサ、リスト** 2 **に** ALU**、リスト** 3 **に制御部、リスト** 4 **にアドレス生成部、リスト** 5 **に** PC**、リスト** 6 **に** MAR**、リスト** 7 **に** Flag**、リスト** 8 **にレジスタ群（**IR **のみ）、リスト** 9 **に各ブロックを接続した最上位ブロックの** VHDL **のソースコードを示す。**

リスト 1　ALU **マルチプレクサ**

```
-- ALU_MULTIPLEXER
library ieee;
use ieee.std_logic_1164.all;
entity ALU_mux is
    port (
        ir_o : in std_logic_vector(7 downto 0);
        acc1_o: in std_logic_vector(7 downto 0);
        acc2_o: in std_logic_vector(7 downto 0);
        ix_o: in std_logic_vector(7 downto 0);
        mdr_o: in std_logic_vector(7 downto 0);
        pc_o: in std_logic_vector(7 downto 0);
        mar_o: in std_logic_vector(7 downto 0);
        cont1 : in std_logic_vector(2 downto 0);
        cont2 : in std_logic_vector(2 downto 0);
```

```
        alu_mux_o1 : out std_logic_vector(7 downto 0);
        alu_mux_o2 : out std_logic_vector(7 downto 0);
         test_sel : in std_logic_vector(2 downto 0);
         test_o : out std_logic_vector(7 downto 0)
    );
end ALU_mux;
architecture BEHAVIOR of ALU_mux is
begin
 process(cont1,cont2,test_sel,ir_o,acc1_o,acc2_o,ix_o,mdr_o,
         pc_o,mar_o)
  begin
  case cont1 is
   when "000" => alu_mux_o1 <= acc1_o;
   when "001" => alu_mux_o1 <= acc2_o;
   when "010" => alu_mux_o1 <= ix_o;
   when "011" => alu_mux_o1 <= mar_o;
   when "100" => alu_mux_o1 <= mdr_o;
   when "101" => alu_mux_o1 <= "00000000";
   when others  => alu_mux_o1 <= "XXXXXXXX";
  end case;
end process;
 process(cont1,cont2,test_sel,ir_o,acc1_o,acc2_o,ix_o,mdr_o,
         pc_o,mar_o)
  begin
  case cont2 is
   when   "000" => alu_mux_o2 <= acc1_o;
   when   "001" => alu_mux_o2 <= acc2_o;
   when   "010" => alu_mux_o2 <= ix_o;
   when   "011" => alu_mux_o2 <= mar_o;
   when   "100" => alu_mux_o2 <= mdr_o;
   when   "101" => alu_mux_o2 <= "00000000";
  when others
     => alu_mux_o2 <= "XXXXXXXX";
  end case;
end process;
 process (cont1,cont2,test_sel,ir_o,acc1_o,acc2_o,ix_o,mdr_o,
```

```
        pc_o,mar_o)
  begin
    case test_sel is
   when  "000" => test_o <= ir_o;
   when  "001" => test_o <= acc1_o;
   when  "010" => test_o <= acc2_o;
   when  "011" => test_o <= ix_o;
   when  "100" => test_o <= mdr_o;
   when  "101" => test_o <= pc_o;
   when  "110" => test_o <= mar_o;
   when others
     => test_o <= "XXXXXXXX";
     end case;
end process;
end BEHAVIOR ;
```

リスト 2　ALU

```
-- ALU
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ALU is
    port (alu_in1 :in std_logic_vector(7 downto 0);
          alu_in2 :in std_logic_vector(7 downto 0);
          alu_cont :in std_logic_vector(2 downto 0);
          alu_cin : in std_logic;

          alu_out :out std_logic_vector(7 downto 0);
          cf : out std_logic;
          zf : out std_logic;
          vf : out std_logic;
          nf : out std_logic
);
```

```
end ALU;
architecture BEHAVIOR of ALU is
 signal tmp : std_logic_vector(8 downto 0);
begin
   alu_out <= tmp(7 downto 0);
  process(alu_cont,alu_in1,alu_in2,alu_cin,alu_out)
   begin
   case alu_cont is
--AND
    when "000" =>
      tmp <= ('0' & alu_in1) AND('0' & alu_in2) ;
      vf <= '0';
 -- OR
    when "001" =>
      tmp <= ('0' & alu_in1) OR ('0' & alu_in2) ;
      vf <= '0';
-- XOR
    when "010" =>
      tmp <= ('0' & alu_in1) XOR ('0' & alu_in2) ;
      vf <= '0';
-- COMP
    when  "011" =>
       tmp <= ('0' & alu_in1) - ('0' & alu_in2);
     if tmp(8) = '1' then
       nf <= '1';
     else
       nf <= '0';
     end if;
-- ADD
    when "100" =>
      tmp <= ('0' & alu_in1) + ('0' & alu_in2);
     if tmp(8) = '1' then
      vf <= '1';
     else
      vf <= '0';
     end if;
```

```
-- ADC
    when "101" =>
      tmp <= ('0' & alu_in1) +('0' & alu_in2) + ("00000000" & alu_cin);
     if tmp(8) = '1' then
      cf <= '1' ;
     else
      cf <= '0';
     end if;
-- SUB
    when "110"  =>
      tmp <= ('0' & alu_in1) - ('0' & alu_in2);
     if tmp(8) = '1' then
       vf <= '1';
     else
       vf <= '0';
     end if;
-- SUC
    when others =>
      tmp <= ('0' & alu_in1) - ('0' & alu_in2) -("00000000" & alu_cin);
     if tmp(8) = '1' then
       cf <= '1';
     else
       cf <= '0';
     end if;
end case;
end process;
-- all zero
process(alu_cont,alu_in1,alu_in2,alu_out)
 begin
 if tmp = "000000000" then
  zf <= '1';
 else
  zf <= '0';
 end if;
end process;
end BEHAVIOR;
```

リスト 3　制御部

```
-- CONTROL
library ieee;
use ieee.std_logic_1164.all;
entity CONT is
    port (
        ck: in std_logic;
        reset: in std_logic;
        start: in std_logic;
        opcode: in std_logic_vector(3 downto 0);
        ads_mode: in std_logic_vector(3 downto 0);
        cf,zf,vf,nf: in std_logic;
        alu_c: out std_logic_vector(2 downto 0);
        alu_mux_c1: out std_logic_vector(2 downto 0);
        alu_mux_c2: out std_logic_vector(2 downto 0);
        ir_c: out std_logic;
        acc1_c: out std_logic;
        acc2_c: out std_logic;
        ix_c: out std_logic;
        pc_c: out std_logic;
        pc_s: out std_logic;
        mar_c: out std_logic;
        mar_s: out std_logic;
        mdr_c: out std_logic;
        ads_mux: out std_logic_vector(1 downto 0);
        dbb_c: out std_logic;
        flag_c: out std_logic;
        flag_s: out std_logic_vector(1 downto 0);
        memr: out std_logic;
        memw: out std_logic
    );
end CONT;
architecture RTL of CONT is
 type state is (idle,read,dec,HALT,BR1,BR1_1,BR2,BR2_2,MOV,SRCF,LSO,
```

```
                MOV2,SRCF2,imm,ab,ix,ads_mem,reg_i,
                ALU_reg,mem_reg,reg_mem
              );
 signal current_state, next_state : state;
begin
 process(ck,reset)
  begin
   if(reset = '0') then
    current_state <= idle;
   elsif(ck'event and ck = '1') then
    current_state <= next_state;
   end if;
 end process;
 process(ck,reset,opcode,ads_mode,start,current_state)
  begin
   case current_state is

    when idle =>
       if(start = '0')then
        next_state <= read;
       else
        next_state <= idle;
       end if;
    when read => next_state <= dec;
    when dec =>
       if (opcode = "0000") then            -- halt
         next_state <= HALT ;
       elsif(opcode = "0001") then           -- load
           if (ads_mode(1 downto 0) = "11" )then
            next_state <= reg_i;
           else
            next_state <= LSO;
           end if;
       elsif(opcode = "0010") then            -- store
          if (ads_mode(1 downto 0) = "11" )then
           next_state <= reg_i;
```

```
   elsif (ads_mode(1 downto 0) = "00")then
    next_state <= idle;
   else
    next_state <= LSO;
   end if;
elsif(opcode = "0101") then -- move
   next_state <= MOV;
elsif(opcode = "0011") then -- set carry flag
  next_state <= SRCF;
elsif(opcode = "0100") then -- reset carry flag
  next_state <= SRCF;
elsif(opcode = "0110") then -- branch
  if (ads_mode = "0000")then -- (always)
    next_state <= BR1;
  elsif (ads_mode = "1000")then --(on carry)
     if (cf = '1')then
       next_state <= BR2;
     else
       next_state <= BR1;
     end if;
  elsif (ads_mode = "0100")then -- (on zero flag)
     if (zf = '1')then
       next_state <= BR2;
     else
       next_state <= BR1;
     end if;
  elsif (ads_mode = "0010")then -- (on overflow flag)
     if (vf = '1')then
       next_state <= BR2;
     else
       next_state <= BR1;
     end if;
  elsif (ads_mode = "0001")then -- (on negative flag)
     if (nf = '1')then
       next_state <= BR2;
     else
```

```
      next_state <= BR1;
    end if;
  end if;
elsif(opcode = "0111") then                    -- and
    if (ads_mode(1 downto 0) = "11" )then
     next_state <= reg_i;
    else
     next_state <= LSO;
    end if;
elsif(opcode = "1000") then                     -- or
    if (ads_mode(1 downto 0) = "11" )then
     next_state <= reg_i;
    else
     next_state <= LSO;
    end if;
elsif(opcode = "1001") then                     -- xor
    if (ads_mode(1 downto 0) = "11" )then
     next_state <= reg_i;
    else
     next_state <= LSO;
    end if;
elsif(opcode = "1010") then                    -- cmpare
    if (ads_mode(1 downto 0) = "11" )then
     next_state <= reg_i;
    else
     next_state <= LSO;
    end if;
elsif(opcode = "1011") then                     -- add
    if (ads_mode(1 downto 0) = "11" )then
     next_state <= reg_i;
    else
     next_state <= LSO;
    end if;
elsif(opcode = "1100") then                     -- adc
    if (ads_mode(1 downto 0) = "11" )then
     next_state <= reg_i;
```

```
        else
         next_state <= LSO;
        end if;
    elsif(opcode = "1101") then                          -- sub
        if (ads_mode(1 downto 0) = "11" )then
         next_state <= reg_i;
        else
         next_state <= LSO;
        end if;
    elsif(opcode = "1110") then                          -- suc
        if (ads_mode(1 downto 0) = "11" )then
         next_state <= reg_i;
        else
         next_state <= LSO;
        end if;
    else
      next_state <= idle;
    end if;
when BR1 =>  next_state <= BR1_1;
when BR2 =>  next_state <= BR2_2;
when BR1_1 => next_state <= read;
when BR2_2 =>  next_state <= read;
when MOV =>  next_state <= MOV2;
when MOV2 => next_state <= read;
when SRCF => next_state <= SRCF2;
when SRCF2 => next_state <= read;
when HALT => next_state <= idle;
when LSO =>
    if (ads_mode(1 downto 0 ) = "00") then
     next_state <= imm;
    elsif (ads_mode(1 downto 0) = "01")then
     next_state <= ab;
    elsif (ads_mode(1 downto 0) = "10")then
     next_state <= ix;
    end if;
when imm =>  next_state <= alu_reg;
```

```
    when alu_reg =>  next_state <= read;
    when ab =>  next_state <= ads_mem;
    when ix =>  next_state <= ads_mem;
    when  ads_mem  =>
       if (opcode = "0010")then
         next_state <= reg_mem;
       else
         next_state <= mem_reg;
       end if;
    when mem_reg =>  next_state <= alu_reg;
    when alu_reg =>  next_state <= read;
    when reg_i =>  next_state <= ads_mem;
    when reg_mem => next_state <= read;
    when others =>  next_state <= idle;
  end case;
end process;
process(current_state)            -- memr,memw,mdr_c
 begin
  if (current_state = read)then
   memr <= '0'; memw <= '1'; mdr_c <= '1';
  elsif(current_state = LSO)then
   memr <= '0'; memw <= '1';  mdr_c <= '0';
  elsif(current_state = BR1)then
   memr <= '0'; memw <= '1'; mdr_c <= '1';
  elsif (current_state = ads_mem)then
   if (opcode = "0010")then
    memr <= '1'; memw <= '0'; mdr_c <= '1';
   else
    memr <= '0';memw <= '1';  mdr_c <= '0';
   end if;
  else
   memr <= '1'; memw <= '1'; mdr_c <= '1';
  end if;
end process;
process(current_state)                -- ir_c
 begin
```

```
  if (current_state = read )then
   ir_c <= '0';
  else
   ir_c <= '1';
  end if;
end process;
process(current_state)        -- acc1_c,acc2_c,ix_c,pc_c, mar_c
 begin
 if(current_state = idle)then
   acc1_c <= '1'; acc2_c <= '1';ix_c <= '1'; pc_c <= '1'; mar_c <= '1';
 elsif (current_state = read)then
   acc1_c <= '1'; acc2_c <= '1';ix_c <= '1'; pc_c <= '0'; mar_c <= '1';
 elsif (current_state = imm )then
   if(opcode = "0010")then
    acc1_c <= '1';acc2_c <= '1'; ix_c <= '1'; pc_c <= '1'; mar_c <= '1';
   elsif (opcode = "0001")then
      if (ads_mode(3 downto 2) = "00")then
       acc1_c <= '0'; acc2_c <= '1';ix_c <= '1'; pc_c <= '1'; mar_c <='1';
      elsif (ads_mode(3 downto 2) = "01")then
       acc1_c <= '1'; acc2_c <= '0';ix_c <= '1'; pc_c <= '1'; mar_c <='1';
      elsif (ads_mode(3 downto 2) = "10")then
       acc1_c <= '1';acc2_c <= '1'; ix_c <= '0'; pc_c <= '1'; mar_c <='1';
      else
       acc1_c <= '1'; acc2_c <= '1';ix_c <= '1'; pc_c <= '1'; mar_c <='0';
      end if;
   else
     acc1_c <= '1'; acc2_c <= '0';ix_c <= '1'; pc_c <= '1'; mar_c <='1';
   end if;
 elsif (current_state = mem_reg )then
   if(opcode = "0010")then
     acc1_c <= '1'; acc2_c <= '1';ix_c <= '1'; pc_c <= '1'; mar_c <='1';
   elsif(opcode = "0001")then
     if (ads_mode(3 downto 2) = "00")then
      acc1_c <= '0'; acc2_c <= '1';ix_c <= '1'; pc_c <= '1'; mar_c <='1';
     elsif (ads_mode(3 downto 2) = "01")then
      acc1_c <= '1'; acc2_c <= '0';ix_c <= '1'; pc_c <= '1'; mar_c <='1';
```

```
    elsif (ads_mode(3 downto 2) = "10")then
     acc1_c <= '1'; acc2_c <= '1'; ix_c <= '0'; pc_c <= '1'; mar_c <='1';
    else
     acc1_c <= '1'; acc2_c <= '1';ix_c <= '1'; pc_c <= '1'; mar_c <='0';
    end if;
   else
    acc1_c <= '1'; acc2_c <= '0';ix_c <= '1'; pc_c <= '1'; mar_c <='1';
  end if;
elsif (current_state = LSO )then
  if(ads_mode(1 downto 0) = "01")then
     acc1_c <= '1';acc2_c <= '1'; ix_c <= '1'; pc_c <= '0'; mar_c <='0';
  else
     acc1_c <= '1';acc2_c <= '1'; ix_c <= '1'; pc_c <= '0'; mar_c <='1';
  end if;
elsif (current_state = BR1 )then
     acc1_c <= '1'; acc2_c <= '1';ix_c <= '1'; pc_c <= '0'; mar_c <='1';
elsif (current_state = BR2 )then
     acc1_c <= '1'; acc2_c <= '1';ix_c <= '1'; pc_c <= '0'; mar_c <='1';
elsif (current_state = MOV)then
  if (ads_mode(1 downto 0) = "00")then
   acc1_c <= '0'; acc2_c <= '1';ix_c <= '1'; pc_c <= '1'; mar_c <= '1';
  elsif (ads_mode(1 downto 0) = "01")then
   acc1_c <= '1'; acc2_c <= '0';ix_c <= '1'; pc_c <= '1'; mar_c <= '1';
  elsif (ads_mode(1 downto 0) = "10")then
   acc1_c <= '1';acc2_c <= '1'; ix_c <= '0'; pc_c <= '1'; mar_c <= '1';
  elsif (ads_mode(1 downto 0) = "11")then
   acc1_c <= '1'; acc2_c <= '1';ix_c <= '1'; pc_c <= '1'; mar_c <= '0';
  else
   acc1_c <= '1';acc2_c <= '1'; ix_c <= '1'; pc_c <= '1'; mar_c <= '1';
  end if;
else
  acc1_c <= '1'; acc2_c <= '1';ix_c <= '1'; pc_c <= '1'; mar_c <= '1';
 end if;
end process;
process  (current_state )          -- pc_s
 begin
```

```
  if (current_state = read )then
   pc_s <= '1';
  elsif(current_state = LSO )then
   pc_s <= '1';
  elsif (current_state = BR2 )then
   pc_s <= '1';
  else
   pc_s <= '0';
  end if;
end process;
process(current_state)              -- mar_s
 begin
  if(current_state = LSO)then
   mar_s <= '1';
  elsif(current_state = ab)then
   mar_s <= '1';
  else
   mar_s <= '0';
  end if;
end process;

process(current_state)             -- address_mux
 begin
  if (current_state = ab)then
   ads_mux <= "01";
  elsif(current_state = reg_i)then
   ads_mux <= "01";
  elsif(current_state = ix)then
   ads_mux <= "10";
  elsif(current_state = ALU_reg) then
   ads_mux <= "00";
  elsif(current_state = reg_mem) then
   ads_mux <= "00";
  elsif(current_state = idle) then
   ads_mux <= "00";
  end if;
```

```
end process;
process(current_state)            -- dbb_c
 begin
  if (current_state = ads_mem)then
   if(opcode = "0010")then
    dbb_c <= '1';
   else
    dbb_c <= '0';
   end if;
  elsif(current_state = reg_mem)then
   if(opcode = "0010")then
    dbb_c <= '1';
   else
    dbb_c <= '0';
   end if;
 else
  dbb_c <= '0';
 end if;
end process;
process(current_state)          -- flag_c,flag_s
 begin
 if(current_state = SRCF)then
   if(opcode = "0011")then
    flag_s <= "01";
    flag_c <= '0';
   elsif(opcode = "0100")then
    flag_s <= "10";
    flag_c <= '0';
   else
     flag_c <= '1';
   end if;
 elsif (current_state = mem_reg)then
   if (opcode > "0110")then
    flag_s <= "00";
    flag_c <= '0';
   else
```

```
    flag_s <= "00";
    flag_c <= '1';
   end if;
 elsif (current_state = imm)then
   if (opcode > "0110")then
    flag_s <= "00";
    flag_c <= '0';
   else
    flag_s <= "00";
    flag_c <= '1';
   end if;
 else
    flag_s <= "00";
    flag_c <= '1';
 end if;
end process;
process (current_state)       -- alu_mux_c1,c2,alu_c
 begin
  if(opcode = "0001")then     -- load
   alu_mux_c1 <= "101";
   alu_mux_c2 <= "100";
   alu_c <= "001";
  elsif(opcode = "0010")then              -- st
    if(ads_mode(3 downto 2) = "00") then
         alu_mux_c1 <= "101";
         alu_mux_c2 <= "000";
         alu_c <= "001";
    elsif(ads_mode(3 downto 2) = "01") then
         alu_mux_c1 <= "101";
         alu_mux_c2 <= "001";
         alu_c <= "001";
    elsif(ads_mode(3 downto 2) = "10") then
         alu_mux_c1 <= "101";
         alu_mux_c2 <= "010";
         alu_c <= "001";
    elsif(ads_mode(3 downto 2) = "11") then
```

```
            alu_mux_c1 <= "101";
            alu_mux_c2 <= "011";
            alu_c <= "001";
        end if;
    elsif(opcode = "0101")then              -- MOV
        if(ads_mode(3 downto 2) = "00") then
            alu_mux_c1 <= "101";
            alu_mux_c2 <= "000";
            alu_c <= "001";
        elsif(ads_mode(3 downto 2) = "01") then
            alu_mux_c1 <= "101";
            alu_mux_c2 <= "001";
            alu_c <= "001";
        elsif(ads_mode(3 downto 2) = "10") then
            alu_mux_c1 <= "101";
            alu_mux_c2 <= "010";
            alu_c <= "001";
        elsif(ads_mode(3 downto 2) = "11") then
            alu_mux_c1 <= "101";
            alu_mux_c2 <= "011";
            alu_c <= "001";
        end if;
    elsif(opcode = "0111")then                  -- and
        if(ads_mode(3 downto 2) = "00") then
            alu_mux_c1 <= "000";
            alu_mux_c2 <= "100";
            alu_c <= "000";
        elsif(ads_mode(3 downto 2) = "01") then
            alu_mux_c1 <= "001";
            alu_mux_c2 <= "100";
            alu_c <= "000";
        elsif(ads_mode(3 downto 2) = "10") then
            alu_mux_c1 <= "010";
            alu_mux_c2 <= "100";
            alu_c <= "000";
        elsif(ads_mode(3 downto 2) = "11") then
```

```
            alu_mux_c1 <= "011";
            alu_mux_c2 <= "100";
            alu_c <= "000";
        end if;
    elsif(opcode = "1000")then                  -- or
        if(ads_mode(3 downto 2) = "00") then
            alu_mux_c1 <= "000";
            alu_mux_c2 <= "100";
            alu_c <= "001";
        elsif(ads_mode(3 downto 2) = "01") then
            alu_mux_c1 <= "001";
            alu_mux_c2 <= "100";
            alu_c <= "001";
        elsif(ads_mode(3 downto 2) = "10") then
            alu_mux_c1 <= "010";
            alu_mux_c2 <= "100";
            alu_c <= "001";
        elsif(ads_mode(3 downto 2) = "11") then
            alu_mux_c1 <= "011";
            alu_mux_c2 <= "100";
            alu_c <= "001";
        end if;
    elsif(opcode = "1001")then                  -- xor
        if(ads_mode(3 downto 2) = "00") then
            alu_mux_c1 <= "000";
            alu_mux_c2 <= "100";
            alu_c <= "010";
        elsif(ads_mode(3 downto 2) = "01") then
            alu_mux_c1 <= "001";
            alu_mux_c2 <= "100";
            alu_c <= "010";
        elsif(ads_mode(3 downto 2) = "10") then
            alu_mux_c1 <= "010";
            alu_mux_c2 <= "100";
            alu_c <= "010";
        elsif(ads_mode(3 downto 2) = "11") then
```

```
        alu_mux_c1 <= "011";
        alu_mux_c2 <= "100";
        alu_c <= "010";
    end if;
elsif(opcode = "1010")then                  -- comp
    if(ads_mode(3 downto 2) = "00") then
        alu_mux_c1 <= "000";
        alu_mux_c2 <= "100";
        alu_c <= "011";
    elsif(ads_mode(3 downto 2) = "01") then
        alu_mux_c1 <= "001";
        alu_mux_c2 <= "100";
        alu_c <= "011";
    elsif(ads_mode(3 downto 2) = "10") then
        alu_mux_c1 <= "010";
        alu_mux_c2 <= "100";
        alu_c <= "011";
    elsif(ads_mode(3 downto 2) = "11") then
        alu_mux_c1 <= "011";
        alu_mux_c2 <= "100";
        alu_c <= "011";
    end if;
elsif(opcode = "1011")then
    if(ads_mode(3 downto 2) = "00") then        -- add
        alu_mux_c1 <= "000";
        alu_mux_c2 <= "100";
        alu_c <= "100";
    elsif(ads_mode(3 downto 2) = "01") then
        alu_mux_c1 <= "001";
        alu_mux_c2 <= "100";
        alu_c <= "100";
    elsif(ads_mode(3 downto 2) = "10") then
        alu_mux_c1 <= "010";
        alu_mux_c2 <= "100";
        alu_c <= "100";
    elsif(ads_mode(3 downto 2) = "11") then
```

```
            alu_mux_c1 <= "011";
            alu_mux_c2 <= "100";
            alu_c <= "100";
        end if;
    elsif(opcode = "1100")then                  -- adc
        if(ads_mode(3 downto 2) = "00") then
            alu_mux_c1 <= "000";
            alu_mux_c2 <= "100";
            alu_c <= "101";
        elsif(ads_mode(3 downto 2) = "01") then
            alu_mux_c1 <= "001";
            alu_mux_c2 <= "100";
            alu_c <= "101";
        elsif(ads_mode(3 downto 2) = "10") then
            alu_mux_c1 <= "010";
            alu_mux_c2 <= "100";
            alu_c <= "101";
        elsif(ads_mode(3 downto 2) = "11") then
            alu_mux_c1 <= "011";
            alu_mux_c2 <= "100";
            alu_c <= "101";
        end if;
    elsif(opcode = "1101")then                  -- sub
        if(ads_mode(3 downto 2) = "00") then
            alu_mux_c1 <= "000";
            alu_mux_c2 <= "100";
            alu_c <= "110";
        elsif(ads_mode(3 downto 2) = "01") then
            alu_mux_c1 <= "001";
            alu_mux_c2 <= "100";
            alu_c <= "110";
        elsif(ads_mode(3 downto 2) = "10") then
            alu_mux_c1 <= "010";
            alu_mux_c2 <= "100";
            alu_c <= "110";
        elsif(ads_mode(3 downto 2) = "11") then
```

```
            alu_mux_c1 <= "011";
            alu_mux_c2 <= "100";
            alu_c <= "110";
        end if;
    elsif(opcode = "1110")then                  -- suc
        if(ads_mode(3 downto 2) = "00") then
            alu_mux_c1 <= "000";
            alu_mux_c2 <= "100";
            alu_c <= "111";
        elsif(ads_mode(3 downto 2) = "01") then
            alu_mux_c1 <= "001";
            alu_mux_c2 <= "100";
            alu_c <= "111";
        elsif(ads_mode(3 downto 2) = "10") then
            alu_mux_c1 <= "010";
            alu_mux_c2 <= "100";
            alu_c <= "111";
        elsif(ads_mode(3 downto 2) = "11") then
            alu_mux_c1 <= "011";
            alu_mux_c2 <= "100";
            alu_c <= "111";
        end if;
    else
          alu_mux_c1 <= "XXX";
          alu_mux_c2 <= "XXX";
          alu_c <= "XXX";
    end if;
  end process;
end RTL;
```

リスト 4　アドレス生成部

```
-- address
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ADDRESS is
    port (
        ads_mode: in std_logic_vector(1 downto 0);
        pc: in std_logic_vector(7 downto 0);
        mar: in std_logic_vector(7 downto 0);
        mdr: in std_logic_vector(7 downto 0);
        ix: in std_logic_vector(7 downto 0);
        address_out: out std_logic_vector(7 downto 0)
    );
end ADDRESS;
architecture BEHAVIOR of ADDRESS is
begin
 process(ads_mode,pc,mar,mdr,ix)
  begin
   if (ads_mode = "00")then
    address_out <= pc;
   elsif(ads_mode = "10")then
    address_out <= mdr + ix;
   elsif(ads_mode = "01")then
    address_out <= mar;
   else
    address_out <= "XXXXXXXX" ;
   end if;
 end process;
end BEHAVIOR;
```

リスト5　PC

```
-- pc
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity PC is
    port (
        ck: in std_logic;
        reset: in std_logic;
        pc_en: in std_logic;
        pc_sel: in std_logic;
        pc_data_in: in std_logic_vector(7 downto 0);
        pc_data_out: buffer  std_logic_vector(7 downto 0)
    );
end PC;
architecture RTL of PC is
 signal tmp : std_logic_vector(7 downto 0);
begin
 tmp <= pc_data_out + 1;
  process(ck,reset)
  begin
   if ( reset = '0')then
    pc_data_out <= "00000000";
   else
      if ( ck'event and ck = '1')then
         if ( pc_en = '0')then
           if (pc_sel = '0')then
            pc_data_out <= pc_data_in;
           else
            pc_data_out <= tmp;
           end if;
         end if;
      end if;
   end if;
 end process;
```

```
end RTL;
```

リスト 6　MAR

```
-- MAR
library ieee ;
use ieee.std_logic_1164.all;
entity MAR is
    port (
        ck: in std_logic;
        reset: in std_logic;
        mar_sel: in std_logic;
        mar_en: in std_logic;
        mar_data_in: in std_logic_vector(7 downto 0);
        mar_alu_in: in std_logic_vector(7 downto 0);
        mar_data_out: out std_logic_vector(7 downto 0)
    );
end MAR;
architecture RTL of MAR is
begin
  process(ck,reset)
  begin
   if ( reset = '0')then
    mar_data_out <= "00000000";
   else
      if ( ck'event and ck = '1')then
         if ( mar_en = '0')then
           if (mar_sel = '0')then
            mar_data_out <= mar_alu_in;
           else
            mar_data_out <= mar_data_in;
           end if;
         end if;
      end if;
   end if;
 end process;
end RTL;
```

リスト 7　Flag

```
-- flag
library ieee;
use ieee.std_logic_1164.all;
entity FLAG is
    port (
        ck: in std_logic;
        reset: in std_logic;
        flag_en: in std_logic;
        flag_sel: in std_logic_vector(1 downto 0);
        flag_in_cf: in std_logic;
        flag_in_zf: in std_logic;
        flag_in_vf: in std_logic;
        flag_in_nf: in std_logic;
        flag_out_cf: out std_logic;
        flag_out_zf: out std_logic;
        flag_out_vf: out std_logic;
        flag_out_nf: out std_logic
    );
end FLAG;
architecture RTL of FLAG is
begin
 process(ck,reset)
  begin
    if (reset = '0')then
      flag_out_cf <= '0';
      flag_out_zf <= '0';
      flag_out_vf <= '0';
      flag_out_nf <= '0';
     else
       if (ck'event and ck = '1')then
        if (flag_sel = "00")then
         if (flag_en = '0') then
          flag_out_cf <= flag_in_cf;
          flag_out_zf <= flag_in_zf;
```

```
          flag_out_vf <= flag_in_vf;
          flag_out_nf <= flag_in_nf;
         end if;
        elsif (flag_sel = "01")then
         flag_out_cf <= '1';
        elsif (flag_sel = "10")then
         flag_out_cf <= '0';
        end if;
       end if;
     end if;
 end process;
end RTL;
```

リスト 8　レジスタ群

```
-- ir
library ieee;
use ieee.std_logic_1164.all;
entity IR is
    port (
        ck: in std_logic;
        reset: in std_logic;
        ir_en: in std_logic;
        ir_data_in: in std_logic_vector(7 downto 0);
        ir_data_out: out std_logic_vector(7 downto 0)
    );
end IR;
architecture RTL of IR is
begin
 process(ck,reset)
  begin
   if ( reset = '0')then
    ir_data_out <= "00000000";
   else
```

```
      if ( ck'event and ck = '1')then
         if ( ir_en = '0')then
           ir_data_out <= ir_data_in;
         end if;
      end if;
   end if;
 end process;
end RTL;
```

リスト９　最上位ブロック

```
-- K_CPU
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity k_CPU is
    port (
        ck: in std_logic;
        reset: in std_logic;
        start: in std_logic;
        test_cont: in std_logic;
        test_sel: in std_logic_vector(2 downto 0);
        data_in: in std_logic_vector(7 downto 0);
        dbb_cont: out std_logic;
        data_out: out std_logic_vector(7 downto 0);
        address_bus: out std_logic_vector(7 downto 0);
        memr: out std_logic;
        memw: out std_logic
    );
end k_CPU;
architecture arch of k_CPU is
component ALU_mux
    port (
```

```
        ir_o : in std_logic_vector(7 downto 0);
        acc1_o: in std_logic_vector(7 downto 0);
        acc2_o: in std_logic_vector(7 downto 0);
        ix_o: in std_logic_vector(7 downto 0);
        mdr_o: in std_logic_vector(7 downto 0);
        pc_o: in std_logic_vector(7 downto 0);
        mar_o: in std_logic_vector(7 downto 0);
        cont1 : in std_logic_vector(2 downto 0);
        cont2 : in std_logic_vector(2 downto 0);
        alu_mux_o1 : out std_logic_vector(7 downto 0);
        alu_mux_o2 : out std_logic_vector(7 downto 0);
         test_sel : in std_logic_vector(2 downto 0);
         test_o : out std_logic_vector(7 downto 0)
    );
end component;
component ALU
    port (alu_in1 :in std_logic_vector(7 downto 0);
          alu_in2 :in std_logic_vector(7 downto 0);
          alu_cont :in std_logic_vector(2 downto 0);
          alu_cin : in std_logic;
          alu_out :out std_logic_vector(7 downto 0);
          cf : out std_logic;
          zf : out std_logic;
          vf : out std_logic;
          nf : out std_logic
      );
end component;
component CONT
    port (
        ck: in std_logic;
        reset: in std_logic;
        start: in std_logic;
        opcode: in std_logic_vector(3 downto 0);
        ads_mode: in std_logic_vector(3 downto 0);
        cf,zf,vf,nf: in std_logic_vector(3 downto 0);
        alu_c: out std_logic_vector(2 downto 0);
```

```
        alu_mux_c1: out std_logic_vector(2 downto 0);
        alu_mux_c2: out std_logic_vector(2 downto 0);
        ir_c: out std_logic;
        acc1_c: out std_logic;
        acc2_c: out std_logic;
        ix_c: out std_logic;
        pc_c: out std_logic;
        pc_s: out std_logic;
        mar_c: out std_logic;
        mar_s: out std_logic;
        mdr_c: out std_logic;
        ads_mux: out std_logic_vector(1 downto 0);
        dbb_c: out std_logic;
        flag_c: out std_logic;
        flag_s: out std_logic_vector(1 downto 0);
        memr: out std_logic;
        memw: out std_logic
    );
end component;
component IR
    port (
        ck: in std_logic;
        reset: in std_logic;
        ir_en: in std_logic;
        ir_data_in: in std_logic_vector(7 downto 0);
        ir_data_out: out std_logic_vector(7 downto 0)
         );
end component;
component acc1
    port (
        ck: in std_logic;
        reset: in std_logic;
        acc1_en: in std_logic;
        acc1_data_in: in std_logic_vector(7 downto 0);
        acc1_data_out: out std_logic_vector(7 downto 0)
        );
```

```
end component;
component acc2
    port (
        ck: in std_logic;
        reset: in std_logic;
        acc2_en: in std_logic;
        acc2_data_in: in std_logic_vector(7 downto 0);
        acc2_data_out: out std_logic_vector(7 downto 0)
         );
end component;
component ix
    port (
        ck: in std_logic;
        reset: in std_logic;
        ix_en: in std_logic;
        ix_data_in: in std_logic_vector(7 downto 0);
        ix_data_out: out std_logic_vector(7 downto 0)
        );
end component;
component mdr
    port (
        ck: in std_logic;
        reset: in std_logic;
        mdr_en: in std_logic;
        mdr_data_in: in std_logic_vector(7 downto 0);
        mdr_data_out: out std_logic_vector(7 downto 0)
        );
end component;
component PC
    port (
        ck: in std_logic;
        reset: in std_logic;
        pc_en: in std_logic;
        pc_sel: in std_logic;
        pc_data_in: in std_logic_vector(7 downto 0);
        pc_data_out: buffer  std_logic_vector(7 downto 0)
```

```
        );
end component;
component MAR
    port (
        ck: in std_logic;
        reset: in std_logic;
        mar_sel: in std_logic;
        mar_en: in std_logic;
        mar_data_in: in std_logic_vector(7 downto 0);
        mar_alu_in: in std_logic_vector(7 downto 0);
        mar_data_out: out std_logic_vector(7 downto 0)
        );
end component;
component ADDRESS
    port (
        ads_mode: in std_logic_vector(1 downto 0);
        pc: in std_logic_vector(7 downto 0);
        mar: in std_logic_vector(7 downto 0);
        mdr: in std_logic_vector(7 downto 0);
        ix: in std_logic_vector(7 downto 0);
        address_out: out std_logic_vector(7 downto 0)
         );
end component;

component MUX
    port (
        data_sel: in std_logic;
        test_output: in std_logic_vector(7 downto 0);
        alu_output: in std_logic_vector(7 downto 0);
        data_out: out std_logic_vector(7 downto 0)
        );
end component;
component FLAG
    port (
        ck: in std_logic;
        reset: in std_logic;
```

```
        flag_en: in std_logic;
        flag_sel: in std_logic_vector(1 downto 0);
        flag_in_cf: in std_logic;
        flag_in_zf: in std_logic;
        flag_in_vf: in std_logic;
        flag_in_nf: in std_logic;
        flag_out_cf: out std_logic;
        flag_out_zf: out std_logic;
        flag_out_vf: out std_logic;
        flag_out_nf: out std_logic
       );
end component;
signal ir_out,acc1_out,acc2_out,ix_out,mdr_out,
           pc_out,mar_out : std_logic_vector (7 downto 0);
signal alu_bus : std_logic_vector(7 downto 0);
signal alu_in1,alu_in2 : std_logic_vector(7 downto 0);
signal alu_c: std_logic_vector(2 downto 0);
signal alu_mux_c1: std_logic_vector(2 downto 0);
signal alu_mux_c2: std_logic_vector(2 downto 0);
signal ir_c: std_logic;
signal acc1_c: std_logic;
signal acc2_c: std_logic;
signal ix_c: std_logic;
signal pc_c: std_logic;
signal pc_s: std_logic;
signal mar_c: std_logic;
signal mar_s: std_logic;
signal mdr_c: std_logic;
signal ads_mux: std_logic_vector(1 downto 0);
signal flag_c: std_logic;
signal flag_s: std_logic_vector(1 downto 0);
signal test_output: std_logic_vector (7 downto 0);
signal cf_o,zf_o,vf_o,nf_o: std_logic;
signal cf_i,zf_i,vf_i,nf_i: std_logic;

begin
```

```
 U1 : ALU_mux port map (ir_out,acc1_out,acc2_out,ix_out,
                        mdr_out,pc_out,mar_out,
                        alu_mux_c1,alu_mux_c2,
                        alu_in1,alu_in2,
                        test_sel,
                        test_output);
 U2 : ALU port map (alu_in1,alu_in2,alu_c,cf_o,
                    alu_bus,
                    cf_i,zf_i,vf_i,nf_i);
 U3 : CONT port map (ck,reset,start,
                     ir_out(7 downto 4),ir_out(3 downto 0),
                     cf_o,zf_o,vf_o,nf_o,
                     alu_c,alu_mux_c1,alu_mux_c2,
                     ir_c,acc1_c,acc2_c,ix_c,pc_c,pc_s,mar_c,mar_s,
                     mdr_c,ads_mux,dbb_cont,flag_c,flag_s,
                     memr,memw );
U4 : IR port map (ck,reset,ir_c,data_in,ir_out);
U5 : ACC1 port map (ck,reset,acc1_c,alu_bus,acc1_out);
U6 : ACC2 port map (ck,reset,acc2_c,alu_bus,acc2_out);
U7 : IX port map (ck,reset,ix_c,alu_bus,ix_out);
U8 : MDR port map (ck,reset,mdr_c,data_in,mdr_out);
U9 : PC port map (ck,reset,pc_c,pc_s,data_in,pc_out);
U10 : MAR port map (ck,reset,mar_s,mar_c,data_in,alu_bus,mar_out);
U11 : ADDRESS port map (ads_mux,pc_out,mar_out,
                              mdr_out,ix_out,address_bus);
U12 : MUX port map (test_cont,test_output,alu_bus,data_out);
U13 : FLAG port map(ck,reset, flag_c,flag_s,
                     cf_i,zf_i,vf_i,nf_i,
                     cf_o,zf_o,vf_o,nf_o);
end arch;
```

4.3.3 シミュレーション

記述した CPU をシミュレーションによって動作を検証する。

（1） ロード命令

ロード命令のシミュレーションを図 4.52 に示す。

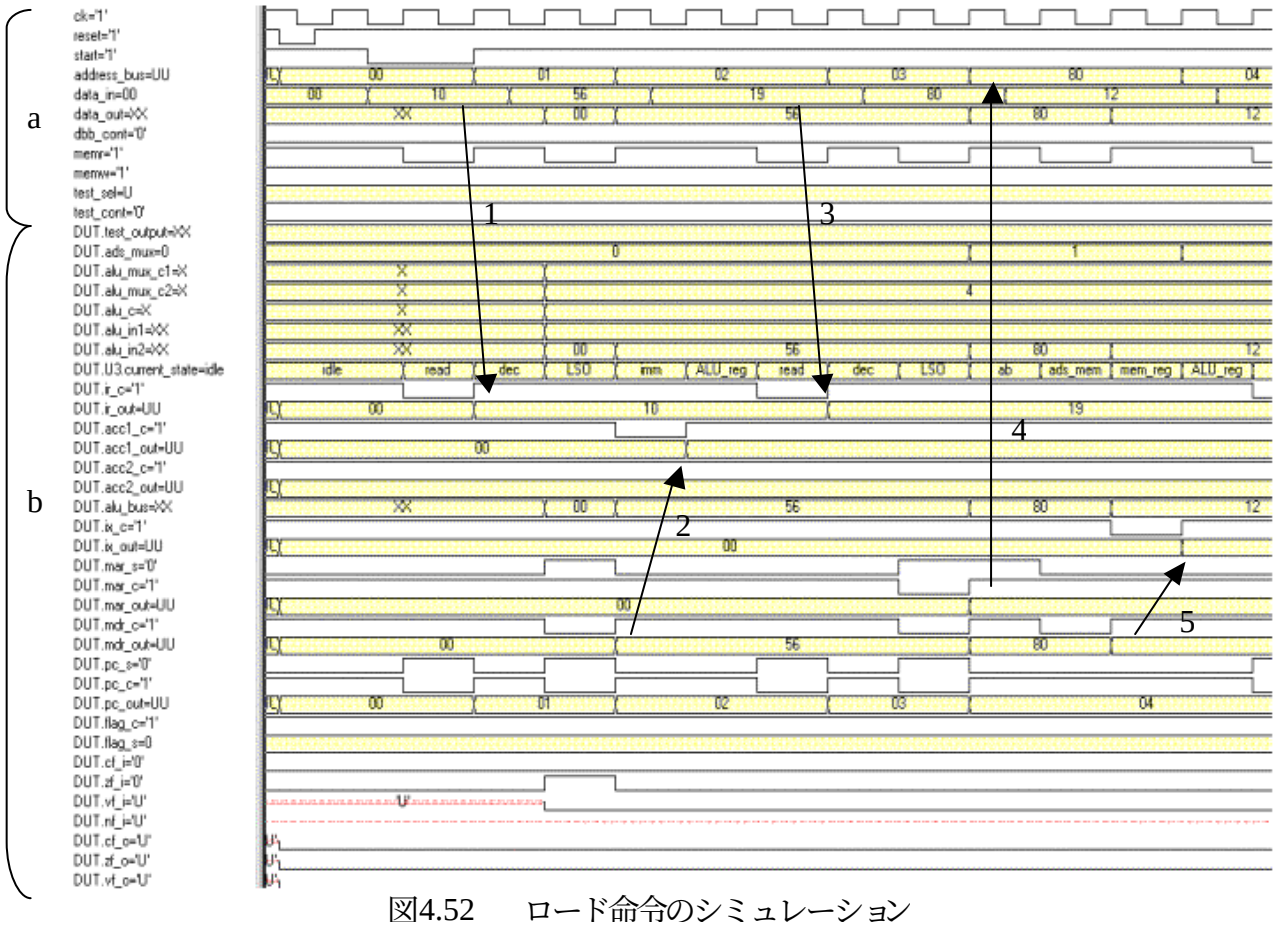


図4.52　　ロード命令のシミュレーション

シミュレーションは即値アドレスで ACC1 へデータ 56H を格納するものと、アドレス 80H のデータ 12H を IX へ格納するものである。

図の左側の a、b は信号線またはデータを示している。a は CPU 本体の入出力信号であり、b は内部の制御線とレジスタの出力である。

図中の矢印 1 は memr が立ち下がって IR に命令が格納された状態であり、2 は MDR に取り込まれたデータ 56H を ACC1 に格納した状態である。3 は次の命令を IR に取り込んだ状態で、4 は MAR に取り込まれたデータ 80H がアドレスとなったことを示し、5 は MDR に取り込まれたデータ 12H を IX に格納した状態である。

（2） **ストア命令**

ストア命令のシミュレーションを図 4.53 **に示す。**

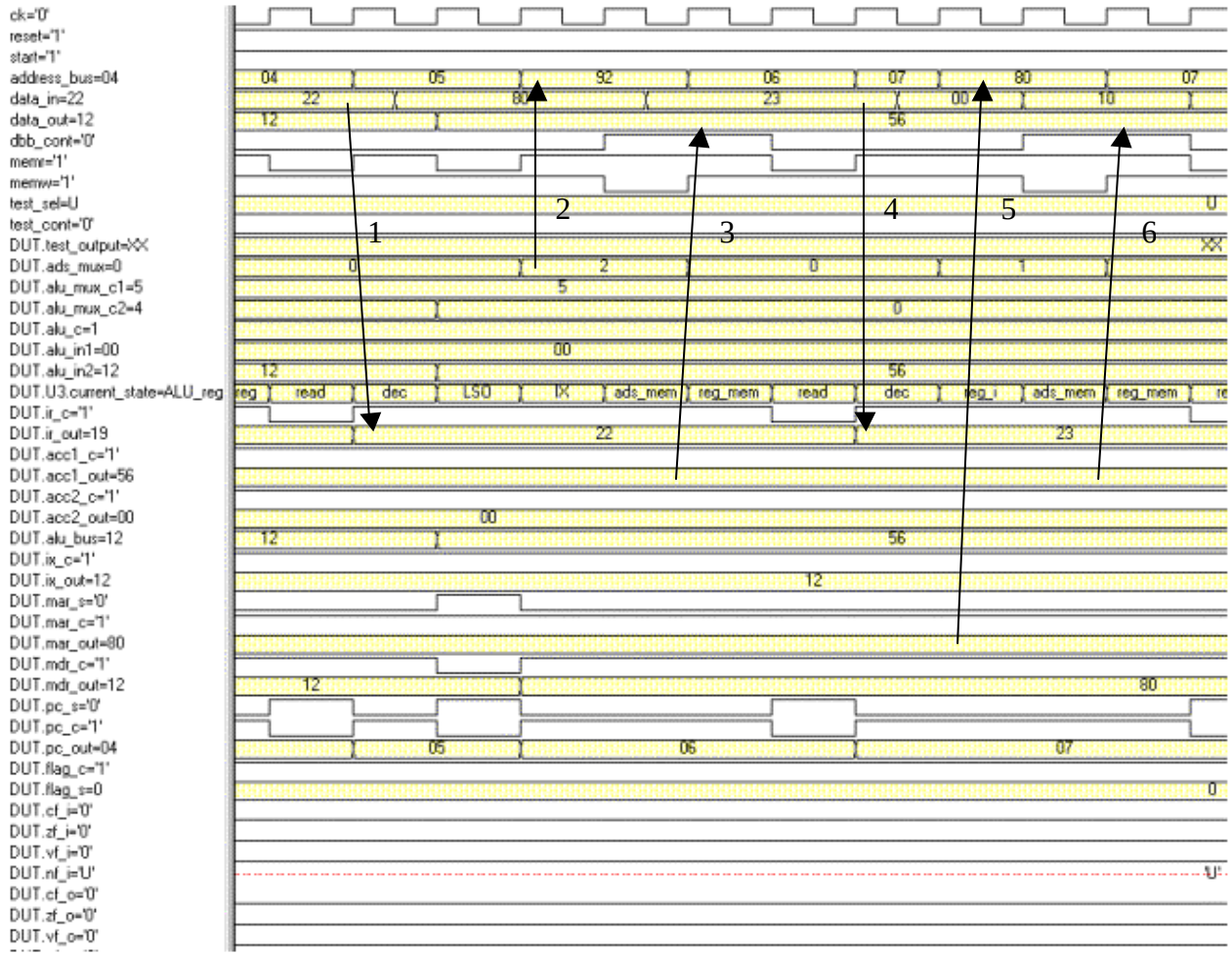


図4.53　　ストア命令のシミュレーション

ストア命令のシミュレーションはインデックス修飾アドレスとレジスタ間接アドレスで ACC1 **のデータをメモリへ格納するものである。図中の矢印** 1 **は命令が** IR **に格納され状態で、あらかじめ** IX **に** 12H **というデータがあり、**80H **を** MDR **に取り込んで、**IX **のデータとの加算結果** 92H **がアドレスとなった状態が矢印** 2 **である。そして矢印** 3 **は** memw **が立ち下がって出力に** ACC1 **の内容が出力された状態である。矢印** 4 **は次の命令を** IR **に取り込んだ状態で、**MAR **の内容（**80H**）がアドレスとなり、**5 **で** memw **が立ち下がって出力に** ACC1 **の内容が出力された状態である。**6 **は** ACC **1 の内容が出力となった状態である。**

（3） データ移動命令

データ移動命令のシミュレーションを図 4.54 **に示す。**

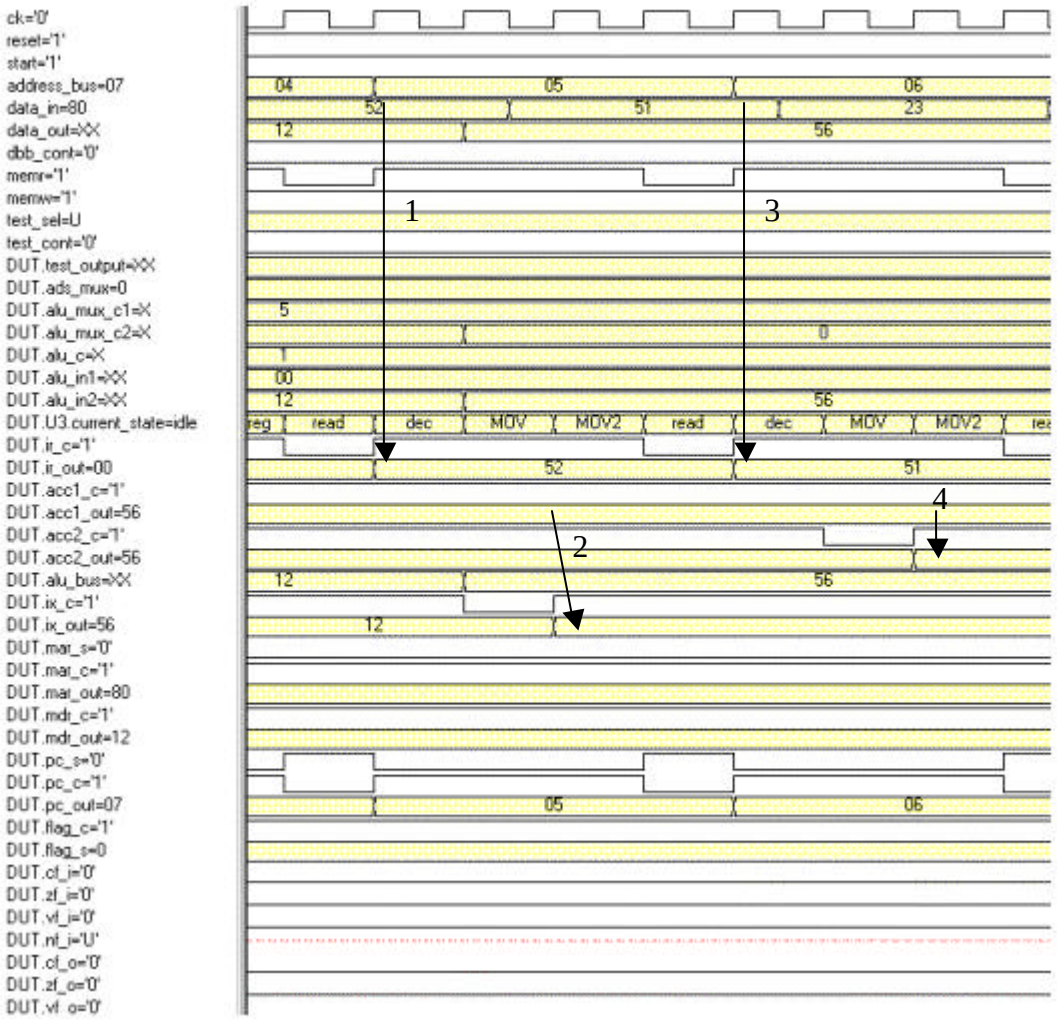


図4.54　データ移動命令のシミュレーション

データ移動命令のシミュレーションは、ACC1 **の内容（**56H**）を** IX**、**ACC2 **にコピーするものである。図中の矢印** 1 **で** IR **に命令が取り込まれ、矢印** 2 **で** ACC1 **の内容が** IX **に移っている。矢印** 3 **で次の命令を取り込み、矢印** 4 **で** ACC1 **の内容が** ACC2 **に移っている。**

（4） フラグセット命令

フラグセット命令のシミュレーションを図 4.55 に示す。

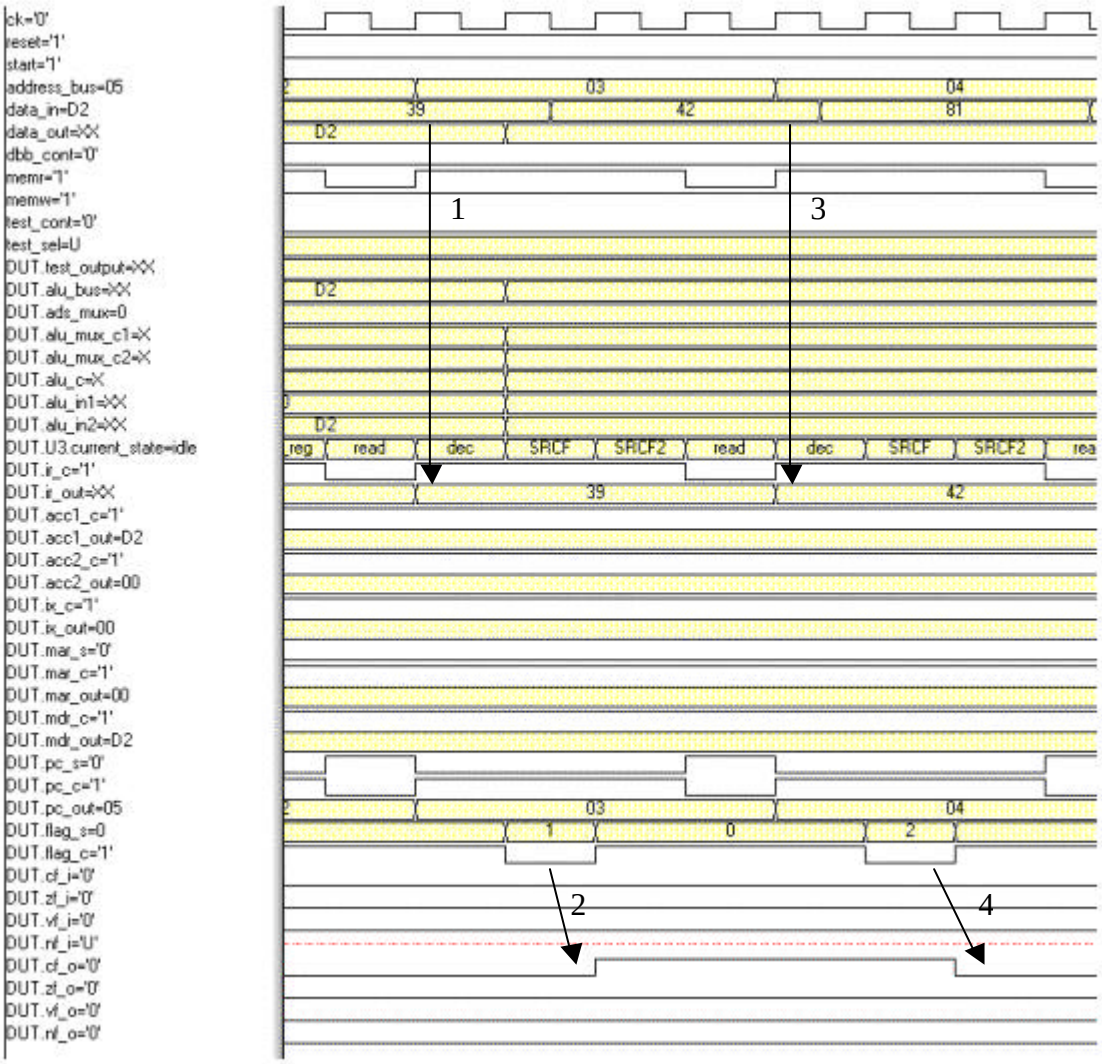


図4.55　フラグセット命令のシミュレーション

フラグセット命令のシミュレーションは CF を“ 1 ”にセットする命令（SCF）と“ 0 ”にリセットする命令（RCF）である。図中の矢印 1 で命令（SCF）が取り込まれ、矢印 2 で flag_c が立ち下がって CF が“ 1 ”にセットされている。このとき flag_s の信号は CF をセットするために“ 01 ”となっている。矢印 3 で次の命令（RCF）が取り込まれ、矢印 4 で flag_c が立ち下がって CF が“ 0 ”に
リセットされている。このとき flag_s の信号は CF をリセットするために“ 10 ”となっている。

（5）演算命令

演算命令のシミュレーションを図 4.56 に示す。

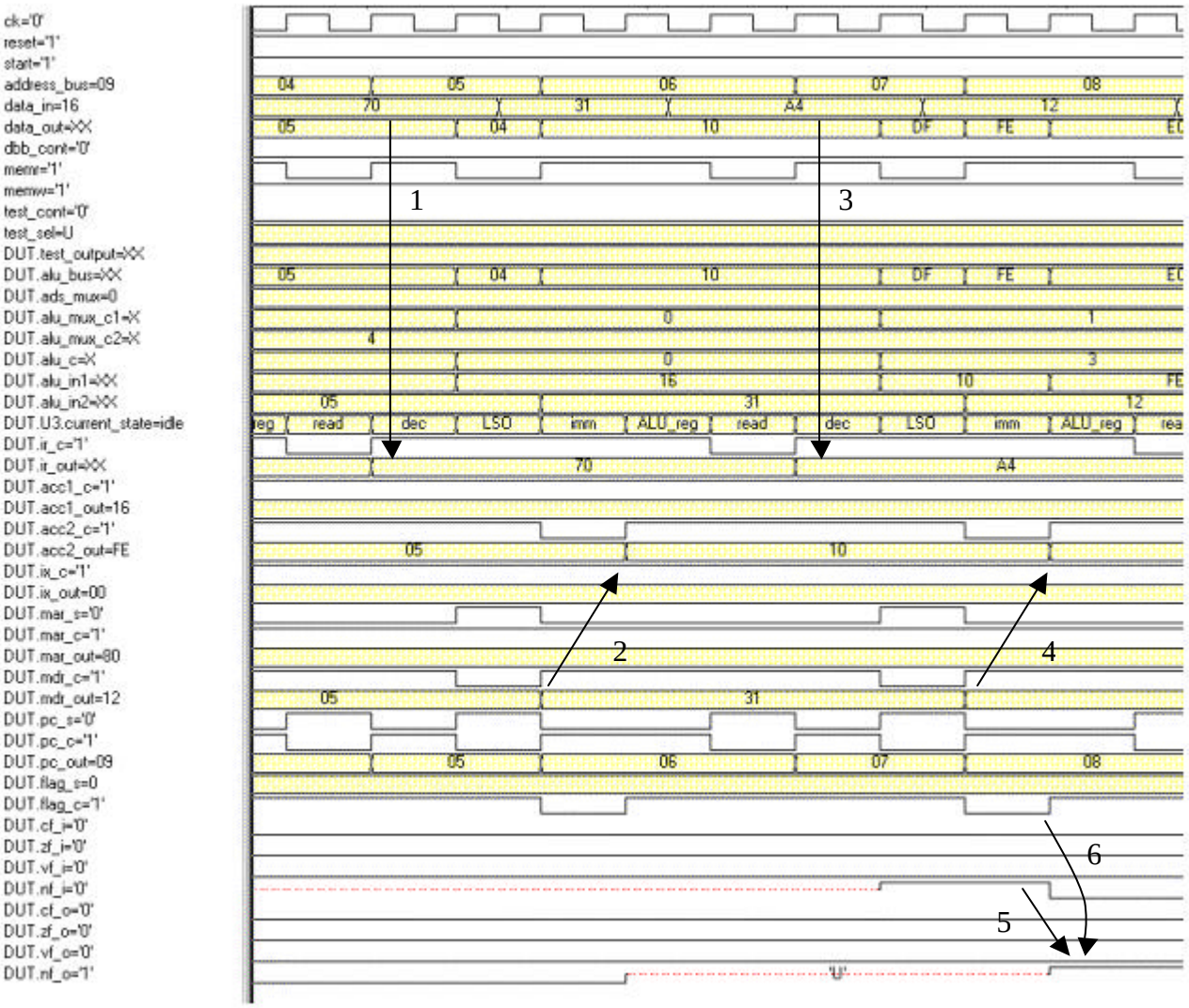


図4.56　演算命令のシミュレーション

演算命令のシミュレーションは、ACC1 のデータ（16H）と即値アドレスで読み込んだデータ（31H）の論理積と、ACC2 のデータ（10H）と即値アドレスで読み込んだデータ（12H）の比較である。図中の矢印 1 は論理積命令を読み込んだ状態であり、矢印 2 は演算結果（10H）が ACC2 に格納された状態である。矢印 3 で比較命令が読み込まれ、矢印 4 は ACC2（10H）とデータ 12H の比較（減算）結果が ACC2 に格納された状態である。そして比較結果は負であり、ネガティブフラグ（NF）が立つ。矢印 5 は NF が立った状態を示し、矢印 6 は flag_c が立ち下がってフラグレジスタにフラグの状態が取り込まれた状態を示している。

（6）分岐命令

分岐命令のシミュレーションを図 4.57 に示す。

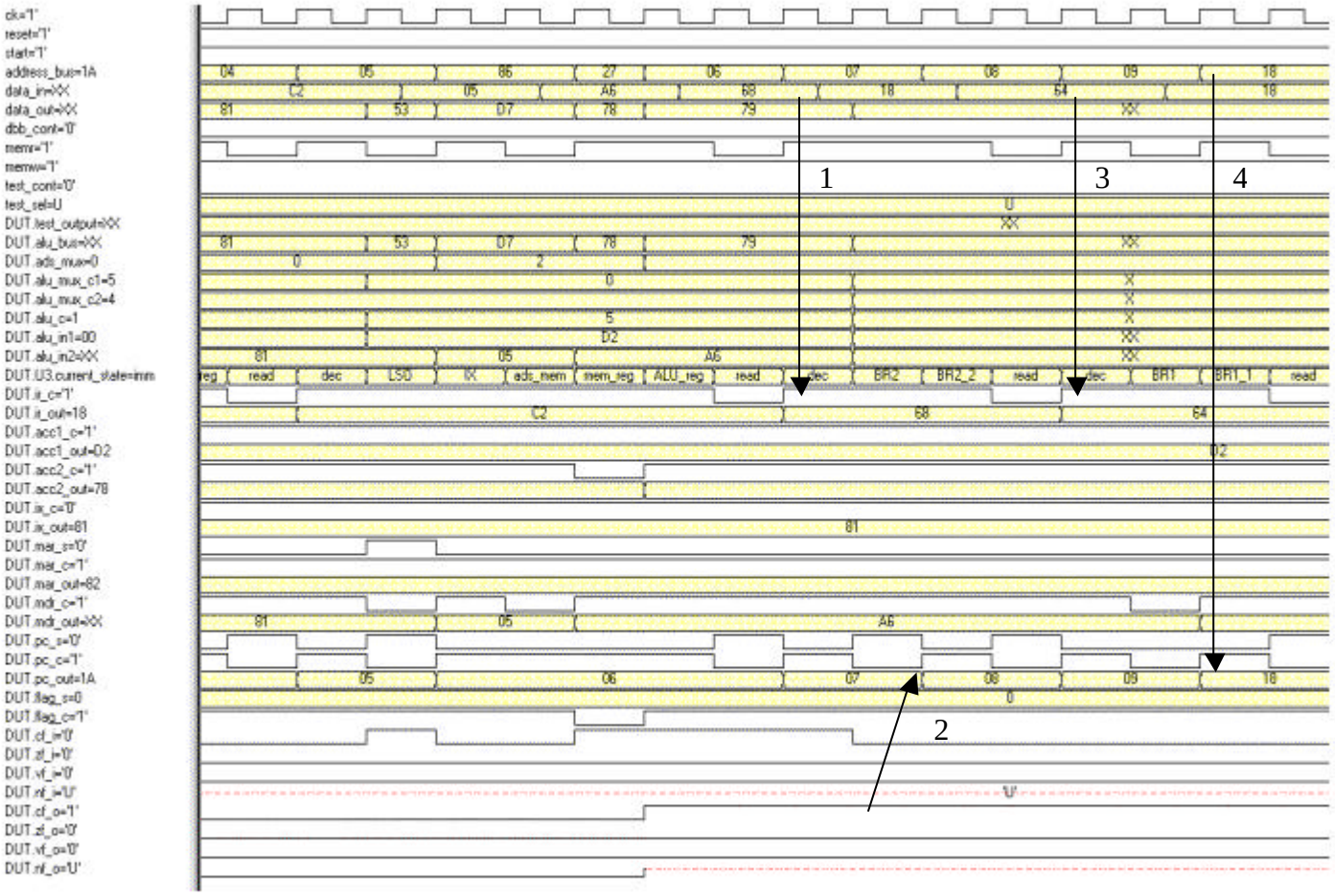


図4.57　分岐命令のシミュレーション

分岐命令のシミュレーションは分岐条件が成立した場合と成立しなかった場合についてである。分岐命令を読み込む前に、あらかじめキャリー付加算を行い CF を立てておく。図中の矢印 1 は CF を判断する分岐命令を IR に読み込んだ状態であり、矢印 2 は CF が立っているので条件成立となり、PC をインクリメントして終了した状態を示している。矢印 3 は ZF を判断する分岐命令を読み込んだ状態であり、矢印 4 は ZF が立っていないので分岐先のアドレスを PC に読み込んだ状態を示している。

（7）テストモード

テストモードのシミュレーションを図 4.58 に示す。

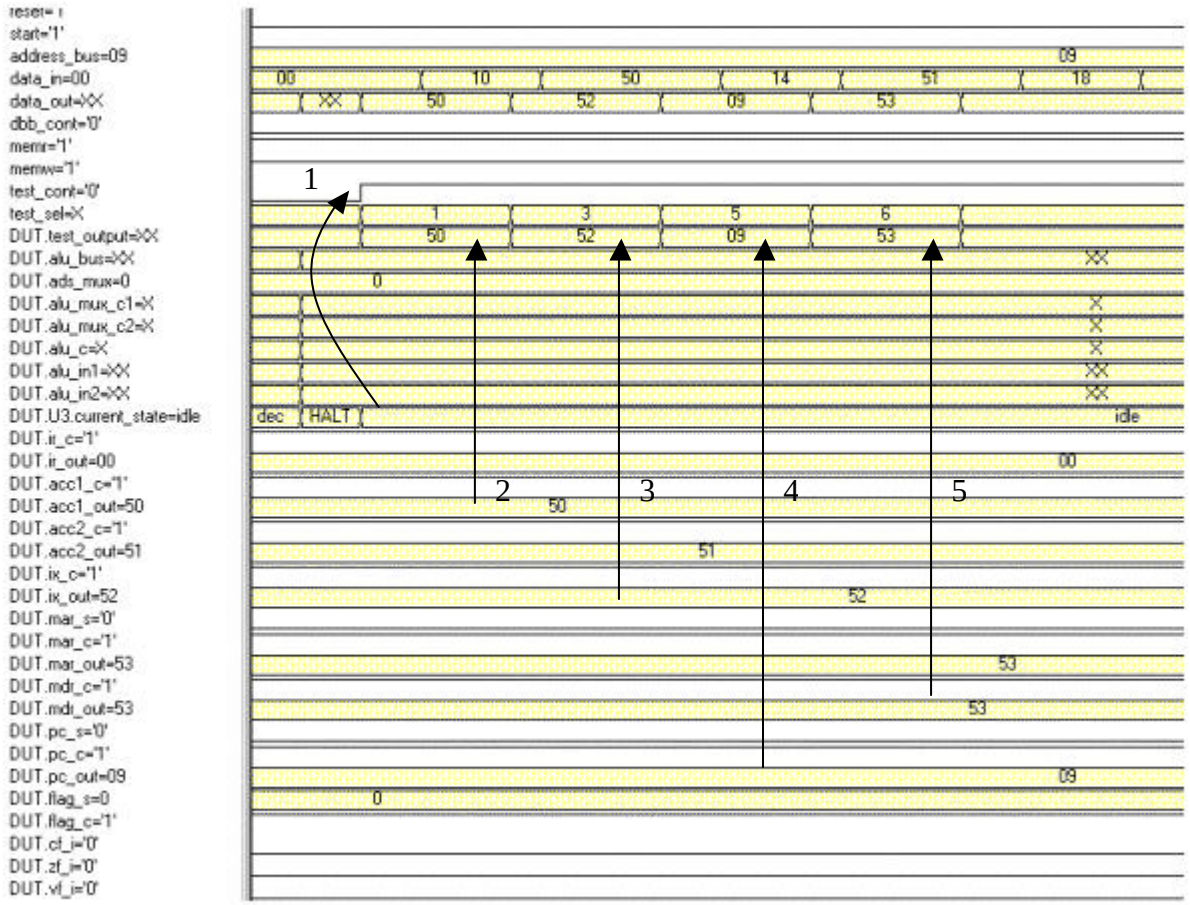


図4.57　テストモードのシミュレーション

テストモードのシミュレーションは CPU の動作を停止してからレジスタの内容を確認する。図中の矢印 1 は動作を停止した CPU からテスト用信号線 test_cont を“1”にしてテストモードにしたことを示している。矢印 2 はレジスタ選択信号 test_sel を‘001’にした場合は ACC1 が選択され、ACC1 の内容（50H）が出力されていることを示し、矢印 3 は test_sel を“011” にした場合の IX の内容（52H）が出力されていることを示し、test_sel を“101”にした場合の PC の内容（09H）が出力されていることを示し、test_sel を“110”にした場合の MDR の内容（53H）が出力されていることを示している。

4.4　考察

独自の CPU のアーキテクチャ設計をするにあたって、様々な文献を調査したこと､そして簡単ではあるが仕様まではできたことは有意義であったと思う。

内容については、やはりタイミングを決定するのと制御部の設計が難関であった。制御部がきっちりできていないとすべてがうまく動作しないため、時間をかけざるを得なかった。

今後の課題としては、制御のやり方をもっとうまくできる方法を考えることと、ハードウェアの構成を意識しながらの HDL の記述ができることである。

5. まとめ

今回の実験で、サイコロ・マシンの製作と、CPU アーキテクチャの設計の 2 つに共通していえることがある。それは全体の見通しの甘さである。設計、製作の後の段階でミスがわかり、そのつど修正、変更あるいはやり直しをした。実験を通して、最初の段階でよく考えることがいかに大切であるかを学び、次からの設計には、この反省をもとにこれをいかに少なくできるかが今後の課題となる。

謝辞

本研究の全過程を通じて懇切な指導をいただいた原 央教授、適宜指導をいただいた矢野 政顯教授、橘 昌良助教授に厚く御礼申し上げます。