第 1

はじめに

1.1 なぜコンピュータアーキテクチャを勉強するのか？

コンピュータアーキテクチャの勉強とは、簡単に言うと、コンピュータの中身を知り、それがどのように動いてどのように作られているかを理解することだ。アーキテクチャというのは「建築」という意味で、コンピュータアーキテクチャはコンピュータという建築物を、どのように作っていくかについての方法論である。

Web を作ったり、グラフィックを作ったりするのは格好いいので、コンピュータのプログラムはするかもしれないけれど、コンピュータを設計するわけではないので、コンピュータの中身には興味がない。

ケータイは便利だし、ゲーム機は凄い。iPod は格好いいし、地デジはきれい、こういう製品を作る所に行きたいけど、コンピュータは関係ない。大体マイクロプロセッサは、もう日本では作ってないし、Intel が作るのを買えばいい。コンピュータの構成なんて、もう進歩が止まっている分野で大きな発明や面白いことは何もない。要するに、コンピュータアーキテクチャは興味も関心もない。多くの電気、電子情報系の大学では、基礎科目としてコンピュータアーキテクチャ、コンピュータ工学などが設けられている。進級のために履修はするけど面倒なだけ、と思っている方々もおられるかと思う。しかし、これは誤った考え方である。

まず第一に、コンピュータのプログラマは、それが何を対象にするのであれ、一通りのコンピュータアーキテクチャの知識を持っている必要がある。コンピュータアーキテクチャはプログラマのために、抽象化を行って、細かい所は気にしなくてもプログラム書けるようにしている。したがって、プログラマを目指す方々は、コンピュータハードウェアの詳細を知る必要はない。しかし、コンピュータアーキテクチャに関して一応の知識がないと、良いプログラムは書けないし、もちろんプロにはなれない。

次に IT 機器の開発者を目指す人たちは、より詳しくコンピュータアーキテクチャに関して知っておく必要がある。ケータイ、ゲーム、情報家電、自動車に至るまで、あらゆる IT 機器にコンピュータは組み込まれている。これらは組み込みコンピュー

タと呼ばれて、多くの場合、その IT 機器を制御する中心としての役割を果たす。現在、日本の半導体業界は、組み込みコンピュータをその部品の一部とし、他にメモリ、専用プロセッサ、インタフェースなどを一つのチップに入れるシステム LSI あるいは SoC(System-on-a Chip) を主力製品としている。ケータイに入っているチップはこのシステム LSI の代表であり、最近はどの IT 機器もこれが頭脳としての役割を果たす。したがって、IT 機器に関連する場合、コンピュータアーキテクチャを理解していないと、製品開発をすることができないし、もちろんマネージャになることも、下手をすると営業をするにも支障が出る。

このように、今やコンピュータは、デスクトップ PC, ラップトップ PC あるいはサーバなどだけではなく、IT 産業の基礎となるパーツであり、IT 産業に携わる者の基礎的な知識として必要である。これが多くの大学の電気、電子、情報系の基礎科目にコンピュータアーキテクチャが入っている理由である。

また、コンピュータアーキテクチャは、進歩が止まってしまった分野でも、重要性が小さくなっていく分野でもない。確かにコンピュータの中心部、CPU(Central Processing Unit:中央処理装置) の構成自体は、ほぼ固まっているし、その動作周波数もこれ以上高くできなくなっている。しかし、このことによって新しいアーキテクチャの研究開発はむしろ活気付いている。多数の CPU を使ったマルチコア、メニーコア、限られた処理のみを高速化するアクセラレータ、汎用化されたグラフィックプロセッサ GP-GPU(General Perpuse Graphic Processing Unit)、柔かいハードウェアと呼ばれるリコンフィギャラブルシステム、動的に構成を変更する動的リコンフィギャラブルプロセッサなど新しいアーキテクチャが次々に登場しており、実際の製品にも使われている。

つい最近まで、コンピュータの性能は圧倒的に半導体のプロセスによって支配された。アーキテクチャが同じでも、新しい半導体プロセスを使えば、性能は上がり、コストと消費電力は下がった。このため、アーキテクチャなんてどうでもいいから新しいプロセスで新しい製品を作れば、うまく行ったのである。しかし、最近、半導体の微細化加工技術が限界に近づくにつれ、新しいプロセスの開発は難しくなり、プロセスの代替わりの間隔は開きつつある。また、プロセスが新しくなっても、特に性能面では目立って改善されないようになった。一つのプロセスが長い時間使われる場合に重要になるのは、それをいかに活用するか、すなわちコンピュータを含めた広い意味でのアーキテクチャである。今後 IT 製品においては、半導体プロセスを新しくするよりも、より良いアーキテクチャを使う方が良い製品につながっていく。すなわちアーキテクチャの重要性は今後どんどん大きくなっていくのである。

今まで、コンピュータアーキテクチャを学ぶことの実利についてを紹介したが、コンピュータアーキテクチャを学ぶもっとも正しい動機は、それが面白いからだ。解説が進むに従って、原理的には非常に簡単なものが、(もちろんソフトウェアの力が大きいとはいえ)、凄いことをやってのけることがご理解いただけると思う。本書を完全にマスターすれば、自分自身でコンピュータを設計し、FPGA(Field Programmable Gate Array) や実際のチップ上で動かすことができるのだ。また、建築物同様、良くで

きたコンピュータは美しい。互換性を守りながら性能を上げるために、理解不能なレベルで複雑化した Intel 系の CPU にすら、長年、増築と改築を重ねた歴史的建造物のような美を感じることができるかもしれない。

1.2 コンピュータの基本

コンピュータは大きく分けて三つの領域で主に使われる。

- デスクトップ PC、ラップトップ PC: いわゆるコンピュータっぽいコンピュータ。Windows, Linux が Operating System として走り、Intel 系の命令が走る CPU(中央処理装置) を用いている。無線や LAN でネットワークに接続して用いる。最も発展が急速で、性能と値段が重要。

- サーバ: 大企業や大学のシステムを統括する大型システム。ネットワークを介してデスクトップ PC やラップトップ PC と接続し、ファイルを管理し、デスクトップなどで手に余る大規模な計算を行う。あるいは、トランザクション処理と言って、データベースに接続されて、データ管理を行う。銀行の ATM、航空券の予約システムはこの例である。Web を管理して、検索等を行う Web サーバも大規模なサーバの一つ。故障しないで長期間動くこと、故障からの回復が早いこと等信頼性 (dependability) が重要。

- 組み込みシステム: 様々な IT 機器のなかに組み込まれており、外からは見えないし、普通はプログラムもしない。洗濯機やクーラーを制御するのは 8bit のコンピュータで可能だが、PS3 などのゲーム機には、Cell という超強力なプロセッサが組み込まれているし、ケータイの CPU も特定用途には強力である。コストと消費電力と、特定分野に限った性能が重要。

1.2.1 コンピュータの基本形

コンピュータはそれがデスクトップ PC であれ、組み込み用であれ、基本的に図 1.1 に示すように、CPU(Central Processing Unit:中央処理装置) と命令やデータを記憶しておくメモリ、外部とのやり取りを行う入出力 (I/O) の三つの部分から出来ている。CPU は、メモリから命令を基本的には順番に取ってきて、その中に書いてあることを実行することを繰り返す。これをプログラム格納型 (Stored Programming) と呼ぶ。データは入出力装置から取り込み、メモリ上に置き、CPU はこれを読み書きしながら処理を進め、結果をメモリから出力装置に出力して表示する。

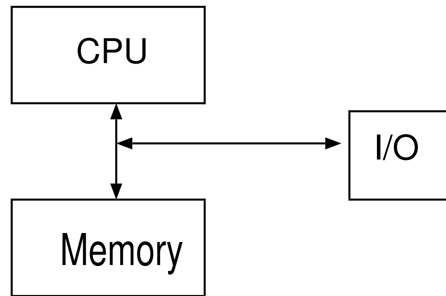


図 1.1: コンピュータの三つの要素

1.2.2　コンピュータの歴史と発展

最初のコンピュータは何で、誰が発明者なのか？という問題は、なかなか難しく、これを検討するだけで一冊の本ができてしまう[1] のだが、最初の電子計算機として最も良く知られているのは、ペンシルバニア大学の ENIAC(1946 年) である。ENIAC は、全ての回路を真空管で実装した（18000 本）点で電子計算機であったが、プログラミングにはスイッチの設定やケーブルの結線を人手で変更する必要があった。この ENIAC の経験を基に、命令をメモリ上に格納し、これを順に取ってきて実行するプログラム格納方式の考え方が生まれ、ケンブリッジ大学の EDSAC(1949 年) が誕生した。これがはじめてのプログラム格納型計算機であり、現在のコンピュータは基本的にこの方式から変わっていない。

コンピュータの歴史についても様々な文献や著書があり、ここでは詳しく触れないが、コンピュータについて歴史上最も重要な変革があったのは実はつい最近である。すなわち、コンピュータの歴史は、その使われ方から以下のように分けられる。

1.2.3　計算機登場〜1980 年後半: 中央集権時代

メインフレームと呼ばれる大型計算機が計算センターにどんと置かれており、これを共同で利用した。初期は、一人の利用者がコンピュータを占有したが、その後オペレーティングシステムが複数のジョブを管理することができるようになり、あらかじめパンチカードから入力したプログラムをテープやディスクに入れておき、順に実行するバッチ処理が可能になった。さらに端末をつないで対話的にジョブを実行する TSS(Time Sharing System) が登場した。この時代、計算機といえばメインフレーム=IBM マシンであった。このメインフレームは、ハードウェアの素子技術によって以

[1] この問題に興味のある方は、星野力著「誰がどうやってコンピュータを創ったのか？」(共立出版) をお勧めする。

下のように世代 (generation) 分けされる[2]

- 第 1 世代: 真空管 (1946〜1950 年代中ごろ)
- 第 2 世代: トランジスタ (1950 年代中ごろ〜1960 年代中ごろ)
- 第 3 世代: 集積回路 (1960 年代中ごろ〜)

IBM は、1964 年に IBM360 を発表し、これがメインフレームのアーキテクチャの方向を決定づけた。キャッシュ、割り込み、仮想記憶、本格的オペレーティングシステム（OS)、リアルタイムシステムやトランザクション処理等、現在の計算機の技術のほとんどはメインフレームによって発達を遂げた。一方で、用途に応じたコンピュータ自体の分化も進んだ。DEC 社の PDP-11 を代表するミニコンピュータは、メインフレームに比べて安価で気軽に使えるため、研究・教育施設に広く用いられた。また、科学技術計算を専用に行うユーザのためには、強力な数値演算能力を持つ CRAY-I に代表されるスーパコンピュータが登場した。

1.2.4 1980 年代後半〜2003 年：マイクロプロセッサ大成長時代

一人が一台の PC(Personal Computer) あるいは WS(Work Station) と呼ばれるコンピュータを持ち、これらがネットワークで接続されていて、互いに情報を交換しながら処理を行なう、現在一般的になっている使い方が確立したのがこの時代である。

初めてのマイクロプロセッサと呼ばれる Intel の i4004 が誕生したのは、1971 年である。マイクロプロセッサは、計算機の CPU を 1(数) チップに格納したもので、電子機器制御、ゲーム、簡単なコンピュータに用いられ、8bit, 16bit, 32bit と bit 数を増やしながら急速に発達を遂げた。さらに、1980 年代前半に普及した RISC(Reduced Instruction Set Computer) 方式により、マイクロプロセッサの性能は年々飛躍的に増大（年間 1.5 倍）するようになった。

ここに以下の状況が加わった結果、1980 年代後半に革命的な事態が起こった。

- ビットマップディスプレイによるウインドウ、マウス等 WS/PC のユーザインタフェース技術が確立した。
- 高級言語（C 言語）の普及により、アセンブラに依らずにシステムが全て記述できるようになった。
- UNIX, MS-DOS などの汎用 OS が普及し、OS の移植が容易になった。

[2] 第 3 世代後に大規模集積回路 (LSI) で構成された第 4 世代を設定する考え方もあるが、LSI は集積回路の単に大きいものに過ぎないので、どこで切るかが苦しくなっている。日本で 1982 年から 10 年間に渡って行なわれた第 5 世代コンピュータプロジェクトが失敗したため、これ以降、第 5 世代という言葉を使う人は居なくなり、計算機を世代で分ける考え方は消滅した。

- ローカルコンピュータネットワークが、Ethernet によって標準化され、さらに、インターネットの登場により、グローバルネットワークの基盤技術が整った。

これらの技術的基盤が整備されることによって、それまで広くもちいられてきたメインフレームに代わって、最新の半導体技術を用いた高性能マイクロプロセッサを用いた、個人用の PC や WS が急速に普及した。これらの PC ／ WS は、Windows, Linux の汎用 OS を搭載し、ネットワークに接続して用いる。この結果、コンピュータの主流はメインフレームから PC や WS に移り、また PC と WS の区別は消滅した。さらには、ラップトップ型の可搬型のコンピュータ、携帯端末が普及した。

1.2.5　2004 年〜：マルチコア時代

2003 年まで、マイクロプロセッサの動作周波数は年々向上し、性能も年間 1.5 倍の伸び率を示した。しかし、以下の原因で、マイクロプロセッサ CPU 単体の性能は壁にぶつかった。

- 高い周波数を用いたことにより発熱が増大し、これ以上周波数を上げると放熱することが困難になった。
- CPU の動作速度にメモリが付いていけなくなり、これ以上動作周波数を上げても性能があまり上がらなくなった。
- 単体 CPU の性能向上を支えてきた、命令レベルの並列処理が限界に達した。

このため、CPU 単体の性能を上げるよりも複数の CPU を 1 チップに搭載した構成が一般的に使われるようになった。2010 年現在、高性能のデスクトップでは 1 チップに 4 つの CPU を搭載する構成が一般化している。また、100 を越える数の CPU を搭載するメニーコアも作られ始めている。

1.3　本書の目的

従来、コンピュータアーキテクチャのテキストは、コンピュータの専門家を対象とした分 Patterson&Hennesy のテキスト [1] および Hennessy&Patterson [2] の厚い決定版テキストを除いては、コンピュータのプログラマやユーザーが、自分達の利用しているコンピュータの構造を「知識」として理解するために書かれたものが多かった。このようなテキストは、おおむねトップダウンに書かれており、コンピュータアーキテクチャの様々な技術を概観し、知識として理解するためには優れたものが数多く書かれている。

しかし、システム LSI 時代にコンピュータのアーキテクチャを学ぶ場合、その構造と動作を概観するのではなく、具体的に構造を理解し、できれば簡単なプロセッサを、

自分で設計した経験を持つことが望ましい。本書は、このような時代の要求に基づき、コンピュータアーキテクチャの初学者を対象として、最終的にはパイプライン化された RISC を目標として、ボトムアップに設計しながら、設計演習とプログラミング演習を通じてコンピュータアーキテクチャの実際的な設計技術を修得するのが目的である。設計には、Verilog-HDL を用い、シミュレーションにはフリーソフトで誰でも使える Ikarus Verilog と、gtkwave を利用している。PC をお持ちの方ならば自宅で誰でも実際に設計演習が可能である。また、web 上の教材を用いれば、実際の FPGA のキット上で実習が可能である。[3]

本書の初版が出版されたのは 2001 年である。今回改訂したのは、以下の点である。

- 初版本では、PARTHENON/SFL を用いた。これは、当時フリーソフトで利用できた唯一のハードウェア記述言語とそれに基づく設計環境であったこと、美しい言語構造で教育用途に優れていた点である。今でも SFL が Verilog-HDL よりも優れたハードウェア言語であるという考えには変わりがない。しかし、SFL を実際の設計現場で利用する機会がほとんどなくなったこと、Verilog-HDL にもフリーソフトで優れたシミュレーション環境が普及したことから改訂に踏み切った。本書の Verilog 記述に対応する SFL 記述は web 上で公開するので、PARTHENON の利用者はご面倒だがそちらを参照していただきたい。

- 初版本では、教育用マイクロプロセッサ PICO を用いた。PICO は、本格的な RISC の骨格を持ちつつ、分かりやすい、ということを目的に設計したプロセッサであった。ところが、「これでもまだ難しい」という批判があり、より分かりやすい POCO に移行した。PICO についての解説、SFL 記述も web 上で公開する予定である。

- 本書はコンピュータアーキテクチャの入門書ではあるが、それでもこの 10 年間にコンピュータは進歩を続けた。この内容を取り込む必要があった。

本書は、プログラミング言語 (特に C 言語) に関する初歩的な知識とブール代数や順序回路設計等、論理設計に関する初歩的な知識を前提としているが、これらは電気情報系の大学あるいは高専の初年度に学習するごく簡単なレベルである。すなわち、本書は大学以上、高専上級生レベルの、コンピュータアーキテクチャの初学者を対象としている。しかし、これらの知識が全くない初学者も各章の例題や演習を実際に実行しながら自習すれば、充分内容が理解できると思う。

---

[3] 私自身翻訳に携わった [3] は本書と全く同じ目的意識に基づいた良書である。しかし、本書の初版本が出版されたのは 2001 年で、[3] よりもずっと以前であり、本書がこれに影響されて書かれたわけではない。また、コンピュータアーキテクチャのテキストについてのサーベイは web を参照されたい。

第 2

データパス：コンピュータで演算をする所

2.1　数の表現

まずはコンピュータ内の数の表現を復習しておこう。ご存じのとおり、ディジタルシステムでは数を 1 と 0 の 2 進数で表現する。これは 1 と 0 が H レベルと L レベルの二つのレベルで処理を行うディジタルシステムと本質的に良く適合するためである。

基本となるのは符号なし数、すなわちただの 2 進数である。一番右の桁が 1 の位であり、これを LSB(Least Significant Bit) という。一番左が 2^n の位であり、これを MSB(Most Significant Bit) という。

コンピュータでは長いビット数を扱うことが多いため、通常の 2 進表現では見にくいことがある。このような場合は、4 桁ずつ区切って、16 進数として表現する。この時、10 進数との混乱を避けるため、C 言語の書き方にならって頭に 0x を付けて表す場合が多い。ちなみに、かつては 8 進数や BCD(Binary Coded Decimal: 2 進化 10 進数) も用いられていたが、現在はほとんど使われないため省略する。

10 進	2 進	16 進	10 進	2 進	16 進
0	0000	0	8	1000	8
1	0001	1	9	1000	9
2	0010	2	10	1000	A
3	0011	3	11	1000	B
4	0100	4	12	1000	C
5	0101	5	13	1000	D
6	0110	6	14	1000	E
7	0111	7	15	1000	F

上記を符号無し数 (unsigned) と呼ぶ。ではマイナスの数を表現するためにはどのようにすれば良いだろうか？単純なアイディアは符号ビット s と符号無し数 (絶対値) を

組み合わせることで、s=1 で 1001 ならば-9、s=0 で 1001 ならば 9 とする方法である。これは可能だが一般的ではない。

マイナスの数を表現するのに普通に使われる方法は、2 の補数表現である。この方法には以下のメリットがある。(1) 一番上の桁で符号がすぐに判定できる。(2) マイナスの数を加算することで減算ができる。(3)+0 と-0 が生じない。(符号ビットと符号無し数の組み合わせだと+0, -0 ができてしまう)。このため、現在すべてのコンピュータで用いられている。

2 の補数とは、足した結果、すべての桁が 0 になり、桁上げが生じる数のことである。具体的には、(1) ある数の 1 と 0 をひっくりかえす。(2) 1 を足す。で、求めることができる。例えば、4 桁の数があるとするならば、下に示すように、-8 から 7 まで 16 個の数を表すことができる。

2 進	2 の補数	符号なし数	2 進	2 の補数	符号なし数
0000	0	0	1000	-8	8
0001	1	1	1001	-7	9
0010	2	2	1010	-6	10
0011	3	3	1011	-5	11
0100	4	4	1100	-4	12
0101	5	5	1101	-3	13
0110	6	6	1110	-2	14
0111	7	7	1111	-1	15

上の表も見てもわかるように、2 の補数を用いると一番上の桁が 1 ならば、それはマイナスの数になる。つまり、2 の補数表現では、MSB を符号ビットと呼ぶ。

2 の補数表現で m 桁の数の桁数を増やす必要が生じた場合は、MSB が 0 ならば 0 を、1 ならば 1 を補う。例えば、4 桁の’6’=0110 を 8 桁で表す場合、上に 0 を 4 つ補い 00000110 とする。しかし、’-6’=1010 を 8 桁で表す場合に同じことをすると、00001010=10 になってしまう。1 を 4 つ補い 11111010 とすれば、8 桁にしてもちゃんと-6 になる。このように桁数を増やす方法を符号拡張 (sign extension) と呼ぶ。符号拡張は後に命令コード中に頻繁に表れて、皆さんを悩ますことになる。

例題 2.1

下の bit 列を 16bit に符号拡張し、2 進数で示せ。

1. 0011
2. 10010010
3. 0x93

答

1. 0011: 頭に 0 を付けて 0000000000000011

2. 10010010: 符号拡張して 1111111110010010
3. 0xa3: 10100011 なので符号拡張して 1111111110100011

例題 2.2

8bit の符号付き数で、(1) -12, (2) -122 を示せ。

答

1. 12 は 8 ビットの 2 進数で 00001100 となる。1 の補数は 11110011 であり、これに 1 を足して 11110100 となる。
2. 122 は 8 ビットの 2 進数を求めることがまず面倒であるが、2 で割った余りを順に下の桁から並べて行けば良い。122/2=61..0, 61/2=30...1, 30/2=15...0, 15/2=7...1, 7/2=3...1, 3/2=1...1 となるので、01111010 である。後は同様に 1000101+1=10000110 となる。

1 より小さい桁が必要な場合、固定小数点 (Fixed Point) 数が用いられる場合がある。この方法では小数点の場所をあらかじめ決めておくだけで、数字自体の記述は、通常の数の表現と同じである。

より桁数の大きな科学技術計算には浮動小数点 (Floating Point) 数が用いられる。現在、IEEE により標準化されており、32 ビットの単精度、64 ビットの倍精度が存在する。図 2.1 に示すように、それぞれ仮数部と指数部を持っており、(仮数部) × (2 の指数部乗) で数を表す。これ自体は簡単だが、浮動小数演算でいつも問題となるのは、仮数部に入らなくなった桁を省略して、指数部に繰り込む際に生じる「丸め」の処理である。この処理は繁雑で、ハードウェアを実装する際に常に悩ましい問題となる。

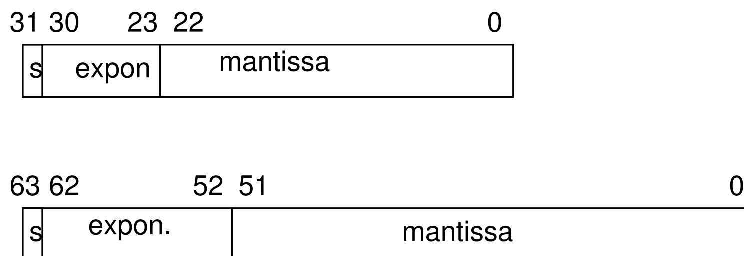


図 2.1: 浮動小数

さらに、コンピュータで扱う 2 進ビット列は、数を表すだけとは限らない。文字コードなど、数以外のデータを表す場合も多い。文字コードとしては英数字を表すコード

として 8bit の ASCII コードが、より多くの文字を表すのには 16bit の Unicode が一般的だが、他にも多種多様なコードが使われる。

さて、ここで覚えておいて欲しい重要な概念がある。コンピュータで扱うデータは、それ自体は意味を持っていない。1/0 の列によって表されたデータは、命令なのか、符号無し数なのか、符号付き数なのか、区別を持っていない。それがどんな意味を持つかは、扱う側、つまりコンピュータのハードウェアやプログラマの解釈次第なのである[1]。

演習 2-1

下の bit 列を 16bit に符号拡張し、2 進数で示せ。またこれを符号無し数と考えて 16 進数で表記するとどのようになるかを示せ。

1. 11000110
2. 0x3D
3. 0x80

演習 2.2

8bit の符号付き数で、(1) -17, (2) -115 を示せ。

2.2 ALU(Arithmetic Logical Unit)

さて、コンピュータの中で行う演算とはどのようなものだろう。まず加減乗除の四則演算が思い浮かぶだろう。その他代表的な演算は以下の通りである。例では A=10110110 と B=11100010 を入力とする。

- 論理演算: 各桁の論理演算を行う。
  - 論理積 (AND): A,B 各桁の論理積を出力する。例では A AND B = 10100010 となる。
  - 論理和 (OR): A,B 各桁の論理和を出力する。例では A OR B = 11110110 となる。
  - 排他的論理和 (XOR): A,B 各桁の排他的論理和を出力する。排他的論理和とは 0 XOR 0 = 0, 0 XOR 1 = 1, 1 XOR 0=1, 1 XOR 1 =0、すなわちビットが異なっていれば 1 となる。例では A XOR B = 01010100 となる。

[1] これが原因で、処理の信頼性が低くなると考えた一部の研究者は、データに意味を付与するタグ付きアーキテクチャというのを考えたが、苦労した割にメリットが少ないため、今の所、成功していない。

- 反転 (NOT): A 各桁の反転を出力する。例では 01001001 となる。

- n ビットシフト: 各桁を n 桁分ずらして (シフトして) 出力する。
  - 左シフト (Shift Left:SL): 左方向にずらす。n 桁シフトすると 2^n 倍したことになる。空いた最下位 (最も右) の桁には 0 を詰める。1bit シフトの場合、01101100 となる。
  - 右シフト (Shift Right:SR): 右方向にずらす。n 桁シフトすると $1/2^n$ 倍したことになる。ここで右シフトには 2 種類ある。
    * 論理シフト (SRL: Shift Right Logical)：空いた最上位 (最も左) の桁には 0 を詰める。例では 1bit シフトの場合、01011011 となる。
    * 算術シフト (SRA:Shift Right Arithmetic)：空いた最上位 (最も左) の桁に符号ビットを詰める。つまり符号を保持したままシフトする。例では 1bit シフトの場合、11011011 となる。

  上記の例を図 2.2 に示す[2]。

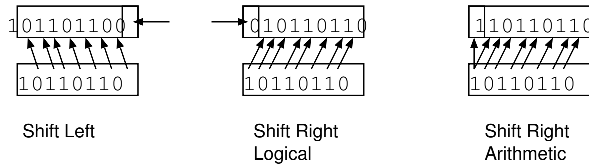


図 2.2: シフト操作

演習 2.3

8bit データについて A=10100110, B=01100100 とする。以下の演算の結果を求めよ。

1. A AND B
2. A OR B
3. A XOR B
4. SL A (1bit)
5. SRA A (1bit)
6. SRA B (1bit)

[2] SLA Shift Left Arithmetic も定義可能だが、あまり一般的ではない

コンピュータでは、上記の演算を行う装置を一まとめにして扱う。これを、ALU(Arithmetic Logic Unit) と呼ぶ。乗算と除算は、やや複雑で 1 クロックで終わらせるとが難しいため、基本的な ALU の中には入れない場合が多い。

ALU は図 2.3 に示すように a, b 二つのデータ入力とコマンド入力 s、出力として y を持っている。コマンド入力 s の内容に応じて、a,b の入力に対して所定の演算を行って、y に出力するのが ALU の役目である。

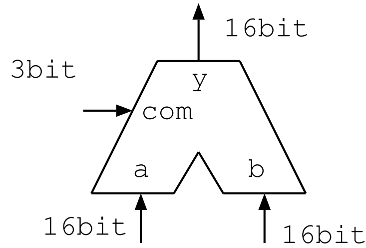


図 2.3: ALU のモデル

ある ALU で 8 つの演算が可能な場合、3 ビットの com の値によってその内容を切り替える。例えばここでは以下のように決めよう。

s	出力	記号
000	a	THA
001	b	THB
010	a AND b	AND
011	a OR b	OR
100	a を 1 ビット左シフト	SL
101	a を 1 ビット論理右シフト	SR
110	a+b	ADD
111	a-b	SUB

y に a,b をそのまま出すのは奇妙な感じがするが、実はこれがないと周辺回路が面倒になるので、たいていの ALU はこの機能を持っている。この命令をここでは THA(スルー A)、THB(スルー B) と呼ぶ。

さて、もちろん ALU の中には、加算器や AND のアレイが入っているのだが、最初は中身については考えず、とにかく com の値によって決った演算をやってくれることにする。このような考え方をブラックボックス化と呼ぶ。中身を知らなければ納得できない、という人は第 4 章に解説があるのでそちらをご覧いただきたい。

2.3　計算をするということ

この ALU で X+Y を計算する場合、両方の入力に X と Y を入れ、s に 110 をセットすれば良い。では、もう少し複雑な

```
X+Y-W+Z
```

などの計算を ALU でどのように行えば良いのだろう。図 2.4 のように複数の ALU を組み合わせればこの演算を実現することができる。しかし、このようなことをしていくと、式が長くなると、どんどん利用しなければならない ALU 数が増えてしまう。

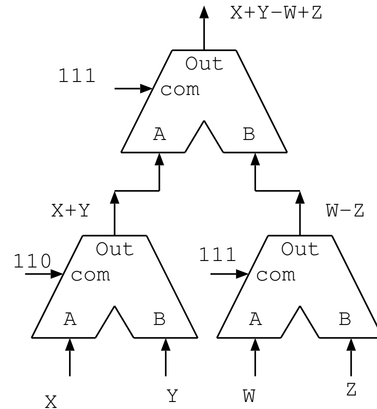


図 2.4: ALU の接続

そこで、数を記憶しておく場所を設ける。これをレジスタ (register) あるいは置数器と呼ぶ。抵抗器の resiter ではないので注意。店のレジと語源は同じである。これも中身が気になる人は第 4 章を参照されたい。ここでは、入力したクロックが L から H に変化した際に、入力データを記憶する装置だと考えてほしい。

図 2.5 に示すように、ALU の片方の入力にレジスタの出力を入れ、レジスタの入力に ALU の出力を接続する。すなわち、計算結果をレジスタに格納し、それを再び ALU の入力として計算に使えるようにするわけである。

X+Y-W+Z をこのデータパスで実行するには以下のようにすれば良い。ここでは、一行分計算を実行した所でクロックを変化させてレジスタに値をセットすると考える。

s	b	クロック変化後のレジスタ
001	X	X
110	Y	X+Y
111	W	X+Y-W
110	Z	X+Y-W+Z

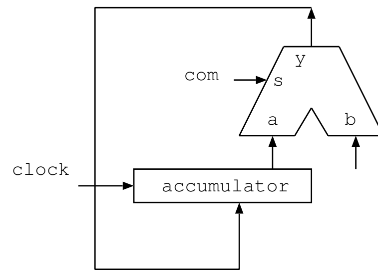


図 2.5: 最も簡単なデータパスの構成

このようにすると、ひとつの ALU を使い回すことが可能になる。計算結果はレジスタに置かれて、更新される。このレジスタは結果が積み重なる場所となり、アキュムレータ (accumlator) と呼ばれる。電卓で計算をする時、我々は無意識にこのアキュムレータの考え方を使っている。また、この一行め (com=001) は、最初に計算すべき値をアキュムレータにセットする役割を果たす。このように外部からレジスタに値をセットする操作をロード (load) 操作と呼ぶ。また、コンピュータで演算を行い、その途中結果を格納しておく部分をデータパス (datapath) と呼ぶ。

演習 2.4

図 2.5 のデータパスで X-Y+Z を計算したい。com と b にどのような値を与えれば良いか。

2.4　メモリを使った計算

これまでのデータパスでは、アキュムレータを使うことで、演算した結果に対して次々に演算を施して行くことができた。しかし、以下のような演算はスムーズに行うことができない。

```
(SL X)+(SL Y)
```

上の式ではまず X をアキュムレータに入れて 1 ビット左シフトし、その結果をどこかにとっておいて、それから Y を左シフトして加えなければならない。つまり、結果をとっておく場所が必要であり、これを可能にするのがメモリである。電卓でもメモリ付きになっているのが普通で、このような枠組みは計算には必須である。

ここでは、メモリは単純に図 2.6 に示す表と考えて欲しい。この表は、各行に番号（アドレス）が振られており、アドレスの値によって、対応する行の中身（データ）を読み出すことができる。この場合、アドレスに 0 を設定すれば 1001 が、1 を設定すれば 1100 が読み出され、Dataout に出力される。

また、書き込みの場合、入力にデータを置き、アドレスを設定し、書き込み制御信号 we(write enable) を H にし、クロックを立ち上げると、そのアドレスにデータが書き込まれる。

この図は簡単で、幅が 4 ビットで深さが 2^4=16 のメモリということになり、全容量は 64bit である。深さが 16=2^4 であるため、番地を識別するアドレス線も 4 本で良い。もちろん、実際のメモリはもっとずっと大きく、アドレスが 32bit-48bit、データは 32bit-64bit で膨大な情報量を格納する。しかし原理は同じである。アドレスが m bit あれば、2^m 行分指定することができ、容量の大きいメモリほど、アドレスの本数も多い。ここでは、幅が 16bit、深さが 256 のメモリを考える。この場合、アドレスの本数は 8 本となる。

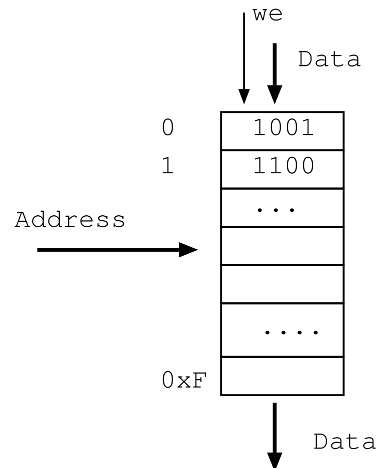


図 2.6: メモリの構成

それでは、このメモリを前回のデータパスにつないでみよう。この様子を図 2.7 に示す。

この構成では、データパスの入力にはメモリの出力、アキュムレータ出力にはメモリの入力が接続されている。すなわち、データを外から与えてやる代わりにメモリ中にあらかじめ格納してデータに対して演算を行い、途中結果もメモリに格納することができる。

2.4.1　命令の基礎

さて、このデータパスに外部から指示を命令の形で与えて計算をやらせよう。

まず演算を行うためには、メモリ中に存在するデータをアキュムレータに格納する必要がある。コンピュータでは、このための命令を、

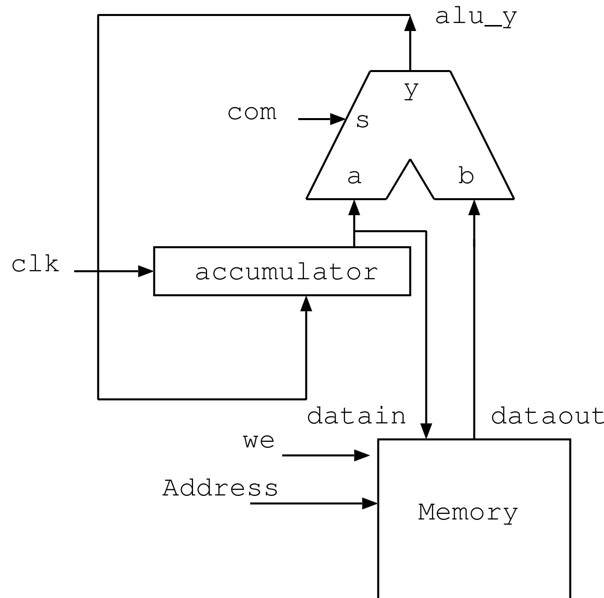


図 2.7: メモリ付きデータパス

（持って来い）　0 番地のデータを

の形で表す。日本語とは語順が逆なのは、命令を最初に作った人が英語を元に考えたからである。（持って来い）の部分、つまり操作を示す部分を命令コード (opcode, オペコード、オプコード) と呼ぶ。0 番地のデータをに対応する操作対象を示す部分をオペランド (operand) と呼ぶ。

メモリからレジスタにデータを持って来いという意味は先に述べたように、英語で Load である。（これは習慣で決っている。get でも take でもない。move という場合もある。）命令としては Load と直接英語で書かずに LD という記号で書く。これをニーモニックと呼ぶ。本来、ニーモニックは命令セット毎に違うのだが、長年の蓄積で、ほぼ共通のコードを使っている。したがって LD と言ったら、情報系の人ならば万国共通で、メモリをレジスタに持ってくる命令だと分かる。オペランドの部分はここでは、単にメモリの番地に相当する。したがって、0 番地のデータを持ってくる命令は

```
LD 0
```

と書く。これは、人間に分かりやすい形であって、アセンブリ言語表記あるいはアレンブラ表記と呼ぶ。コンピュータのハードウェアは、1/0 のパターンに対応付けたものを実行する。この場合、LD に 0001、アドレスはそのまま 0000 を割り当てると命令は以下のようになる。

```
0001 00000000 // LD 0 0番地の内容を読み出してアキュムレータに格納
0001 XXXXXXXX // LD X X番地の内容を読み出してアキュムレータに格納
```

これが機械語である。LD 命令が実行されると、オペランドがメモリのアドレスに与えられ、読み出されたデータは ALU の B 入力から Y 入力に抜ける。このために ALU のコマンドを 001 にセットする。そして、アキュムレータに書き込んでやる。

LD の逆の操作、すなわち、アキュムレータからメモリにデータを書き込む操作を Store と呼ぶ。これもニーモニックでは ST と書いてやる。ここでは、ST には 1000 を割り当てることにしよう。オペランドは以前と同様でメモリのアドレスである。

```
1000 00000000 // ST 0  0番地にアキュムレータの内容を書き込む
1000 XXXXXXXX // ST X  X番地にアキュムレータの内容を書き込む
```

ST 命令実行時には、メモリのアドレスにはオペランドを与え、アキュムレータから直接引かれているデータ線を用いて、データをメモリに書き込む。

2.4.2 演算命令

典型的な命令は演算命令である。これは、アキュムレータの内容とメモリの指定されたアドレスのデータを演算して、答えをアキュムレータに格納する。ここで使う ALU は、6 つの演算操作ができるので、以下の 6 命令が定義できる。命令コードはどう決めてもいいのだが、ここでは下 3 ビットを ALU の com と一致させ、最上位ビットを 0 として決めよう。オペランドは以前通りメモリの番地である。ここで、SL(左シフト、shift left) と SR(右シフト、shift right) は、B 入力は使われないため、アドレスには何を入れても良い。(ドントケア)

```
0010 XXXXXXXX // AND X (X番地とアキュムレータの内容をAND)
0011 XXXXXXXX // OR  X (X番地とアキュムレータの内容をOR)
0100 -------- // SL  (アキュムレータの内容を左1ビットシフト)
0101 -------- // SR  (アキュムレータの内容を右1ビット論理シフト)
0110 XXXXXXXX // ADD X (X番地とアキュムレータの内容を加算)
0111 XXXXXXXX // SUB X (アキュムレータの内容からX番地の内容を減算)
```

実は LD はこれらの演算命令の一種であったことに気づいただろうか？それでは、今まで定義した命令を使って、演算処理を実行してみよう。

例題 2.3

X,Y,Z がそれぞれ 0,1,2 番地に格納されているとする。

```
Z=X+Y
```

は以下のような命令を順に実行すれば計算できる。

```
LD 0  0001 00000000
ADD 1 0110 00000001
ST 2  1000 00000010
```

では、以下はどうであろうか？

例題 2.4

```
Z= X + Y - Z
```

これは、A+B の結果がアキュムレータに残っているのでそれがそのまま使えるので簡単である。これがアキュムレータの良い点である。

```
LD 0   0001 00000000
ADD 1  0110 00000001
SUB 2  0111 00000010
ST 2   1000 00000010
```

しかし、最初に示した演算では、アキュムレータの値を直接利用することはできない。

```
(X<<1)+(Y<<1)
```

この場合 X の 1 ビットシフトを計算し、その結果をどこかにとっておく必要がある。この中間結果をとっておく場所をここでは 2 番地としよう。

```
LD 0  0001 00000000
SL    0100 00000000
ST 2  1000 00000010
LD 1  0001 00000001
SL    0100 00000000
ADD 2 0110 00000010
ST 2  1000 00000010
```

2 番地に一度中間結果を入れてから Y を LD し、シフトをした後、加算をする。入れる。これは、ちょうどメモリ付きの電卓を使う際に中間結果をメモリにとっておくのと似ている。

演習 2.5

0 番地,1 番地,2 番地にそれぞれ A,B,C が格納されている。(A-B) OR (B+C) の演算を行い、3 番地に答えを格納するプログラムをアセンブラ表記、機械語の両方で示せ。

2.5 Verilog での記述

2.5.1 加算器の記述

Verilog では、一つのまとまりをもった回路をモジュールと呼ぶ。Verilog の記述は、まずモジュールの名前と入出力を定義することから始まる。

```
module 名前 (
  input 入力1,
  input 入力2,
  output 出力1,
  output 出力2 );
```

この定義の方法は、C 言語の関数定義とちょっと似ているが、最後にセミコロン (;) が付いて文の形になっている点が違っている。まず a, b の２入力を持ち、s の出力を持つ 1bit の加算器の記述を示す。これは以下のように非常に簡単である。

```
/* 1 bit adder */

 module adder (
 input a, b, output s );

  assign s = a + b  ;  // add a b

endmodule
```

コメントは C 言語同様２種類 (/* */と//) が可能である。さて、加算器は、後の章に解説するようにリプルキャリアダーからプリフィックスアダーまで様々な構成があるが、Verilog でも VHDL でもここは抽象化して単に'+' で表す。どのような加算器を使うかは、後に論理合成の段階で、必要なコストと性能を考えて、自動合成用の CAD(Computer Aided Design) プログラムが選択する。設計者はこの選択時に必要な指示を与えてやる。s=a+b; の前に assign が付いているのに注意されたい。assign 文は、ある回路の出力ある端子に出力する、あるいは接続することを意味する。ここでは加算の結果を s に出力する、あるいは加算結果出力を s と接続することを示している。最後にモジュール文は endmodule で終わるが、ここは; は付かない。

この記述は adder.v というファイルに入れておく。C 言語のプログラムの拡張子が.c であるのと同様、Verilog-HDL 記述の拡張子は.v である。

次にこれをシミュレーションしてみよう。第 1 章に紹介したように、Verilog は元々シミュレーション記述用の言語であり、シミュレーションに対する指示の方法は多様で、理解するのが大変である。ここはまずは動かしてみるのが良いと思う。

下の test.v は上記の加算器のモジュールをテストするシミュレーション用の Verilog 記述である。この記述は、ハードウェアの回路を表すのではなく、テストするための入力の与え方と、出力の表示の仕方を示す。このような記述をテストベンチと呼ぶ。

Verilog を習得する時にやっかいなのは、テストベンチ中には結構難しい文法が必要とされるのに、これを書かないとテスト対象のモジュールがいかに簡単なものであっても全くシミュレーションができない点である。これを避けるためにある種の CAD では、パラメータを与えてテストベンチを自動生成する機能を持つ。

で、ここでは、テストベンチの解説は一応しておくが、重要な文法は後でまた解説することにして、直感的に理解する程度で深く突っこまないで先に進みたい。実の所、テストベンチの作り方は定型的なので、一定のパターンをマスターしてこれを場合に応じて修正すれば、かなりの範囲で応用が効く。どうか細部にこだわらず、大体の意味を掴んでほしい。

```
/* test bench */
`timescale 1ns/1ps

module test;
   parameter STEP = 10;
   reg ina, inb;
   wire outs;
   adder adder_1(.a(ina), .b(inb), .s(outs));
   initial begin
      $dumpfile("adder.vcd");
      $dumpvars(0,adder_1);
      ina <= 1'b0;
      inb <= 1'b0;
     #STEP
      $display("a:%b b:%b s:%b", ina, inb, outs);
      ina <= 1'b0;
      inb <= 1'b1;
     #STEP
      $display("a:%b b:%b s:%b", ina, inb, outs);
      ina <= 1'b1;
      inb <= 1'b0;
     #STEP
      $display("a:%b b:%b s:%b", ina, inb, outs);
      ina <= 1'b1;
      inb <= 1'b1;
     #STEP
```

```
      $display("a:%b b:%b s:%b", ina, inb, outs);
      ina <= 1'b0;
      inb <= 1'b0;
     $finish;
   end
endmodule
```

最初に出てくるタイムスケール文は、シミュレーションの基本時間、時刻刻みの最小時間を定義する。

```
`timescale 1ns/1ps
```

ここでは前者は 1ns(ナノ=10^{-9} 秒)、後者は 1ps(ピコ=10^{-12} 秒) とした。
次に、パラメータ文を使って今回シミュレーションを行う 1 ステップの時間を定義する。

```
parameter STEP = 10;
```

このパラメータ文は実行時に外部から値を設定できるのがミソで、便利なので良く用いられる。ここでは 10nsec を 1 ステップとする。
次に加算器に外部から値を与えるための入力、値を取り出すための出力用の端子を定義してやる。ここで、入力には、値を保持するためレジスタ (reg) として宣言し、出力は単に値を取り出せば良いので端子名を宣言 (wire) してやる。

```
   reg ina, inb;
   wire outs;
```

次に、adder モジュールの実体を定義して入出力を接続する。先にモジュール文で adder.v 内に定義した加算器のモジュール名を最初に指定し、次に実体名 (ここでは adder_1) を宣言して実体を生成する。一つのモジュールから多数の実体を別の名前で生成することができる。括弧内で、入出力の接続を行う。. の後にモジュール内の入出力名、() の中に接続する外部端子の名前を記述する。ここでは adder 内で宣言された a 入力に外部の ina を、b 入力に外部の inb を、s 出力に外部の outs を接続する。

```
   adder adder_1(.a(ina), .b(inb), .s(outs));
```

これで宣言が終わりで、次は initial 文で、シミュレーションの動作を記述する。initial 文は、開始後一回だけ begin から end までの文が順に実行される。

```
      $dumpfile("adder.vcd");
      $dumpvars(0,adder_1);
```

最初に のついた二つの文が実行される。*Verilog* 内で は、シミュレーションを制御する上で必要な特殊なタスク (操作) を示す。この dumpfile と dumpvars では波形ビュアーで見るためのファイル名 (adder.vcd) と、信号を記録する範囲が指定されている。

次にいよいよシミュレーションを開始する。

```
     ina <= 1'b0;
     inb <= 1'b0;
    #STEP
     $display("a:%b b:%b s:%b", ina, inb, outs);
     ina <= 1'b0;
     inb <= 1'b1;
    #STEP
     $display("a:%b b:%b s:%b", ina, inb, outs);
     ina <= 1'b1;
     inb <= 1'b0;
    #STEP
     $display("a:%b b:%b s:%b", ina, inb, outs);
     ina <= 1'b1;
     inb <= 1'b1;
    #STEP
     $display("a:%b b:%b s:%b", ina, inb, outs);
     ina <= 1'b0;
     inb <= 1'b0;
    $finish;
```

ina, inb への値の代入は、後に解説するノンブロック代入文 <= で行っている。これで 0,1 が順に設定される。#STEP は、10nsec の時間経過を示す。すなわち、以下の文は、

```
     ina <= 1'b0;
     inb <= 1'b0;
    #STEP
     $display("a:%b b:%b s:%b", ina, inb, outs);
```

まず、ina, inb に 0 を設定し、10nsec 時間が経過したら、ina, inb, outs を表示する、という意味である。さて、Verilog では定数を以下のように表現する。

<ビット幅>'<基数><数値>

基数は

- b : 2 進数
- h : 16 進数
- o : 8 進数

である。なにも書かないと 10 進数になるので、数を直接書く際は、必ず桁数をはっきりさせて 2 進数、あるいは 16 進数で書くことをお勧めする。ここで 1'b0 は 1bit の 0 を、1'b1 は 1bit の 1 をそれぞれ示す。

```
$display("a:%b b:%b s:%b", ina, inb, outs);
```

display 文はその名の通り、ina,inb,outs の値を表示するためのものである。display 文の記述は C 言語の printf に似ているが、2 進数を表示するための%b を持っている。また、自動的に改行が入る。以下、入力を変えて、それぞれ 10nsec 時間を経過させて結果を出力していく。最後は finish 文でシミュレーションを終了する。

2.5.2 シミュレーションの実行

では、シミュレーションをしてみよう。第 1 章で紹介した ikarus verilog をインストールしてある環境を想定する。

```
iverilog test.v adder.v
```

で、シミュレーションの実行形が生成される。シミュレーション自体は、

```
vvp a.out
```

で実行される。ここでは最初は a 入力、次は b 入力、最後に加算結果が出力される。a.out の名前が気になる方は、

```
iverilog test.v adder.v -o adder
vvp adder
```

としても良い。

```
a:0 b:0 s:0
a:0 b:1 s:1
a:1 b:0 s:1
a:1 b:1 s:0
```

が出力される。次に波形を見てみよう。テストベンチで記述した波形データを入れておくファイルを指定して立ち上げる。

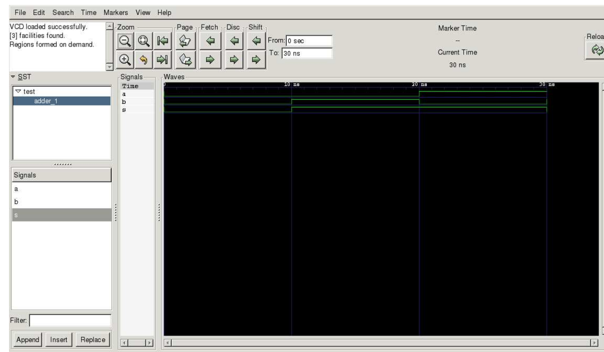


図 2.8: gtkwave による波形表示

```
gtkwave adder.vcd
```

左側の SST と書いてある所に test とテストベンチのモジュール名が表示されるのでそこをクリックすると adder_1 という加算器の実体名が表示される。ここをクリックすると信号名が Signals ウインドウに表示される。ここで、見たい信号をクリックしてから Append をクリックすると、Waves ウインドウに波形が表れる。gtkwave を始めて使う時に戸惑うのは、Waves の設定が最小時刻刻みになっているため、最初のきわめて一部しか表示されない点である。Zoom の所の虫眼鏡の-を連打して表示スケールを調整すると、見たい範囲で波形が見られる。gtkwave は優れた GUI を持っているため、勘で結構使える。ここではあまり解説しないが、色々機能を試して欲しい。ちなみに終了の際は File→Quit で確認用の小ウインドウが出てくるので、ここで Yes を押してやる。

演習 2.6

adder の記述を書き換えて、減算、AND、OR などにして試してみよ (減算は実は加算と同じになる)。

2.5.3　ALU の記述

次にやや難しい ALU を記述してみる。まず、1 ビットはあんまりなのでビット数を拡張することにしよう。ハードウェアでは配線を何本か束にして扱う場合が多い。これをバス (Bus) と呼び、Verilog では大括弧でくくって，MSB:LSB の形でその範囲を表わす。本書では簡単のため LSB は常に 0 としよう。

入出力 16 ビット、コマンド 3 ビットの ALU は次のように定義される。

```
module alu (
```

```
  input [15:0] a, b,
  input [2:0] com,
  output [15:0] y );
```

次に中身を定義する。ALU の場合、入力が特定の条件ならば入力同士の演算の結果を出力する。このような場合、Verilog では以下のように記述する。

```
assign 出力 = 条件? 式 1  :  式 2;
```

条件が真であれば式 1 の結果が出力され、そうでなければ式 2 の結果が出力される。条件を複数書くこともできる。これは case 文と似ている。条件は前から順にチェックされるので、優先順位は先に書いた方が高いことになる。下の例では、条件 1-3 がどれも真でなれば、コロンの後に書いた式 4 の結果が出力される。

```
assign 出力 = 条件 1? 式 1 :
              条件 2? 式 2 :
              条件 3? 式 3 : 式 4;
```

今回の ALU は、以下のような機能を定義した。

com	出力	記号
000	a	THA
001	b	THB
010	a AND b	AND
011	a OR b	OR
100	a を 1 ビット左シフト	SL
101	a を 1 ビット論理右シフト	SR
110	a+b	ADD
111	a-b	SUB

これを Verilog 記述すると、以下のようになる。

```
 assign y =  com==3'b000 ? a:
             com==3'b001 ? b:
             com==3'b010 ? a & b:
             com==3'b011 ? a | b:
             com==3'b100 ? a<<1:
             com==3'b101 ? a>>1:
             com==3'b110 ? a + b: a - b ;
```

Verilog は、表 2.1 に示す演算を記述できる。ここでは AND, OR, 左右のシフト、加算、減算を使っている。AND, OR では、16 ビットの入力のそれぞれの桁同士で演算が行われ、答えも 16 ビットになる。

ここで、論理演算は、C 言語で用いるのと同様に、条件の論理的な真偽に対する演算を行い、結果は 1 ビットの値となる。ここでは比較演算子==を使っている。比較対象は 3 ビットの 2 進数なので、3'b000 と書く。

演算子には優先順位があるので注意されたい。表 2.2 で上位にあるもの程強い。

表 2.1: Verilog の基本的演算子

<table>
<tr><td>ビット演算</td><td>~</td><td>NOT</td></tr>
<tr><td></td><td>&</td><td>AND</td></tr>
<tr><td></td><td>~&</td><td>NAND</td></tr>
<tr><td></td><td>|</td><td>OR</td></tr>
<tr><td></td><td>~|</td><td>NOR</td></tr>
<tr><td></td><td>^</td><td>Ex-OR</td></tr>
<tr><td></td><td>~^</td><td>Ex-NOR</td></tr>
<tr><td>シフト演算</td><td><<</td><td>左シフト</td></tr>
<tr><td></td><td>>></td><td>右シフト</td></tr>
<tr><td>等号、関係演算</td><td>==</td><td>等しい</td></tr>
<tr><td></td><td>! =</td><td>等しくない</td></tr>
<tr><td></td><td>===</td><td>等しい (x,z も比較)</td></tr>
<tr><td></td><td>! ==</td><td>等しくない (x,z も比較)</td></tr>
<tr><td></td><td><</td><td>小さい</td></tr>
<tr><td></td><td><=</td><td>以下</td></tr>
<tr><td></td><td>></td><td>大きい</td></tr>
<tr><td></td><td>>=</td><td>以上</td></tr>
<tr><td>算術演算</td><td>+</td><td>加算</td></tr>
<tr><td></td><td>−</td><td>減算</td></tr>
<tr><td></td><td>*</td><td>乗算</td></tr>
<tr><td></td><td>/</td><td>除算</td></tr>
<tr><td></td><td>%</td><td>剰余算</td></tr>
<tr><td>論理演算</td><td>!</td><td>否定</td></tr>
<tr><td></td><td>&&</td><td>論理積</td></tr>
<tr><td></td><td>||</td><td>論理和</td></tr>
</table>

2.5.4 define 文を使って格好を付ける

先に示した ALU の記述はきちんと動作し、合成もできるが、あまり格好良いと見なされない。これは、コード中に 3'b000 とか 15:0 など定数が直に記述されているからである。これらのハードウェアの記述に必要な定数 (マジックナンバーと呼ばれることもある) は、直接コード内に書かない方が良い。これは、コードが読みにくくなることと、仕様変更があった際にコード中のあちこちを変更しなければならないためである。そこで、Verilog では C 言語と同様 define 文でこれらの定数をあらかじめ定義しておく。ただし C 言語と違って define 文の前と定義された文字列を使う場合はバックシングルコーテーション (`) を使う。下記は 4 種類の演算のみこの変換を行った例である。

表 2.2: 基本的演算子の優先順位

<table>
<tr><td>高い</td><td>^ ~ !&| + - (単項演算子)</td></tr>
<tr><td></td><td>*/ %</td></tr>
<tr><td></td><td>+ -</td></tr>
<tr><td></td><td><<>><<<>>></td></tr>
<tr><td></td><td><<=>>=</td></tr>
<tr><td></td><td>==! ====! ==</td></tr>
<tr><td></td><td>& ~& ~ ~^</td></tr>
<tr><td></td><td>| ~|</td></tr>
<tr><td></td><td>&&</td></tr>
<tr><td></td><td>||</td></tr>
<tr><td>低い</td><td></td></tr>
</table>

```
‘define DATA_W 16  // bit width
‘define SEL_W 3 //control width
‘define ALU_THA ‘SEL_W’b000
‘define ALU_THB ‘SEL_W’b001
‘define ALU_AND ‘SEL_W’b010

module alu (
  input [‘DATA_W-1:0] a, b,
  input [‘SEL_W-1:0] s,
  output [‘DATA_W-1:0] y );
  assign y = s==‘ALU_THA ? a:
             s==‘ALU_THB ? b:
             s==‘ALU_AND ? a & b: a + b ;
endmodule
```

演習 2.7

上記の記述を全ての演算について行え。

define 文は parameter 文よりも変更が少なく、基本的に決めたら滅多に変えないものを定義するのに使う。ただ両者の特徴は微妙で、これについては、web のうんちくを参照されたい。

2.5.5　ALU のテストベンチ

下の test.v は ALU をテストするシミュレーション用のテストベンチである。基本的なやり方は、加算器用と同じである。

```
/* test bench */
‘timescale 1ns/1ps
‘define DATA_W 16  // bit width
‘define SEL_W 3 //control width
‘define ALU_THA ‘SEL_W’b000
‘define ALU_THB ‘SEL_W’b001
‘define ALU_AND ‘SEL_W’b010
‘define ALU_ADD ‘SEL_W’b110

module test;
parameter STEP = 10;
   reg [‘DATA_W-1:0] ina, inb;
   reg [‘SEL_W-1:0] sel;
   wire [‘DATA_W-1:0] outs;

   alu alu_1(.a(ina), .b(inb), .s(sel), .y(outs));
   initial begin
      $dumpfile("alu.vcd");
      $dumpvars(0,alu_1);
      ina <= ‘DATA_W’h1111;
      inb <= ‘DATA_W’h2222;
      sel <= ‘ALU_THA;
     #STEP
      $display("a:%h b:%h s:%h y:%h", ina, inb, sel, outs);
      sel <= ‘ALU_THB;
     #STEP
      $display("a:%h b:%h s:%h y:%h", ina, inb, sel, outs);
      sel <= ‘ALU_AND;
     #STEP
      $display("a:%h b:%h s:%h y:%h", ina, inb, sel, outs);
      sel <= ‘ALU_ADD;
     #STEP
      $display("a:%h b:%h s:%h y:%h", ina, inb, sel, outs);
     $finish;
```

```
   end
endmodule
```

今回は、桁数が 16bit になっており、a,b 入力には１６進数を、com には２進数を使って定義をしている点に注意されたい。16 進表示は C 言語の 0x ではなく h を使う。

```
      a <= 16'h1111;
      b <= 16'h2222;
      com <= 3'b000
```

同様に、display 文も 16 進表示は%x でなくて、%h である。では、加算器と同様にシミュレーションをしてみよう。

```
iverilog test.v alu.v
```

で、シミュレーションの実行形が生成される。シミュレーション自体は、

```
vvp a.out
```

で実行される。ここでは最初は a 入力、次は b 入力、最後に加算結果が出力される。

```
a:1111 b:2222 com:0 y:1111
a:1111 b:2222 com:1 y:2222
a:1111 b:2222 com:6 y:3333
```

演習 2.7

AND,OR, 減算のシミュレーションを行い、結果を確認せよ。

2.5.6 データパスの記述

図 2.5 のデータパスを記述してみよう。このためには、レジスタを記述する必要がある。レジスタはクロックに同期してデータを格納する点でやや特殊な素子であり、ALU での論理演算とは雰囲気が違っている。

Verilog でレジスタを宣言する際はそのものずばり

```
reg [15:0] acum;
```

とすれば良い。ここでのアキュムレータは 16 ビットのデータを記憶するので、入出力同様バスの形にして宣言する。図 2.5 を見ると、信号線を区別するため alu_y などの名前が付いている。Verilog では、wire 文で信号の名前を宣言する。

```
wire [15:0] alu_y;
```

レジスタと違って、alu_y は ALU の出力データそのものを示していて、データを記憶することはできない。ALU 同様、直接数字を書かないことにして、ここでは、

```
reg [‘DATA_W-1:0] accum;
wire [‘DATA_W-1:0] alu_y;
```

と記述する。ここで ALU の時と違って、この define 文を 1ヶ所にまとめて def.h としておく。そして

```
‘include ‘‘def.h’’
```

とすれば、様々なファイルで同じ定義を共有することができる。この辺の発想は、C 言語と同じである。ただし、include 文の先頭にはシングルバックコーテーション (‘) が付く点に注意されたい。def.h の中身は以下の通りである。

```
‘define DATA_W 16
‘define SEL_W 3
‘define ADDR_W 8
‘define DEPTH 256
‘define ALU_THA ‘SEL_W’b000
‘define ALU_THB ‘SEL_W’b001
‘define ALU_AND ‘SEL_W’b010
‘define ALU_ADD ‘SEL_W’b110
‘define ALU_SUB ‘SEL_W’b111
‘define ENABLE 1’b1
‘define DISABLE 1’b0
‘define ENABLE_N 1’b0
‘define DISABLE_N 1’b1
```

では、全体の記述を見てみよう。

```
‘include "def.h"
module datapath(
input clk, input rst_n,
input [‘DATA_W-1:0] datain,
input [‘SEL_W-1:0] com,
output [‘DATA_W-1:0] accout);

reg [‘DATA_W-1:0] accum;
wire [‘DATA_W-1:0] alu_y;
```

```
assign accout = accum;
alu alu_1(.a(accum), .b(datain), .s(com), .y(alu_y));

always @(posedge clk or negedge rst_n)
begin
    if(!rst_n) accum <= ‘DATA_W’b0;
    else accum <= alu_y;
end
endmodule
```

このデータパスは、アキュムレータの中身をメモリに書き込むために外部に出さなければならない。このような場合、assign 文を用いる

```
assign accout = accum;
```

この文は、accum のデータを dataout に出力すると考えてもよいし、accum を出力 dataout に接続すると考えても良い。さて、ALU は別モジュールとして宣言されているので、これを部品として使う。これが以下の記述である。

```
alu alu_1(.a(accum), .b(datain), .com(com), .y(alu_y));
```

alu がモジュール名であり、alu_1 が実体 (インスタンス) 名である。同じモジュールから実体を多数作ることも良く行われる。先に述べたように、Verilog では、モジュール内の入出力名を.(ピリオド) で表し、それに接続する外部信号名を括弧の中に書いて対応を付ける。つまり、ALU の a 入力には accum を、b 入力には datain を、com 入力には com を接続し、y に alu_y を接続して出力を取り出す。ここで、com は内部も外部も同じ名前を使っているが、module alu と module datapath の両方で宣言されていれば問題ない。

```
always @(posedge clk)
    accum <= alu_y;
```

最後の部分が、アキュムレータ accum に ALU 出力 alu_y を覚えさせる記述である。この書き方がとっつきにくいのが、Verilog が入門者に嫌われる理由の一つになっているが、決った使われ方しかしないので、「おまじない」と思えば良い。

コンピュータでは、レジスタやメモリにデータを覚えさせるのは、一定の信号の変化に合わせて行なう。この信号をシステムクロックあるいは単にクロックと呼ぶ。3GHz で動作する CPU とか呼ぶが、これはクロックの周波数を示している。これが高い程、CPU は高速で動くので宣伝文句として使われる。[3]

さて、

[3]しかしクロック周波数は性能と直接結びつかないので注意。これは後に取り上げる

```
always @(posedge clk)
```

は、always=いつも、@=at つまり.. の時、(posedge clk)=クロックが L レベルから H レベルに変化する (立ち上がり：positive edge) という意味である。つまり、「clk の立ち上がりの時にはいつも」ということを示す。ちなみに立ち下がりで動作する際は、negedge：negative edge を使う。

```
    accum <= alu_y;
```

<= は、ノンブロッキング代入文と呼び、レジスタにデータを覚えさせる時に使う。先のテストベンチでも使っていた。ここには同じクロックの立ち上がりで動作する複数の文を begin end で使って書くことができる。これは後に紹介する。

2.5.7　データパスのシミュレーション

ここまでのデータパスにはメモリが含まれていない。後に解説するがメモリは普通は論理合成の対象としないため、テストベンチの中に入れて記述しよう。メモリはレジスタ同様 reg 文で記述するが、名前の後に

最初の番地 : 最後の番地

の形で、記憶できる範囲を示す。

```
reg [15:0] dmem [0:15];
```

この場合、dmem は 0 番地から 15 番地まで記憶できるアドレスを 16 持つ。ということはアドレスは 4 ビットで表すことができる。

```
reg [15:0] dmem [0:1023];
```

と書けば、1024=1K の範囲で、アドレスは 10 本になる。原則として範囲は 2 のべき乗で示す。メモリからの読み出しは、4 ビットの daddr を宣言しておき、それを直接括弧の中い入れれば良い。

```
dmem [daddr];
```

書き込みはレジスタ同様 always 文を用いるが、メモリは、データを書き込む場合と、何も書き込まず単に読み出しだけを行う場合があるので、we(write enable) 端子でこれを指示する。we=1 とした時のクロックの立ち上がり時にデータが書き込まれる。そこで、記述は下のようになる。

```
always @(posedge clk)
   if(we) dmem[daddr] <= ddataout;
```

if 文は C 言語など多くのプログラミング言語と同様、括弧内の条件が真になった時に次に書く動作が実行される。最初は奇妙に思われるかもしれないが Verilog は always 文など限られた構造の中でのみ、if 文の利用が許される。

さて、このメモリの記述を含むデータパスのテストベンチは以下のようになる。

```
/* test bench */
`timescale 1ns/1ps
`include "def.h"
module test;
parameter STEP = 10;
   reg clk, rst_n;
   reg [`DATA_W-1:0] datain;
   reg [`SEL_W-1:0] com;
   reg [`ADDR_W-1:0] addr;
   reg we;
   wire [`DATA_W-1:0] accout;
   wire [`DATA_W-1:0] dmem2;
   reg [`DATA_W-1:0] dmem [`DEPTH-1:0];
   always @(posedge clk)
   begin
      if(we) dmem[addr] <= accout;
   end

   always #(STEP/2) begin
            clk <= ~clk;
   end
   datapath datapath_1(.clk(clk), .rst_n(rst_n), .com(com),
               .datain(dmem[addr]), .accout(accout));
   initial begin
      $dumpfile("datapath.vcd");
      $dumpvars(0,test);
      $readmemh("dmem.dat", dmem);
      clk <= `DISABLE;
      rst_n <= `ENABLE_N;
      {we,com,addr} <= {`DISABLE,`ALU_THB,`ADDR_W'h00}; // LD 0
   #(STEP*1/4)
   #STEP
      rst_n <= `DISABLE_N;
      $display("we:%b com:%h addr:%h accout:%h", we, com, addr, accout);
```

```
    $display("dmem[0]:%h dmem[1]:%h dmem[2]:%h", dmem[0], dmem[1], dmem[2]);
  #(STEP*1/2)
     {we,com,addr} <= {‘DISABLE,‘ALU_ADD,‘ADDR_W’h01};  // ADD 1
  #(STEP*1/2)
     $display("we:%b com:%h addr:%h accout:%h", we, com, addr, accout);
    $display("dmem[0]:%h dmem[1]:%h dmem[2]:%h", dmem[0], dmem[1], dmem[2]);
  #(STEP*1/2)
     {we,com,addr} <= {‘DISABLE,‘ALU_ADD,‘ADDR_W’h02};   // ADD 2
  #(STEP*1/2)
     $display("we:%b com:%h addr:%h accout:%h", we, com, addr, accout);
    $display("dmem[0]:%h dmem[1]:%h dmem[2]:%h", dmem[0], dmem[1], dmem[2]);
  #(STEP*1/2)
     {we,com,addr} <= {‘ENABLE,‘ALU_THA,‘ADDR_W’h02};   // ST 2
  #(STEP*1/2)
     $display("we:%b com:%h addr:%h accout:%h", we, com, addr, accout);
    $display("dmem[0]:%h dmem[1]:%h dmem[2]:%h", dmem[0], dmem[1], dmem[2]);
  #(STEP*1/2)
     {we,com,addr} <= {‘DISABLE,‘ALU_THA,‘ADDR_W’h02};  // NOP
  #(STEP*1/2)
     $display("we:%b com:%h addr:%h accout:%h", we, com, addr, accout);
    $display("dmem[0]:%h dmem[1]:%h dmem[2]:%h", dmem[0], dmem[1], dmem[2]);
    $finish;
  end
endmodule
```

まず、クロック信号 clk が一定の間隔で L,H,L,H を繰り返すようにしている。これが下記の文である。

```
  always #(STEP/2) begin
          clk <= ~clk;
  end
```

この場合、STEP の半分で反転するということは、２回反転すればもとのレベルに戻ることから、クロック周期は STEP となることがわかる。この記述でクロックを発振させるには、clk を reg で定義して、初期化する必要がある。

initial 文以降は以前の記述と似ているが、データメモリに初期値を設定するための文が必要である。

```
     $readmemh("dmem.dat", dmem);
```

この場合、データメモリの実体 dmem に対して dmem.dat という名前のファイル中の16 進データ (readmemh だから) が初期設定される。あらかじめ dmem.dat に以下のフォーマットでデータを入れておく。

```
0002
0003
0004
0001
```

ちなみに、2 進数のデータを読み込むためには、readmemb を用いる。dmem.dat はシミュレーションを起動するディレクトリに置いておく必要がある。次に、以下の記述に注目されたい。

```
{we,com,addr} <= {‘DISABLE,‘ALU_THB,‘ADDR_W’h00};
```

Verilog では中括弧は、信号をくっつけて一つにして扱うことを示す。すなわち、we,com.addr はこの順番にくっついて一連の信号線として扱われる。ここでは、we=0, com=001, addr=0000 が設定される。この場合、データパスに対する命令を操作の部分とアドレスの部分に分離している。ここでは、一定の時間間隔で、LD 0, ADD 1, ADD 2, ST 2 の命令が順に与えられる。このため、2+3+4=9 が dmem[2] に入る。

それぞれの display 文で、クロックが立ち上がる度に、アキュムレータの値とメモリの 0-2 番地を表示する。最初に STEP*1/4 としているのは、値を設定する時刻と表示する時刻をクロックの立ち上がりとずらしてやるためである。シミュレーション上完全に同じ時刻に行うという指定をすると、どのような順番でシミュレーションが実行されるかわからなくなるため、困ったことになる。これを防ぐためである。

先ほどと同じように、

```
iverilog test.v datapath.v alu.v
vvp a.out
```

と打ち込みシミュレーションを行い結果が正しいことを確認せよ。

演習 2.8

A を 0 番地、B を 1 番地のデータとして、(SR A) OR (SR B) のデータを 2 番地にしまう命令の実行をテストベンチを改造してシミュレーションせよ。

演習 2.9

A を 0 番地、B を 1 番地、C を 2 番地のデータとして、(A+B) OR (A-B) の結果を 3 番地にしまう命令の実行をテストベンチを改造してシミュレーションせよ。

演習 2.10

このデータパスは、ST 命令の実行時にも accum 中のデータが壊されずに動作する。これはなぜだろう？

第 3

アキュムレータマシン

3.1　アキュムレータマシンの構成

3.1.1　命令メモリと pc

2 章のデータパスでは、計算するデータはデータメモリにあらかじめ入れておけば良いのに、命令は外からクロックに合わせて与えてやらなければならず、これは大変である。そこで、データ同様、命令もあらかじめメモリに入れておき、これを順番に読み出してデータパスに与えるようにすれば便利である。この構成を図 3.1 に示す。

ここでは、8 ビット× 256 エントリの命令メモリ (Instruction Memory) を想定し、この中に命令が並んで格納されていると想定しよう。例えば前回の C=A+B を計算するプログラムが以下のように入っている。

```
0 番地  LD  0     0001 00000000
1 番地  ADD 1     0110 00000001
2 番地  ST 2      1000 00000010
```

ここで、命令を順番に実行するためには、現在実行しているプログラムの番地を保持しているレジスタが必要である。このレジスタをプログラムカウンタ PC(Program Counter) と呼ぶ。今、命令メモリは 256 エントリを持つので、PC は 8 ビットあれば良い。計算機は、以下の動作を繰り返す。

1. PC の指し示す命令を取ってくる（命令フェッチ）同時に PC に 1 を加える
2. 解読し（命令デコード)、
3. 前回定義したデータパス上でこれを実行する（命令実行)。その後 1 に戻る

前回の Verilog の実装では、ST 命令実行時には com=000(THA:スルー A) とすることで、アキュムレータに自分自身の値が書き込まれて、データが変化しなかった。し

かし、この方法では accum に値を書き込まない時は常に com=000 にしなければならず、命令数を増やすことができない。そこで、特定の命令でのみ accum に値を書き込み、それ以外では書き込まないようにしてやる。このために、accum に accset というセット入力を与えてやり、これが 1 の時のみ、結果を書き込むようにする。

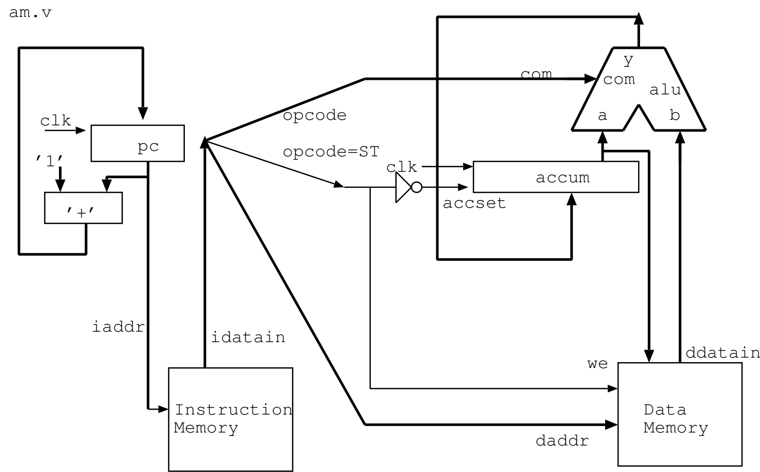


図 3.1: アキュムレータマシンの構成

この構成が、プログラム格納型計算機の第一歩で、アキュムレータマシンと呼ばれる。

3.1.2　アキュムレータマシンの動作

図 3.1 の動作を追ってみよう。まず、命令メモリ (Instruction Memory) のアドレスには pc が繋がっており、この内容の示すアドレスの命令が読み出さる。

読み出された命令が ST 命令以外ならば、Data memory から命令中の下位 8 ビットをアドレスとしてデータが読み出される。つまり LD 1 ならば 1 がアドレスに載る。そして読み出されたデータは ddatain から alu の b 入力に入る。alu の s 入力には、opcode の下位 3 ビットが入っている。LD 命令の場合、opcode は 0001 なので、com は 001 となり、THB が実行されて、読み出されて B 入力に入ったデータが、クロックの立ち上がリで、そのまま accum にセットされる。同時に pc には pc+1 がセットされ、次の番地の命令が指し示される。

読み出された命令が ADD 2 ならば、opcode は 0110 なので、com には 110 が与えられ、alu は a 入力と b 入力の加算を行う。a 入力には accum の値が常に入力され、b 入力は 2 番地から読み出したデータがやってくる。加算結果は EX 状態の終わりに accum に書き込まれる。

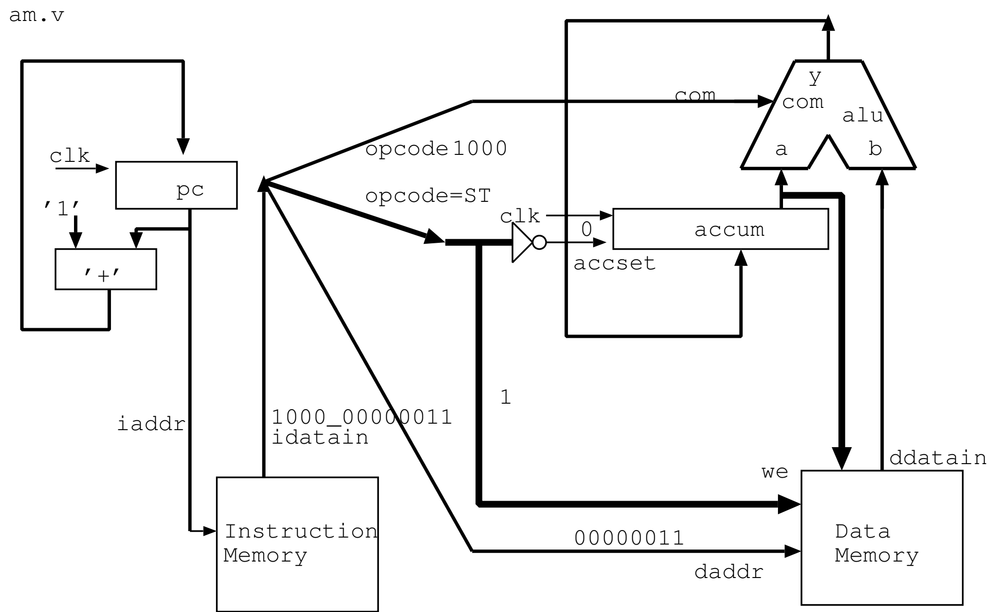


図 3.2: ST 命令の実行例

メモリへの書き込みを行うST命令でのみ、データパスの動作はやや異なっている。ST 3を実行する場合を図3.2に示す。ST命令のアドレスは00000011であるので、最上位が1ならばST命令であるという判断ができる。ここで、Data memoryの入力ポートには常にaccumの出力がdataoutを経由して接続されている。アドレスは他の命令同様、operandの値が与えられるため、この場合は3である。ここでweをHにすることで、data memoryにaccumの値を書き込む。

今の所、以下の命令が利用可能である。Xは番地の2進数を示し、-は使われないので何でも良いこと（ドントケアと呼ぶ）を示す。

```
ニーモニック    機械語         意味
NOP           0000 --------: No Operation: 何もしない
LD X          0001 XXXXXXXX: Load X: acc-m<- X番地中の値
AND X         0010 XXXXXXXX: AND X: acc-m <- acc-m AND X番地中の値
OR X          0011 XXXXXXXX: OR X:  acc-m <- acc-m OR X番地中の値
SL            0100 --------: Shift Left: acc-m <- acc-m<<1
SR            0101 --------: Shift Right: acc-m <- acc-m>>1
ADD X         0110 XXXXXXXX: Add X: acc-m <- acc-m +  X番地中の値
SUB X         0111 XXXXXXXX: S-b X: acc-m <- acc-m -  X番地中の値
ST X          1000 XXXXXXXX: Store X: accum -> X番地中
```

ここで、NOPはアキュムレータの値が自分自身に書き込まれるため、何もしない命令である。何もしない命令なんて意味があるんだろうか？と思われるかもしれないが、実はこの命令は時間稼ぎのために重要で、全てのコンピュータで装備されている。

演習 3.1

0番地の内容から1番地の内容を引き算し、2番地に格納する操作を実行するプログラムを書け。

演習 3.2

0番地の内容から1番地の内容を引き算した結果に、2番地の内容と3番地の内容を加算した結果とORするプログラムを書け。

3.1.3　分岐命令

さて、このアキュムレータマシンは、命令メモリに格納された命令を順番に実行するに過ぎない。前の章の演習プログラムとて、実行が終わったら、次の番地の命令の実行に移ってしまう。完全な猪突猛進型で、演算処理の終了後に、停止することすらできない。

これではあらかじめ必要な処理に対応する命令を全て並べておかなければならず、ちっとも便利ではない。コンピュータが便利なのは、膨大な繰り返し処理をやってくれることと、結果に応じて判断して、実行する処理を変えてくれることである。これにより、プログラムを作って複雑なアルゴリズムを実行させることができる。つまり、「判断」して「処理を繰り返す」という命令が必要だ。

一定の処理を繰り返し実行させるためには、アキュムレータの内容を判断し、PC の内容を変更する命令を設ければよい。これを分岐命令（Branch）と呼ぶ。ここでは以下の命令を想定する。

```
BEZ X　アキュムレータの内容が 0 ならば、PC を XXXXXXXX にする。　　1001 XXXXXXXX
BNZ X　アキュムレータの内容が 0 でなければ、PC を XXXXXXXX にする。1010 XXXXXXXX
```

例えば BNZ 0 を実行すると、アキュムレータの内容が 0 でなければ、PC が 0 になり、次は 0 番地の命令を実行することになる。これをコンピュータ屋は「0 番地に飛ぶ」と言う。この場合アキュムレータの内容が 0 でなければ、分岐命令は何も行わず、次の番地の命令が実行される。

以下の命令が順に実行される場合を想定しよう。

```
0   LD 0     0001 00000000
1   ADD 1    0110 00000001
2   ST 0     1000 00000000
3   LD 2     0001 00000010
4   SUB 3    0111 00000011
5   ST 2     1000 00000010
6   BNZ 0    1010 00000000
7   BEZ 7    1001 00000111
```

ここでデータメモリには、

```
0 0000
1 0002
2 0003
3 0001
```

が格納されているとする。何が起こるか考えて見よう。最初に 3 命令で、0 番地の内容← 0 番地の内容＋ 1 番地の内容が実現できる。0 番地の内容は 0 に初期化しておくことに注意。次の 3 命令で、2 番地の内容← 2 番地の内容－ 3 番地の内容が実行される。ここで 3 番地は 1 に初期化されているため、この 3 命令は 2 番地の内容から 1 を引いて書き戻すことになる。ここで、アキュムレータの内容が 0 ならば 0 番地に戻ることになる。つまり、このプログラムは 2 番地から 1 つづつ引きながら 0 になるま

で1番地の内容を0に足し続ける。2番地が0になると、プログラムは7番地に飛び続けることで停止する(このような止め方をダイナミックストップと呼ぶ)。つまりこれは掛け算のプログラムであることがわかる。ここでは1番地に2が、2番地に3が格納されているので、2 × 3が実行され、0番地が6になる。アセンブラのプログラムは単純で原始的なので、図3.3に示すような流れ図を作って考えるとよい。

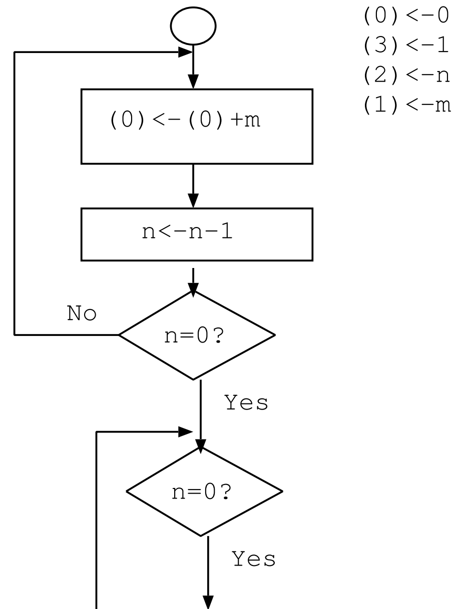


図3.3: 掛け算プログラムの流れ図

分岐命令を実行するためには、(1)命令メモリから読み出した命令が分岐命令であり、(2)条件が成立する場合に限り、PCに命令の下位8ビットの飛び先をセットできるようにすれば良い。

図3.4にこの改造したアキュムレータマシンのハードウェア構成を示す。台形印は、マルチプレクサあるいはデータセレクタというハードウェアモジュールで、この場合、2つの入力の片方を、選択信号pcselに応じて選んで出力する。ここでは、pcsel=1ならば、pcにoperandを送り、S=0ならば、pcには以前通りpc+1を送る。マルチプレクサは、今後データパスを複雑にしていく場合不可欠なものである。また、accumには、accset信号をLにして、分岐命令実行時にはデータをセットしないようにしてやる。この辺はST命令と同様である。太線にBEZ, BNZ実行時のデータの流れを示す。まとめると、命令コードとaccumの値に応じてpcsel, accset, we, comの値を以下のように制御する必要がある。

- ALU演算命令(LD,NOPを含む): accset=1, pcsel=0,we=0, com:opecodeの下位3

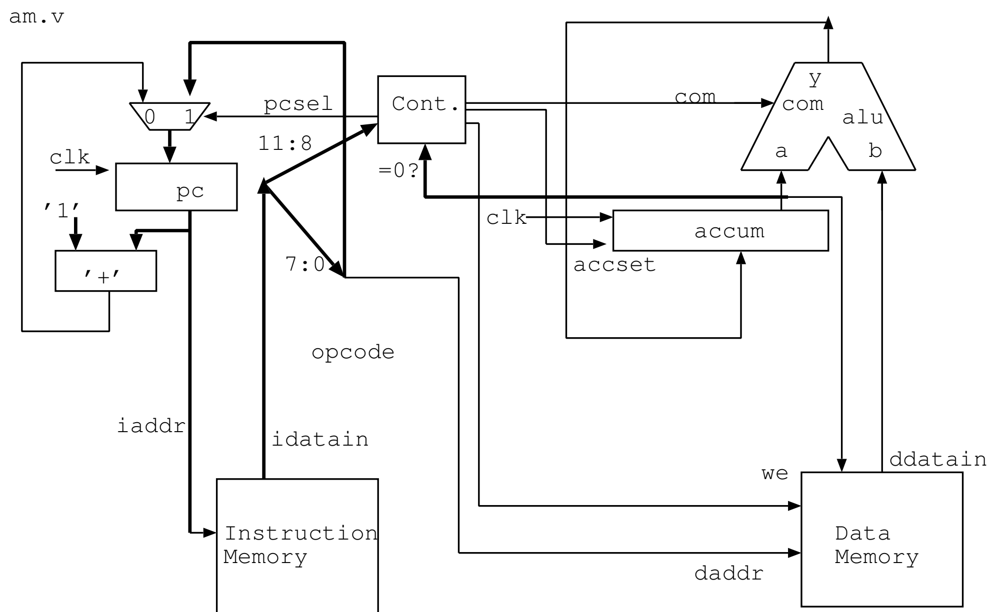


図 3.4: アキュムレータマシンの構成 (分岐命令付き)

ビット

- ST 命令:accset=0, pcsel=0,we=1,com:ドントケア
- 分岐命令：accset=0, pcsel:条件が成立すれば1、そうでなければ0、we=0、com:ドントケア

この信号はコントローラ (Cont) で生成される。コントローラはここでは非常に簡単な組み合わせ回路で済むが、命令が増えるにつれて複雑になっていく。

演習 3.3

今、1番地にAという数が格納されているとする。1＋2＋…Aを計算して0番地（あらかじめ0に初期化されている）に格納するプログラムを書き、機械語に変換せよ。ここでは例題同様、3番地は1という数字で初期化されているとする。

アキュムレータマシンの命令コード

本章のアキュムレータマシンの命令コードをまとめる。

```
NOP         0000 --------: No Operation: 何もしない
LD X        0001 XXXXXXXX: Load X: acc-m<- X番地中の値
AND X       0010 XXXXXXXX: AND X: acc-m <- acc-m AND X番地中の値
OR X        0011 XXXXXXXX: OR X:  acc-m <- acc-m OR X番地中の値
SL          0100 --------: Shift Left: acc-m <- acc-m<<1
SR          0101 --------: Shift Right: acc-m <- acc-m>>1
ADD X       0110 XXXXXXXX: Add X: acc-m <- acc-m +  X番地中の値
SUB X       0111 XXXXXXXX: S-b X: acc-m <- acc-m -  X番地中の値
ST X        1000 XXXXXXXX: Store X: accum -> X番地中
BEZ X  アキュムレータの内容が0ならば、PCをXXXXXXXXにする。    1001 XXXXXXXX
BNZ X  アキュムレータの内容が0でなければ、PCをXXXXXXXXにする。1010 XXXXXXXX
```

3.2　アキュムレータマシンの Verilog 記述

3.2.1　Verilog 記述の解説

図 3.1 の verilog 記述を説明しよう。分岐命令がないのでまだコンピュータとは言えないことからモジュール名は datapath2 となっている。ここでは、図の構造がほとんどそのままの形で記述されている。命令メモリから読み出された来た命令 (idatain) は、opcode と operand に分離される。この opcode が ST 命令 (4'b1000) のときに書き込みが行われるようにしている。逆に accum には、ST 命令以外で alu で演算された

値が格納されるようになっている。pc はリセット時に 0 になり、後は順に 1 ずつ増えて行く。すなわち、0 番地から順に命令が実行される。

```
‘include "def.h"
module datapath2(
input clk, input rst_n,
input [‘OPCODE_W-1:0] opcode,
input [‘DATA_W-1:0] ddatain,
output we,
output [‘ADDR_W-1:0] pcout,
output [‘DATA_W-1:0] accout);

reg [‘DATA_W-1:0] accum;
reg [‘ADDR_W-1:0] pc;
wire [‘DATA_W-1:0] alu_y;

assign we = (opcode == ‘OP_ST);
assign accout = accum;
assign pcout  = pc;
alu alu_1(.a(accum), .b(ddatain), .s(opcode[‘SEL_W-1:0]), .y(alu_y));

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else pc <= pc + 1;
end

always @(posedge clk or negedge rst_n)
begin
    if(!rst_n) accum <= 0;
    else if(opcode !=‘OP_ST) accum <= alu_y;
end

endmodule
```

ここでは、一つだけ新しい構文が出てくる。それは、

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
```

```
   else pc <= pc + 1;
end
```

の部分である。

今回のアキュムレータマシンにはリセット入力が付いている。ご存じと思うが、すべての CPU にはリセットボタンが付いていて、これを押すとシステムは初期化されてしまう。無闇に初期化するとデータを破壊する可能性があるので、このボタンは目立たない所にあって長時間押していないとリセットが掛からないようになっている。我々のアキュムレータマシンにもこのリセット入力が付くわけだが、ここでは、リセット信号線が L の時、リセットが掛かるようになっている。このような信号線をアクティブ-L と呼び、ことでは信号名の後ろに_n を付けて区別する。ちなみに多くのディジタル回路でリセット入力がアクティブ-L[1]なので、このアキュムレータマシンもこの習慣に従っている。

さて、プログラムカウンタ pc は、リセット時に 0 にしておく必要がある。このための記述が always @(posedge clk or negedge rst_n) である。ここでは、clk が 0 から 1 に変化する (立ち上がる:posege) か (or)、rst_n が 1 から 0 に変化する (たちさがる:negedge) 時 (@:at) はいつでも (always)、リセットが掛かる記述になっている。これを非同期リセットと呼ぶ。リセット時の動作は、if 文の中に記述され、今回は単に pc を 0 にしている。ちなみにリセットには同期と非同期があるが、ここでは非同期リセットしか使わない[2]。

もう一つ分かりにくいかと思う記述は、alu モジュールを利用する部分である。

```
alu alu_1(.a(accum), .b(ddatain), .s(opcode[`SEL_W-1:0]), .y(alu_y));
```

今回、ALU の制御入力 s には、opcode の下位 3bit を入れる必要がある。ここでは、SEL_W は def.h の define 文で 3 にしてあり、コロン:で書いた範囲のビット、すなわち 2:0 が切り出されて、s に接続される。

では次に、これをテストするテストベンチ test.v を解説しよう。

```
/* test bench */
`timescale 1ns/1ps
`include "def.h"
module test;
parameter STEP = 10;
   reg clk, rst_n;
   wire [`DATA_W-1:0] ddataout ;
   wire [`ADDR_W-1:0] pcout;
   wire [`OPCODE_W-1:0] opcode;
```

[1]理由については web の解説を参照

[2]理由については web の解説を参照

```
   wire [‘ADDR_W-1:0] operand;
   wire we;

   reg [‘DATA_W-1:0] dmem [‘DEPTH-1:0];
   reg [‘INST_W-1:0] imem [‘DEPTH-1:0];

   assign {opcode, operand} = imem[pcout];

   always #(STEP/2) begin
            clk <= ~clk;
   end
   always @(posedge clk)
        if (we) dmem[operand] = ddataout;

   datapath2 dp2(.clk(clk), .rst_n(rst_n), .pcout(pcout), .accout(ddataout),
        .opcode(opcode), .we(we), .ddatain(dmem[operand]) );

   initial begin
      $dumpfile("datapath2.vcd");
      $dumpvars(0,test);
      $readmemh("dmem.dat", dmem);
      $readmemb("imem.dat", imem);
      clk <= ‘DISABLE;
      rst_n <= ‘ENABLE_N;
   #(STEP*1/4)
   #STEP
      rst_n <= ‘DISABLE_N;
   #(STEP*12)
   $finish;
   end
  always @(negedge clk) begin
      $display("pc:%h inst:%h acc:%h", pcout, {opcode,operand}, dp2.accum);
      $display("dmem:%h %h %h %h", dmem[0], dmem[1], dmem[2], dmem[3]);
   end
endmodule
```

以前の演習での記述と同じく、周期 10nsec(=100MHz) のクロックを発生している。
命令メモリとデータメモリの初期設定を行っている。データメモリと同様に imem を

定義して、その初期値を以下の文で与えている。

```
$readmemb("imem.dat", imem);
```

命令は2進数の方が読みやすいと思い、2進数になっている。imem.datを以下のように初期化してみよう。データファイルの中でも_とコメントは同様に用いることができる[3]。

```
0001_00000000 // LD 0
0110_00000001 // ADD 1
1000_00000010 // ST 2
```

このメモリから読み出した命令はopcodeとoperandに分離され、operandはそのままデータメモリのアドレスに使われる。

```
assign {opcode, operand} = imem[pcout];
```

この書き方は格好いいのだが、サイズをまちがえると酷い目に合うので注意が必要である。

演習3.4:

X,Y,W,Zが0,1,2,3番地に格納されているとする。(X-Z) AND (Y-W)を演算する命令コードを書き、実行して答えを確認せよ。なお、答えはaccum上に出た時点で終わりにして良い。

3.2.2　分岐命令の付加

次に分岐命令を取り付けてみよう。BEZ(1001)命令ではアキュムレータが0の時、BNZ(1010)命令では0でない時に、分岐条件が成立する。この信号をpcselとして図に表しているが、Verilogの記述ではこの信号名を使っていない。これは、always文中のif文中に、この程度の条件ならば簡単に書けるからである。条件が成立すれば、pcにoperandを書き込みそれ以外はpcを1増やす。もちろん、rst_nが0ならばそちらが優先されてpcは0になる。

```
‘include "def.h"
module accum(
input clk, input rst_n,
input [‘OPCODE_W-1:0] opcode,
```

[3]実はちょっと問題があり、webを参照のこと

```
input [`ADDR_W-1:0] operand,
input [`DATA_W-1:0] ddatain,
output we,
output [`ADDR_W-1:0] pcout,
output [`DATA_W-1:0] accout);

reg [`DATA_W-1:0] accum;
reg [`ADDR_W-1:0] pc;
wire [`DATA_W-1:0] alu_y;
wire op_st, op_bez, op_bnz, op_addi ;

assign op_st = opcode == `OP_ST;
assign op_bez = opcode == `OP_BEZ;
assign op_bnz = opcode == `OP_BNZ;

assign we = op_st;
assign accout = accum;
assign pcout = pc;
alu alu_1(.a(accum), .b(ddatain), .s(opcode[`SEL_W-1:0]), .y(alu_y));

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if (op_bez & (accum == 0) | op_bnz & (accum != 0))
        pc <= operand;
   else pc <= pc + 1;
end

always @(posedge clk or negedge rst_n)
begin
    if(!rst_n) accum <= 0;
    else if(!op_st & !op_bez & !op_bnz ) accum <= alu_y;
end

endmodule
```

ここでは、ALU を使った命令以外が 3 つに増えているため、これをそれぞれ識別して信号名を付けている。このように命令によって対応する信号を設けることをデコード (decode:解読) と呼ぶ。

```
assign op_st = opcode == `OP_ST;
assign op_bez = opcode == `OP_BEZ;
assign op_bnz = opcode == `OP_BNZ;
```

命令をデコードすると、対応する信号の値によって所定の制御を行えば良いので記述がすっきりする。

実装した accum.v で、掛け算のプログラムを実行してみよう。imem.dat を下のように設定する。

```
0001_00000000 // LD 0
0110_00000001 // ADD 1
1000_00000000 // ST 0
0001_00000010 // LD 2
0111_00000011 // SUB 3
1000_00000010 // ST 2
1010_00000000 // BNZ 0
1001_00000111 // BEZ 7
```

このプログラムでは 1 番地と 2 番地の掛け算を行って 0 番地に書き込むため、dmem.dat は、例えば以下のようにしておけば、2×3 が計算されるはずだ。

```
0000
0002
0003
0001
```

テストベンチは以前の例題とほとんど同じなので省略する。

演習 3.5

1 番地に X が格納されている。X+(X-1)+...+1 を計算するプログラムをシミュレーションで実行せよ。

第 4

アキュムレータマシンの中身

4.1　抽象化のレベル

今まで、CPU の構成要素はブラックボックス化して扱い、その詳細に触れることはしなかった。ALU は計算をする所、レジスタはデータを格納する所など、その機能を中身を気にしないで理解してもらえば良かった。Verilog の記述もある意味で、抽象化された記述である。ブラックボックス化あるいはモデル化はシステムを理解する上で重要で、中身を完全に知ることはできないし、その必要もない。コンピュータは物理的には大規模な集積回路 (LSI) チップで実現されるが、命令のレベルからチップまでには下に示す階層が存在する。

- 命令セットアーキテクチャ：命令がどのように動くかを決める。次の章でくわしく紹介する。
- マイクロアーキテクチャ : CPU、メモリ、I/O が、どのタイミングでどのように演算やデータのやりとりを行い、制御を行うかを、大雑把にモデル化したもの。
- RTL (Register Transfer Leve) : マイクロアーキテクチャをさらに詳細化し、演算、データのやりとりを詳細にモデル化したもの。
- ゲートレベル : RTL を実現するディジタル回路の論理ゲートの接続を示すレベル。
- 回路レベル：論理ゲートのトランジスタ接続を示すレベル。現在使われるトランジスタとは、CMOS (Complimentary Metal Oxide Semicondoctor) トランジスタであり、NMOS と PMOS という対称的な性質を持つ二種類のトランジスタを使って回路を構成する。
- レイアウトレベル：トランジスタを半導体チップ上に実装する方法。

今までは、マイクロアーキテクチャと RTL を中心に、命令セットアーキテクチャを理解しようと試みてきた。ここでは、きちんと命令セットアーキテクチャを紹介し、RISC の構造を説明する前に、下の階層であるゲートレベルについて触れる。この章は、コンピュータを下のレベルまで理解しなくても良い場合は、スキップしても良い。

4.2　それぞれのモジュール

4.2.1　ALU(Arithmetic Logical Unit)

ALU は、加算、減算、乗算、論理演算、シフト、比較など、データを処理するための様々な演算を行ってくれる装置である。

ALU は図 2.4 に示すように A, B 二つのデータ入力と制御入力 s があって、出力として Y がある。com の内容に応じて、A,B の入力に対して所定の演算を行って Y に出力するのが ALU の役目である。

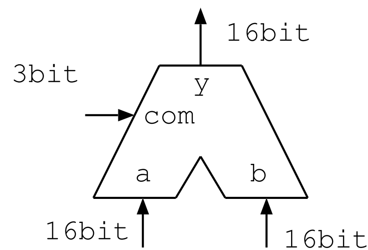


図 4.1: ALU のモデル

ここでは、

s	出力	
000	A	
001	B	
010	A AND B	AND
011	A OR B	OR
100	A を 1 ビット左シフト	SL
101	A を 1 ビット右シフト	SR
110	A+B	ADD
111	A-B	SUB

を想定した。では、この ALU の中身がどうなっているかを知っておこう。

マルチプレクサ

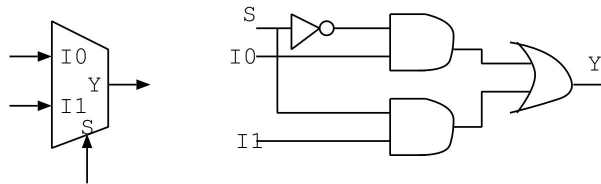


図 4.2: 2 入力マルチプレクサ

ALU は複数の演算を行う演算ユニットがあって、これを com の値によって切り替える機能を持つ。これは実はマルチプレクサ、あるいはデータセレクタと呼ばれるディジタル回路で実現される。2 入力のマルチプレクサを図 4.2 に示す。これは図中に示すゲートの組み合わせで簡単に実現できる。マルチプレクサは、ディジタル信号の切り替えスイッチで、S=0 で入力 I0 が、S=1 で入力 I1 が Y に出力される。ゲートの図は面倒なので書かないが、4 入力のマルチプレクサ、8 入力マルチプレクサも想定できる。図 4.3 には 8 入力のマルチプレクサを用いて ALU を構成する方法を示す。

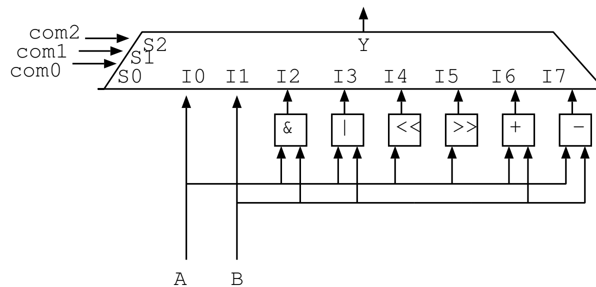


図 4.3: ALU の実現方式

このマルチプレクサという回路は、すごく簡単だが、実は CPU のデータパスを構成する上で鍵となる部品であり、後で繰り返し登場することになる。

4.2.2 加算器

図 2.3 中のそれぞれの演算に相当するモジュールのうち、AND や OR はそれぞれ AND ゲート、OR ゲートで構成できる。シフトは実は 1 ビットシフトならば、線の繋ぎ変えだけで良い。となるとやはりここで中心になるのは加算器ということになる。まずは、もっとも簡単な方法を説明しよう。

二進数のたし算を行うためには、下の桁から順に一桁ずつ足して行き、結果を上に繰り上げる。この繰り上がりのことをキャリ (Carry:桁上げ) と呼ぶ。

```
  11
  0110
+ 0011
------
  1001
```

この例では一番上の段にキャリを示している。加算を実行するためには、それぞれの桁の入力を A, B, 下の桁からの入力を Ci とすると、以下の真理値表を実現すれば良い。

```
A  B  Ci  S  Co
0  0  0   0  0
0  0  1   1  0
0  1  0   1  0
0  1  1   0  1
1  0  0   1  0
1  0  1   0  1
1  1  0   0  1
1  1  1   1  1
```

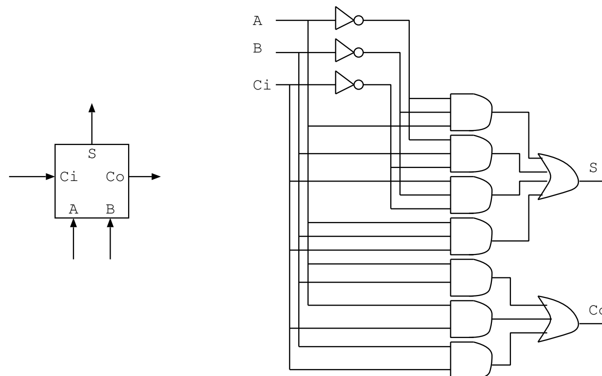


図 4.4: 全加算器

この論理回路も簡単なゲートの組み合わせで図 4.4 に示すように実現できる。これが 1 桁の全加算器 (full adder) である。n 桁の全加算器を作る場合、必要な桁分だけ並

べて、下の桁の Co を自分の Ci に、自分の Co を上の桁の Ci に接続して数珠繋ぎにする。これで、ちょうど手で計算する場合と同様に下のけたから桁上げ信号が伝搬して行って、加算ができる。一番下の桁 Ci 入力は使わないので Low レベルにして、0 を入れる。この様子を図 4.5 に示す。キャリ信号がさざ波 (リプル) のように伝わって行くことからこの加算器をリプルキャリ加算器 (ripple carry adder) と呼ぶ。

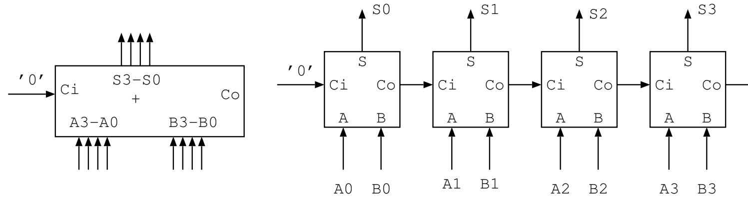


図 4.5: リプルキャリ加算器

加算器は、ALU の中でもっとも複雑な構成要素なので、動作速度の足を引っ張ることが多い。リプルキャリアダーは、単純だが桁数分だけ遅延が積み重なるため、動作が遅く、よほど簡単な場合以外は使われない。キャリが発生するかだけを調べる先見回路を使うキャリ先見加算器 (Carry Look-ahead Adder)、アレイ状に部分加算器を並べるプリフィックス加算器 (Prefix Adder) などが実際の CPU では用いられる。

4.2.3 減算器

第 2 章で触れたが、減算を行うためには 2 の補数を加算すれば良く、2 の補数を作るには以下の手順を踏めばよかった。

- 1 と 0 を反転する。
- +1 する。

1 と 0 の反転は NOT ゲートあるいはインバータと呼ばれる簡単な論理ゲートで実現できる。+1 は加算器で可能だが、この目的だけで加算器を使うのはもったいないので、ちょっとトリックを使って、図 4.6 に示す回路で A-B の減算器が構成可能である。

ここでは、B を反転して A と加算するが、この際、加算器の一番下の桁のキャリ入力を H レベルにして 1 を入れておく。このことで、A+(NOT B)+1 が実現できる。加算器と減算器は多くの部分が共有可能なので、図 2.3 のように別モジュールにするのはバカバカしい。そこで、多くの場合、図 4.7 に示す加減算器を使う。

図の回路では、com が 0 の時は、普通に B が入力されて加算器になる。1 の時は、NOT B が入力されると共に、もっとも下の桁の入力が 1 になることで+1 が行われ、結果として減算器となる。

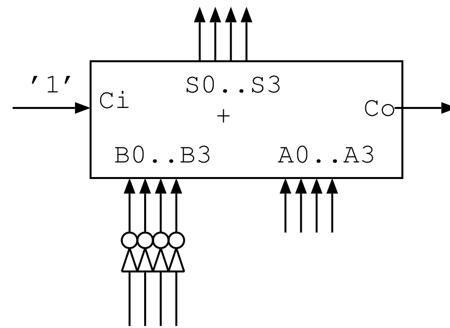


図 4.6: 減算器

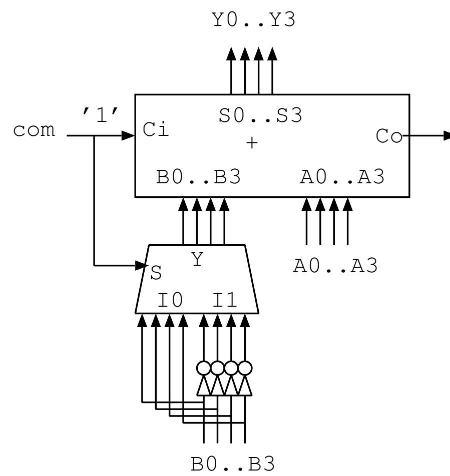


図 4.7: 加減算器

4.2.4 レジスタ

基本的にレジスタは Flip Flop(FF) から作る。FF は 1 ビットの記憶回路で、最も良く使われる D-FF は、図 4.8 に示す記号で表され、D 入力の値をクロックの立ち上がりに同期して記憶して Q に出力する。FF の内部構造はここでは深く触れないが、トランジスタを 12-14 個程度必要とするやや複雑な回路である。

3 章で触れたように CPU を含めて多くのディジタル回路はクロックに同期して全回路が動作する同期式を用いている。この方式では、クロックの立ち上がりで、いつでも FF にデータがセットされることになり、使い難い。そこで、通常用いるのは、D-FF に Enable(イネーブル) 入力を付けた Enable 付き D-FF である。この FF は EN(Enable) 入力が 1 の時にのみ D 入力からデータを取り込み、0 の場合は、現在記憶しているデータをとっておく。この Enable 付き D-FF を必要なビット数分並べてクロックと EN 入力を共通化したものがレジスタである。レジスタは、EN 入力が 1 の時にクロックの立ち上がりに同期してデータを記憶する。

アキュムレータは、この Enable 付き Flip Flop で実現するが、2 章に示したアキュムレータではプログラムカウンタ PC は毎回 1 を足すので、Enable はなくても良い。

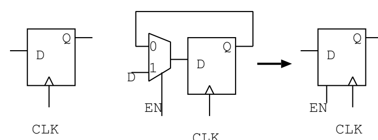


図 4.8: D-FF

レジスタよりも大量にデータを記憶させる素子がメモリであるが、これについては 10 章に紹介する。ここで使うメモリは、同期式の SRAM(Static Random Access Memory) である。半導体チップ内のメモリの多くはこの形を取る。

4.3 論理合成と論理圧縮

4.3.1 Verilog 記述とゲートレベル

Verilog 記述の一部は、対象とするハードウェアモジュールとそのまま対応する。例えば、図 4.2 に示すマルチプレクサは、以下の記述がそのまま対応する。

```
assign Y = S ? I1: I0;
```

AND, OR などの論理演算はそれぞれ対応する論理ゲートに変換される。

```
assign Y = A & B;
assign Y = A | B;
```

図 4.8 に示す D-FF は、Verilog 記述の以下に対応する。

```
always @(posedge CLK)
  if(EN) Q <= D;
```

しかし、Verilog の記述が常に一対一でゲートレベルの回路図に対応するのではない。

```
assign Y = A +B ;
```

と書いた場合、これがどのような加算器に変換されるかは、論理合成用ツールとそれに与えるパラメータに依存する。論理合成用ツールは遅延時間に余裕があれば、コストの小さいリプルキャリ方式の加算器を生成し、高速な演算が要求されれば、プリフィックス方式の加算器を生成する。また、実は先に示したマルチプレクや論理演算の記述も場合によっては他の回路と共用されたりする関係で、違った形で論理合成される可能性がある。次にこの論理合成ツールについて解説する。

4.3.2 合成ツールと使い方

今までは、Verilog はハードウェア構造を記述し、シミュレーションするだけだった。しかし、記述されたハードウェアは何らかのデバイス上で実現しなければ意味がない。このためには Verilog 記述をゲート部ベルの接続図 (ネットリスト) に変換し、様々の最適化を行う作業が必要である。これを論理合成、圧縮と呼ぶ。今日はこの合成ツールの使い方を習得し、これを使って今まで設計してきた POCO を合成、圧縮することが目的である。

論理合成圧縮ツールは、プログラミング言語におけるコンパイラに相当するが、対象がハードウェアであるために、コンパイラよりもはるかに処理が複雑で、時間がかかり、また、ツール自体も高価である。現在、学生レベルで無料あるいは安価に使える論理合成圧縮ツールには以下のものがある。

- FPGA(Field Programmable Gate Array)、すなわち書き換え可能な大規模 LSI のベンダの提供する合成ツール。Xilinx 社の ise、Altera 社の Quartus II は、web パックと呼ばれる簡易版を無料で提供している。web パックは一部の機能が制限されているが、本書で記述するの回路ならば十分合成可能である。

- VDEC(VLSI Design and Education Center: http://www.vdec.u-tokyo.ac.jp/welcome.html) で、研究教育目的で安価で本格的な CAD を利用可能である。

- PARTHENON、このテキストで、以前、用いていた合成ツール。元々NTT が開発したが現在は NPO 法人が管理し、独自の記述言語である SFL のみを受け付け、Verilog は受け付けない。

ここで、論理合成、圧縮用のツールとその利用法については、本書中ではではこれ以上記述せず、web 上で以下の環境を提供することにする。論理合成圧縮用のツールはバージョンアップが激しく、対象となる LSI の進歩も急速である。本書中に詳細を記述してもあっという間に記述が古くなってしまうだろう。web 上に置けばその都度更新することで最新の利用法をサポートすることができる。現在利用可能なのは以下の環境である。

- Xlinix 社の WebPack を利用した環境。Windows/Linux の走る PC があれば無料で個人レベルでインストールして使うことができる。また、同社が販売している Spartan 3E 用のテストボード上で POCO を実装し、もぐらたたきやピンポンゲームなどを動かしてみることができる。
- VDEC により提供される Synposys 社の Design Compiler を用いた論理合成および圧縮。これは、ライセンスの利用を許可された大学や高等専門学校のみで実験ができる。計算センターや、研究室の指導教員に相談されたい。CAD ツールのみの演習は、オクラホマ大の開発したフリーのライブラリを用いて可能である。さらに進んで、富士通 e-shuttle 社のライブラリを用いて Synopsys 社の IC Compiler を利用して実際のチップを設計実装することができる。これも web 上で掲示しているので、利用のためには研究室の指導教員に相談されたい。

第 5

汎用レジスタマシンへの拡張

5.1 汎用レジスタマシン

今までアキュムレータマシンの命令について検討してきた。3 章で分岐命令を導入したことで、簡単なプログラムが書けるようになった。プログラムを書くことのできる命令の一揃いを命令セットと呼び、命令セットの作り方、あるいはその作り方に基づいて出来上がった命令セットのことを命令セットアーキテクチャと呼ぶ。命令セットアーキテクチャとは 4 章で紹介した抽象化の上のレベルに当たり、ハードウェアとソフトウェアのインタフェースである。ハードウェア設計者は、この命令セットアーキテクチャを実現するために様々な要求に応じて様々な構成を作ることができる。お金は掛かっても高速な CPU が欲しい場合と、遅くても良いので安くて電力消費が少ないものを作りたい場合では、構成は全く違ったものになる。しかし、同じ命令セットアーキテクチャを使っていれば、同じプログラムが動く。一方、プログラマは、どのようなハードウェアになっているのかを気にしないで、命令セットだけを知っていれば、プログラムを書くことができる。

この章では、より高度な命令セットアーキテクチャをもっとも簡単な実現モデル (マイクロアーキテクチャ) を示しながら紹介する。

さて、アキュムレータマシンは、ループを含むアルゴリズムの実行が可能であったが、以下の問題があった。

- アキュムレータだけしかレジスタがないので、演算の度に結果をメモリに格納する必要がある。
- アキュムレータだけしかレジスタがないため、メモリのアドレスを動的に指し示す機能がない。つまりポインタの実現ができない。

さらに、これはアキュムレータマシンの一般的な問題ではないのだが、アドレスの指定が長くなりすぎると面倒なので、アドレス空間を 4 ビット、すなわち全体で 16 と

した点にも問題がある。コンピュータの歴史を振り返ると、開発するプログラムサイズは年々増大し、必要とされるアドレス空間は直線的に増加している。現在の高性能のマイクロプロセッサが64ビットアーキテクチャを採用している主な原因は32ビットアーキテクチャではアドレス空間が不足するためである。このことを考えると、アドレス空間4ビットはいくら演習用でもちょっとひどすぎる。

そこで、以下の改造を行おう。

- レジスタ数を8個に増強する
- アドレスを16bitに拡張して、アドレス空間を64Kとする

まず、8個のレジスタは、基本的に全てが同じ機能を持たせるようにする。初期のマシン（現在でも8ビットマシンなどは）ではレジスタの機能が専門化され、レジスタによって可能な演算や操作（メモリのアドレスのみ格納する等）が決っている場合が多かった。これを専用レジスタマシンと呼ぶ。しかし、専用レジスタマシンは、レジスタの機能を考えてプログラムをするのが大変で、コンパイラも作りにくい。このため、半導体の面積の余裕ができるにつれ、一部の例外を除いてすべてのレジスタが同様にすべての機能を実現できるマシン、すなわち汎用レジスタマシンに移行した。我々も汎用レジスタマシンを考えよう。

汎用レジスタマシンでは、レジスタが複数あるので、代表的な演算命令のオペランドが2または3となる。すなわち、2オペランド命令あるいは3オペランド命令となる。

```
ADD r0, r1          r0<-r0+r1
ADD r0, r1, r2      r0<-r1+r2
```

ここで、r0は結果が格納されるレジスタであり、ディスティネーションレジスタ（オペランド）と呼ぶ。一方、r1(r2)は、元となるデータが格納され、演算によって変化しないレジスタであり、ソースレジスタ（オペランド）と呼ぶ。マシンによっては、オペランドに、アキュムレータマシン同様、メモリのアドレスを書くことができるものもある。

```
ADD r0, 1000             r0<-r0+(1000)
ADD 1000, 2000, 2002     (1000)<-(2000)+(2002)
```

最初の命令はr0と1000番地の中身を足して、答えをr0に格納し、次の命令は2000番地と2002番地の中身を足して答えを1000番地に格納する。

ここでは、ディスティネーションオペランドを一番左に書く表記法を採用している。この表記法はIBMやIntelのマシンで使われている方法で、現在一般的である（ディスティネーションオペランドを一番右に書くやり方もあり、DECやモトローラのマシン、日立のSHなどに使われている）。

5.2　汎用レジスタマシンの分類

汎用レジスタマシンの命令セットは、演算命令のオペランドにメモリを指定できるかどうかによって以下のように分類される。

- まったく指定できない: register-register アーキテクチャ、load-store アーキテクチャまたは RISC(Reduced Instruction Set Computer) と呼ばれる。オペランドにメモリを指定できないため、演算を行う場合、必ずレジスタに値を Load してからレジスタ間で演算し、答えをメモリに Store する。一定の処理を行うための命令数は多くなるが、個々の命令は単純化され固定長で実装可能である。SPARC,MIPS,ARM,SH-4 などのマイクロプロセッサはこの命令形式を用いている。
- どこでも指定できる: memory-memory アーキテクチャ、オペランドのどの部分でもメモリが指定できる。もちろんレジスタ間でも演算が可能だが、メモリ中のデータ同士を直接計算する場合が多くなる。一定の処理を行うための命令数は少ないが、個々の命令は可変長にする必要があり、複雑になるため CISC(Comptex Instruction Set Computer) と呼ばれる。DEC の VAX-11 で用いられ、80 年代はメジャーであったが、性能向上が難しいことから現在はほとんど使われていない。
- 1 つだけ指定できる: register-memory アーキテクチャ、オペランドに一ヶ所だけメモリを指定できる。多くの場合 2 オペランド命令でソースオペランドに限られる。register-register 型と memory-memory 型の中間的な性質を持つ。一定の処理を行うための命令数はさほど多くならず、命令は可変長にする必要があるが、memory-memory 型ほど悲惨なことにはならない。現在、デスクトップやラップトップで最も良く用いられている Intel/AMD の IA32,IA64 はこの型に属する。

例えば 0x1000 番地のデータと 0x1010 番地の内容を加算して 0x2000 に格納する場合、register-register 型では以下のようになる。

`0x1000` の中身をレジスタに持ってくる (LD)
`0x1010` の中身を別のレジスタに持ってくる (LD)
加算
`0x2000` に答えを格納 (ST)

このように register-register 型は、命令数が多いことがわかる。しかし、メモリのアドレスは LD,ST のみで済み、命令長を固定にすることが可能である。
一方、memory-memory 型では一命令ですむ。

`0x1000` の中身と `0x1010` の中身を足して `0x2000` に格納

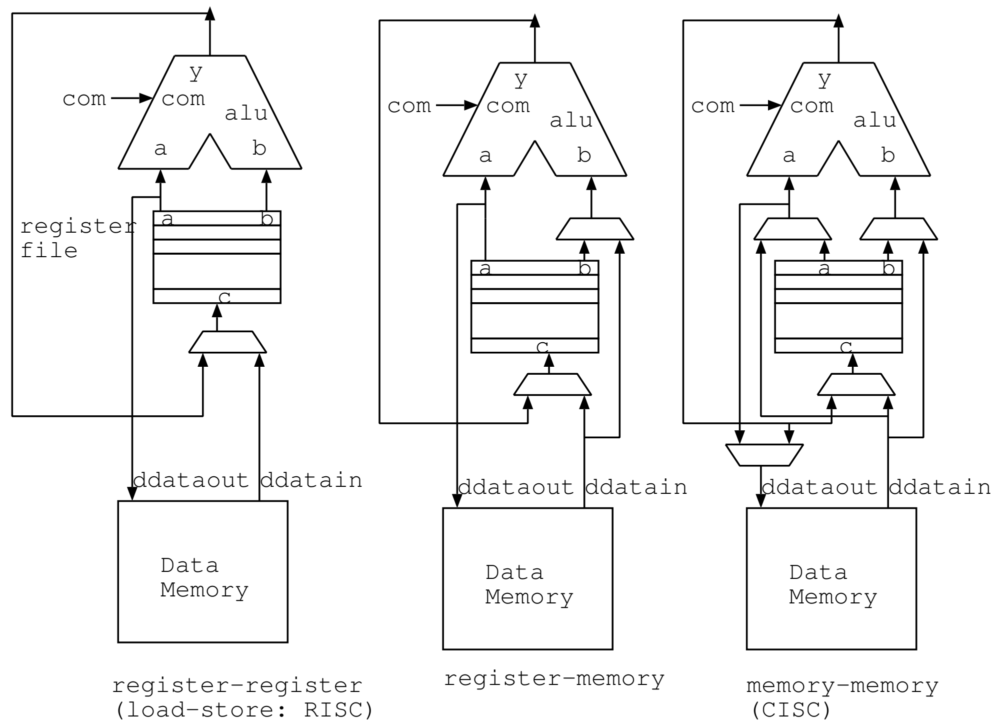


図 5.1: 汎用レジスタマシンのデータパス

しかし、オペランドにはレジスタも指定できるので、命令長は長くなったり短くしたりしなければならないことがわかる。

register-memory 型は、ディスティネーションは、レジスタのみ許される場合が多いので以下のようになる。

0x1000 の中身をレジスタに持ってくる (LD)
0x1010 の中身を加算
0x2000 に格納 (ST)

これも、ADD 命令にメモリアドレスを書く場合と書かない場合で、命令長を変える必要があるが、その可変の程度は memory-memory 型よりもひどくはない。一方、一命令では済まず、3 命令必要になっている。

3 つの方式のハードウェア構成を考えてみよう。複数の汎用レジスタは、レジスタファイルというハードウェアモジュールで実装される。レジスタファイルは小規模なマルチポートメモリであり、ここでは図 5.1 に示すように、読み出しに 2 ポート (a,b)、書き込みに 1 ポート (c) を持っている。今、レジスタファイルの中に 8 個のレジスタが入っているとすると、好きな番号のレジスタを a ポート、b ポートから独立に読み出すことができる。同時に、c ポートからデータを書き込むことができる。

これを単純にアキュムレータの代わりに入れ替えた構成が、図の真ん中に示す register-memory 型である。この点で、register-memory 型はアキュムレータマシンのもっとも自然な発展であると言える。次に左に示す register-register 型は、メモリから ALU への直接入力が存在せず、全てのデータが一度レジスタを介して演算される。一方で、memory-memory 型はレジスタとメモリが同様に ALU に接続されている構成である。

さて、register-memory 型は、アキュムレータマシンの自然の発展であり、また、最も良く用いられている Intel/AMD の CPU の命令セットは、この形である。しかし、この型が最もうまく行っていると勘違いしてはならない。実は Intel/AMD のマイクロプロセッサは、Pentium II 以降は、register-memory 型の命令を内部的に register-register 型に変換して実行している。register-memory 型の見掛けを維持しているのは、アキュムレータマシンの自然な発展として register-memory 型を採用してしまったが故に蓄積したプログラムとの互換性 (Compativility) を保持するために過ぎない。高速に実行するという点でいうと、register-register 型つまり RISC はとことん優れており、わざわざ実行時に変換して実行せざるを得ないのだ。つまり、これらのプロセッサは外面が register-memory 型でも、一皮剥けば RISC であり、組み込み用ではほとんどの CPU が RISC であることを考えると、プロセッサの命令セットという点では RISC の圧勝である。

そこで、ここでも、今までのアキュムレータマシンを register-register 型、すなわち RISC に拡張していくことにする。ここで、この RISC に名前を付け、POCO と呼ぶことにする。

5.3　16bit RISC: POCO

POCO の基本構成を以下のように決める。

- 命令メモリ:16bit ×アドレス 16bit=アドレス空間 64K
- データメモリ:16bit ×アドレス 16bit=アドレス空間 64K
- レジスタ:16 本 (全部 16bit):r0-r7
- 機械語命令：16bit 固定長

これは、小規模な組み込み CPU としてはそこそこリアルな構成である。

5.3.1　メモリの読み書きをどうするか？

まずもっとも問題となるのはメモリの読み書きである。メモリのアドレスを 16bit に拡張したため、LD 命令と ST 命令で直接アドレス指定すると、命令の全体長を越えてしまう。また、ポインタや配列の実現ができないなどのアキュムレータマシンの問題点を解決できない。

そこで、POCO では他の RISC と同じくレジスタの中身で、メモリのアドレスを表す方法を採用する。これをレジスタ間接メモリアクセスと呼ぶ。

```
LD r0, (r1)      r0 <- (r1)
```

上記の命令を実行すると、r1 に中身がアドレスとなり、読み出されたデータが r0 に格納される。r1 が 0 ならば 0 番地、100 ならば 100 番地が読み出される。r0-r7 には 16 ビットの数が格納されるので、全アドレス空間をアクセスすることができる。この場合、r1 はポインタそのものであり、この値に演算を行うことで配列、スタックなどのデータ構造を実現することができる。

ST も同様に表される。

```
ST r0, (r1)     r0 -> (r1)
```

この場合、r0 の中身が、r1 で指定するアドレスに書き込まれる。すなわち、r1 が 0 ならば 0 番地、100 ならば 100 番地に r0 の中身が書き込まれることになる。ここで、データの移動の方向が ST 命令のみは左（レジスタ）から右 (メモリ）になる点に注意されたい。

5.3.2　基本演算命令

次に、基本的な演算命令はレジスタ同士の演算である。

```
ADD r0, r1    r0 <- r0+r1
SUB r1, r2    r1 <- r1-r2
```

今回、ALU についてはビット幅が 16 ビットになっただけで、アキュムレータマシンと同様の機能を想定する。この場合以下の命令が定義できる。

```
AND r0, r1    r0 <- r0 and r1
OR r1, r2    r1 <- r1 or r2
SL r0         r0 <- r0 << 1
SR r2         r2 <- r2 >> 1
```

さて、POCO は RISC すなわち register-register 型なので、演算はレジスタ同士でしか許していない。ところが、ここで問題があることに気づく。メモリからのデータがレジスタ間接指定で、演算がレジスタ同士でしか許されないのならば、レジスタにアクセスする番地を入れることができないではないか？最初にレジスタに値を設定する手段を設けておかないと、メモリからデータを取ってくることができなくなってしまう。これでは困る。そこで、命令中に直接数字を指定する命令を用意する。これがイミーディエイト命令である。

```
LDI r1,#3   r1 <- 3
```

これは、r1 に 3 という数値を直接入れてしまう命令である。メモリの番地を指定しているのではなく、あくまで 3 という数自体を入れていることに注意されたい。この指定方式を直値（即値）指定あるいはイミーディエイト (Immediate) 指定と呼び、この指定を使った命令を直値命令あるいはイミーディエイト命令と呼ぶ。同様に、演算についてもこの考え方を用いて、

```
ADDI r2,#5   r2 <- r2 + 5
```

とできると便利である。命令のオペランドが直接レジスタに入ったり、演算に使われたりするのは、やや特殊なケースと言って良い。このため、アセンブラ表記ではこのことを強調するために、イミーディエイトデータの前に # を付けて表すのが普通である。

ここでさらに一つ問題がある。このイミーディエイトデータを指定する領域を命令内に確保しなければならないが、命令は全体で 16 ビットに納めなければならない。この中にはオプコードが最低 5 ビット（32 命令分）は必要であるし、レジスタを指定する 3 ビット (r0-r7 で 8 つ分) も必要である。したがって、イミーディエイトデータ

に使える分は 8 ビットしかない。そこで、8 ビットのデータを 16 ビットに幅を広げる手段が必要である。ここで 2 章で紹介した符号拡張が登場する。POCO 内で使う符号付き数は、通常のコンピュータと同じく 2 の補数表現であらわす。そこで、符号を考えて拡張してやるのが原則である。しかし一方、符号を考えない数も扱う必要があるため、イミーディエイト命令は符号付きと符号無しの両方を用意する必要がある。

POCO では以下の 4 つを用意する。

```
LDI rd,#X          Load Immediate          rd <- X (符号拡張)
LDIU rd,#X         Load Immediate Unsigned rd <- X (符号拡張なし)
ADDI rd,#X         ADD Immediate           rd <- rd + X (符号拡張)
ADDIU rd,#X        ADD Immediate Unsigned  rd <- rd + X (符号拡張なし)
```

SUBI は用意する必要はない。これは、

```
ADDI r1, #-1
```

が可能であるからだ。他の演算も重要度はさほど高くないので、イミーディエイト命令としてはこれだけで良いだろう。ちなみに、イミーディエイトフィールドを 8bit にする、という選択は、「命令の長さが足りないから」という理由だけでなく、「16bit 全部持たせると無駄だから」という点からも合理的である。プログラムを書いてみると多くの場合、イミーディエイト命令では、1(あるいは 2 とか 3 とか小さな数) を足すとか引くとかが多く、意味のある桁数はさほど長くない。このような場合、全部持たせるというのは命令セットの設計上無駄が大きくなってしまう。

以上で基本的な演算が可能になる。例えば 0 番地の内容と 1 番地の内容を加算して 2 番地に格納するプログラムは以下のようになる。

```
LDIU r0,#0
LD r1,(r0)
ADDIU r0,#1
LD r2,(r0)
ADD r1,r2
ADDIU r0,#1
ST r1,(r0)
```

この場合アドレスが小さいので、LDIU, ADDIU は LDI, ADDI と同じである。ここでは r0 をポインタとして 1 ずつ進める方式を取っているが、その都度値をじかに LDIU 命令で設定しても良い。実はここまでのイミーディエイト指定では、下位 8bit しか指定されず、上位 8bit は符号ビットか 0 になってしまう。このため、16bit レジスタの上位 8bit に値を設定することができず不便だが、これに対処する命令 LHI(Load High Immediate) は、演習で実装しよう。

5.3.3 命令フィールドの設計

他の RISC 同様、POCO では 16bit 固定長に全命令を実現する必要がある。まず、オプコードを 5bit とする。これだと 32 命令実現可能である。32 はいくら教育用といえ、少なすぎるような気がするが、後で RISC 共通の方法を使って拡張する。

イミーディエイト命令は、レジスタを 1 つ指定する必要があるので、オプコードの後は、そのレジスタの番号を入れて残りをイミーディエイトデータに割り当てて、以下のようにする。

```
LDI rd,#X          01000 ddd XXXXXXXX    rd <- X (符号拡張)
LDIU rd,#X         01001 ddd XXXXXXXX    rd <- X (符号拡張なし)
ADDI rd,#X         01100 ddd XXXXXXXX    rd <- rd + X (符号拡張)
ADDIU rd,#X        01101 ddd XXXXXXXX    rd <- rd + X (符号拡張なし)
```

例えば

```
ADDI r5,#100      01100 101 01100100
```

となる。

このようにオプコード (opcode: 01100), レジスタ (ddd)、イミーディエイト (XXXXXXXX) など、命令中のそれぞれ意味を持った部分のことを命令フィールドと呼ぶ。

次に ADD などレジスタを 2 つ指定するタイプの命令について考えよう。同様に opcode は 5 ビットとし、レジスタ二つを指定するフィールドを設けると、残りは、16-5-3*2=5bit 余る。ここで、この余りのフィールドを用いて命令数を増やすことを考える。すまわち、全ての ALU を用いた命令は、opcode を 00000 とし、この余りの 5 ビットのうち下位 3 ビットを ALU のコマンドに入れて演算を指定する。この下位 5 ビットを function(funct) フィールドと呼ぶ。3bit 目、4bit 目の 2 ビットが完全に余るが、これは将来のために取っておくことにし、ALU 命令ではここを 00 とする。すなわち、opcode が 00000 で、function が 00XXX であれば、ALU 命令であり、XXX が ALU のコマンドとなる。

今までと同じ ALU を利用すれば以下の命令が定義される。

```
NOP                                00000 --- --- 00000
MV  rd,rs    rd <- rs              00000 ddd sss 00001
AND rd,rs    rd <- rd AND rs       00000 ddd sss 00010
OR  rd,rs    rd <- rd OR rs        00000 ddd sss 00011
SL  rd       rd <- rd<<1           00000 ddd --- 00100
SR  rd       rd <- rd>>1           00000 ddd --- 00101
ADD rd,rs    rd <- rd + rs         00000 ddd sss 00110
SUB rd,rs    rd <- rd - rs         00000 ddd sss 00111
```

ddd 同様、sss にはソースレジスタ rs を示す番号 (r0-r7:000-111) が入る。RISC では先の図 5.1 に示すように、ALU の B 入力につながっているのはレジスタファイルの B ポートなので、ALU のコマンド 001: THB は、LD ではなくレジスタ間の転送命令である MV(move) となる点に注意されたい。THA は、今まで同様に NOP(No-operation) 命令となり何も行われない。この方法はなんだかセコいようだが、命令数を増やし、ハードウェアを簡単にするのに有効なので、ほとんどの RISC で同様の方法を用いている。

それでは LD, ST 命令を ALU 命令と同じ形で表現することにしよう。ここでは function の 5,4 ビット目が 01 で LD, ST を表すこととした。ST を先に持ってきたのはアキュムレータマシンと命令コードを同じにした方が慣れていていいかと思ったためで、深い意味はない。aaa はメモリアドレスを示すレジスタという意味で、やはり r0-r7 までのレジスタ番号が入り 000-111 で表す。

```
ST rs, (ra)     rs -> (ra)     00000 sss aaa 01000
LD rd, (ra)     rd <- (ra)     00000 ddd aaa 01001
```

演習 5.1

0 番地,1 番地にそれぞれ A,B が格納されている。(A-B) OR B の演算を行い、2 番地に答えを格納するプログラムをアセンブラ表記、機械語の両方で示せ。

5.4　POCO の内部構造

アキュムレータマシンと同じく、POCO のデータパスは、大きく分けて、命令を命令メモリから取ってくる部分（フロントエンドと呼ぶ）取ってきた命令に基づいて演算やメモリアクセスを行う（バックエンドと呼ぶ）に分けることができる。図 5.2 に概略図を示す。

フロントエンドでは PC(Program Counter) の指し示すアドレスに格納されている命令を読み出す。読み出された命令は idatain からバックエンドに送られる。PC は次の命令を読み出すためにカウントアップされる。

5.4.1　レジスタ間演算命令の実行

次にバックエンドに目を向けよう。バックエンドの主な構成要素は、r0-r7 までのレジスタがまとめて格納されているレジスタファイルと演算を行う ALU、データメモリであり、データの流れを切り替えるためにマルチプレクサが配置されている。レジスタファイルは図 5.3 に示す 3 ポート構造を想定する（現時点では、実は読み出しの a ポートアドレスと書き込みの c ポートアドレスは共有可能である）。レジスタファイ

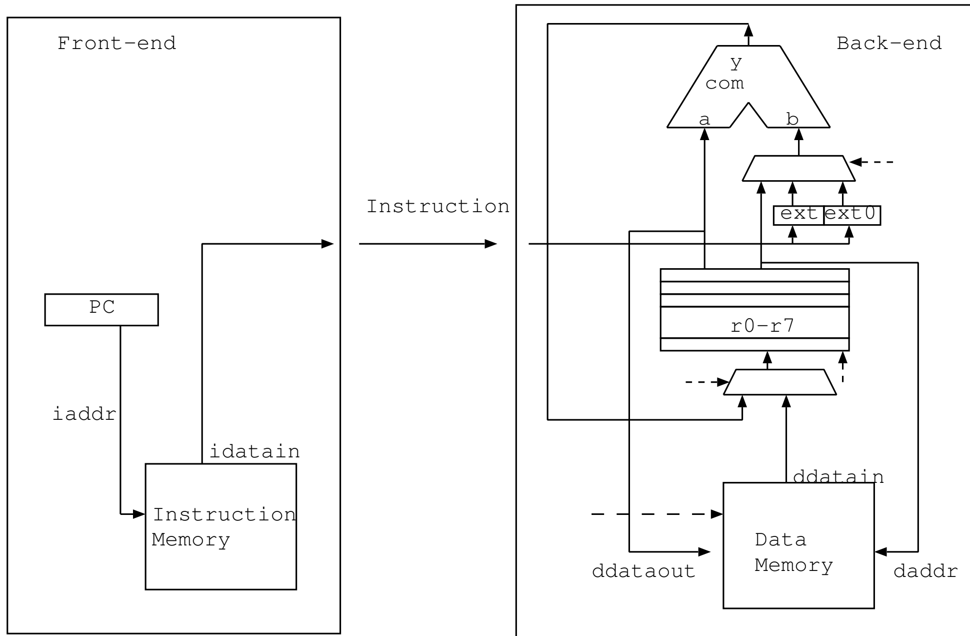


図 5.2: データパスの概略図

ルは一種のメモリであり、aadr から与えられた番号（3 ビット）のレジスタが内容が rf_a に読み出され、badr に与えられた番号（3 ビット）のレジスタの内容が rf_b に読み出される。一方、書き込みについては、cadr に与えた番号のレジスタに対して、rwe が 1 の時のクロックの立ち上がりで、rf_c からのデータが書き込まれる。aadr, badr, cadr は後で解説するように命令中のレジスタを表すフィールドにより指定する。

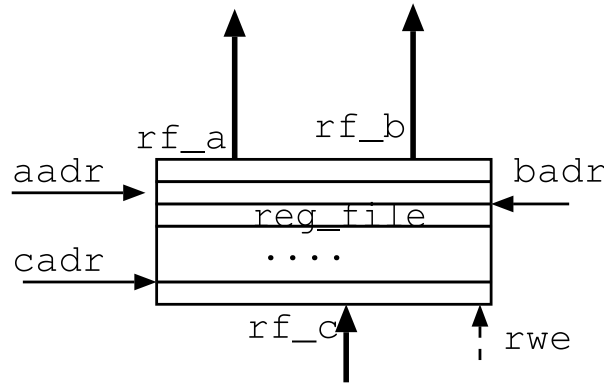


図 5.3: レジスタファイル

POCO は、register-register マシンなので、メモリから ALU への直接パスは存在しない。メモリから読み出したデータは、マルチプレクサを介してレジスタファイルの rf_c から書き込まれる。

まずは、レジスタ同士の演算命令を実行する場合を考える。

レジスタ同士の演算の機械語フォーマットは以下の通りである。まず opcode は 00000 で、これに続く 3 ビット (10 から 8 ビット) でディスティネーションレジスタを、次の 3 ビット (7 から 5 ビット) でソースレジスタを、最後の function5 ビット (4 から 0 ビット) で命令の種類を表す。

例えば、ADD r0,r1 を例に取ると、この命令の機械語は、以下のフォーマットになっている。

```
ADD r0,r1  00000 000 001 00110
```

10 から 8 ビット目のディスティネーションレジスタ番号をレジスタファイルの A ポートアドレス (addra) に繋いで、ディスティネーションレジスタを A ポートから読み出す。7 から 5 ビット目のソースレジスタ番号の方は B ポートアドレス (addrb) に繋いで、ソースレジスタを B ポートから読み出す。ALU の入力用マルチプレクサの制御入力 alu_bsel は 00 として、レジスタファイルからの値を入力してやる。

ALU で行う演算の種類は com で指定する。POCO の命令コードは、最後の funct コードの下 3 ビットが、com と一致するようになっている。つまり、110 ならば加算

が行われる。先に述べたように、この方法はセコいようだが、単に繋げば良いため楽なことがわかる。ディスティネーションレジスタとソースレジスタ間の演算結果は、レジスタファイルに書き込まれる。このため、C ポートのアドレスには、ディスティネーションレジスタ番号を入れ、レジスタファイル書き込み用マルチプレクサの制御線 rf_csel を 0 にする。書き込み制御信号 rwe=1 とすればレジスタファイルへの書き込みが行われる。上記のデータの流れについて図 5.4 に示す。

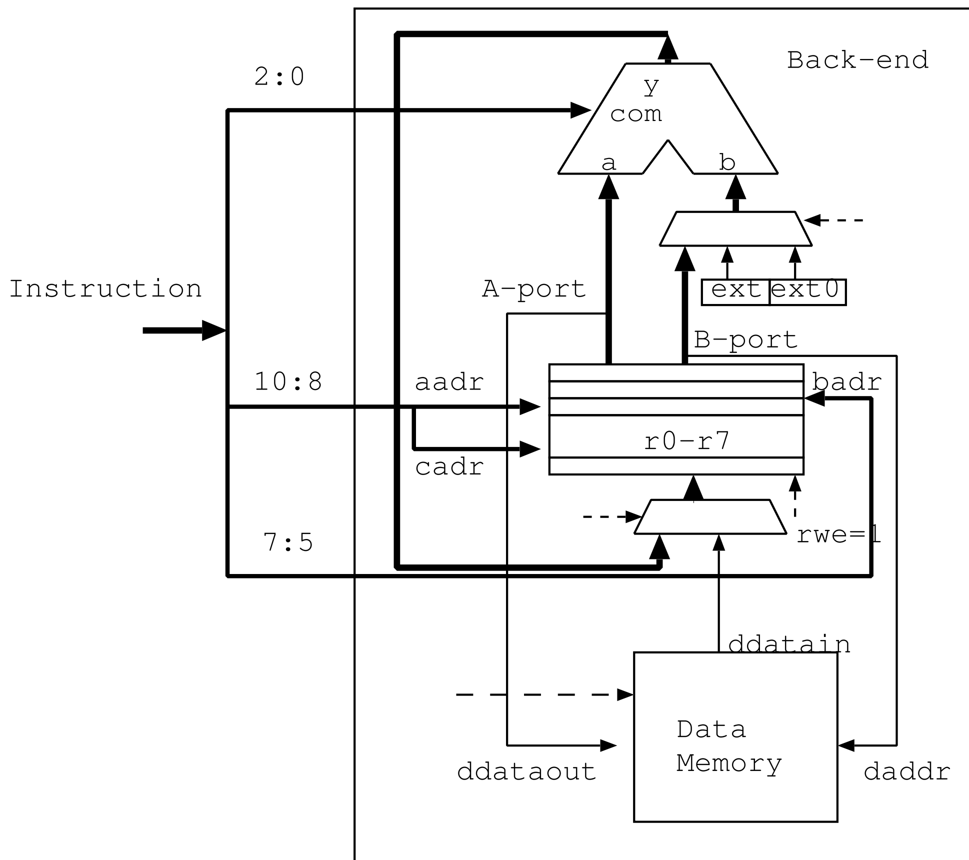


図 5.4: レジスタ間演算命令の動き

5.4.2　LD/ST 命令の実行

LD 命令、ST 命令の場合、メモリのアドレスを格納しているレジスタは、ソースレジスタとして 8-6 ビット目に指定されている。A ポートの出力は、直接データメモリのアドレスに接続しているので、レジスタから値を読み出せば自動的にアドレスが与えられる。LD 命令の場合、読み出されたデータは、ALU からの演算結果に代わって C ポートで与えられたディスティネーションレジスタに格納される。このため、レジスタファイルの書き込み用マルチプレクサの制御線 rf_csel を 1 とする。一方、B ポートの出力は直接データメモリの入力データに接続されているので、ST 命令が実行される場合、ソースレジスタの中身が書き込まれる。この場合はデータメモリの書き込み制御線 (we) を 1 にし、レジスタファイルの書き込み制御線 (rwe) を 0 にする。この様子を図 5.5 に示す。

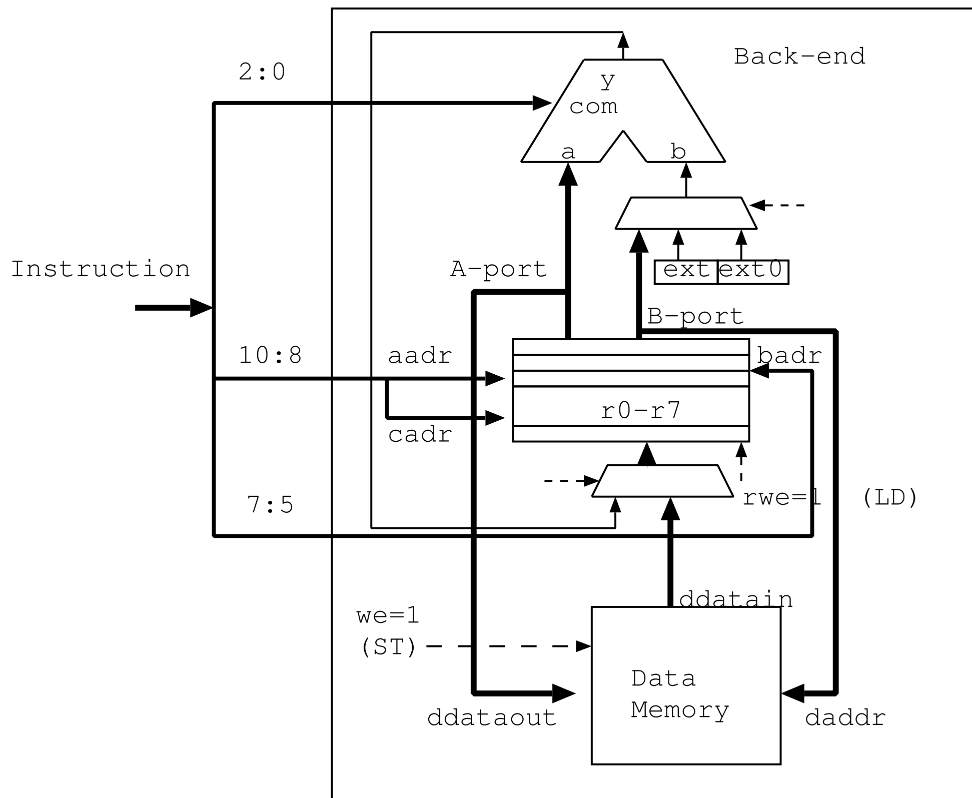


図 5.5: LD/ST 命令の動き

5.4.3 イミーディエイト命令の実行

イミーディエイト命令は、ソースレジスタの指定がなく、下位 8 ビットがコード中に指定されている。例を以下に示す。命令は opcode により判定する。

```
LDI   r0,#-32  01000 000 11100000
LDIU  r0,#224  01001 000 11100000
ADDI  r0,#-32  01100 000 11100000
ADDIU r0,#224  01101 000 11100000
```

ここで、機械語中の 8 ビットを 16 ビットのレジスタ中のデータと演算する際にビットの長さを揃えて 16 ビットにする必要がある。この際に、符号を考えて 16 ビット化する、すなわち符号拡張する方式と 0 を埋めて 16 ビット化する 0 拡張する方式がある。例えば ADDI は前者なので、0xffe0 すなわち-32 が r0 に加算され、ADDIU は後者なので 0x00e0 すなわち 224 が加算される。

この命令を実行するには、まず機械語の下 8 ビットを ALU の B 入力まで引っ張って来る必要がある。引っ張ってきた結果は、符号拡張 (ext) あるいはゼロ拡張 (ext0) されて入力される。ちなみに ext, ext0 は、図中では箱で表しているが、実際は最上位ビットや 0 を並べて出力するだけで、非常に簡単なハードウェアである。ALU の B 入力選択用のマルチプレクサの制御入力は符号拡張の場合は 01、ゼロ拡張の場合は 10 にしてやる。

さて、イミーディエイト命令は直接 B 入力のデータを書き込むこと (LDI, LDIU) と加算する (ADDI, ADDIU) だけなので、これを直接 ALU の com に入れてやる。ALU コマンド選択用のマルチプレクサ制御入力 comsel を命令に応じて切り替える。レジスタファイルへの書き込みは、レジスタ同士の演算命令と同じである。この様子を図 5.6 に示す。

5.4.4 制御回路

今までの解説で、各部のデータの流れはご理解いただけたと思う。この流れを実現するように、命令の種類に応じて制御信号を設定してやればそれぞれの命令を動作させることができる。制御する対象は ALU のコマンド入力を選択する comsel、ALU の B 入力の選択用 alu_bsel、レジスタファイルの書き込みを選択する rf_csel で、これらはマルチプレクサの制御信号である。マルチプレクサの制御信号は、何が選択されても構わない場合があり、このような時はドントケアとなる。一方、レジスタファイルの書き込み信号 rwe とメモリの書き込み信号 we は、書き込む際は 1 にするが、その他の場合は 0 にしておかなければならない。レジスタやメモリはデータを取っておく

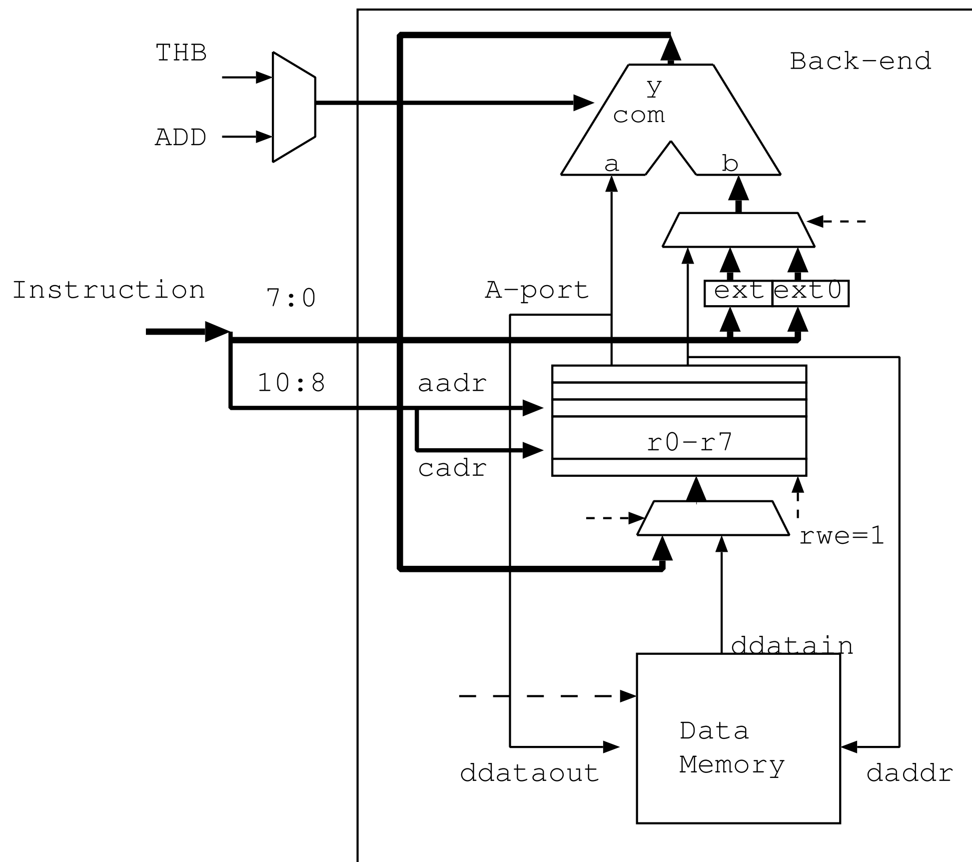


図 5.6: イミーディエイト命令の動き

のが役目なので、書き込み信号がドントケアということはあり得ず、書く場合以外には 0 にしておかなければならない。表 5.1 にそれぞれの命令に対する制御信号を示す。

表 5.1: 各命令の制御信号

	comsel	alu_bsel	rf_csel	we	rwe
ADDI	10	01	0	0	1
LDUI	01	10	0	0	1
SUB	00	00	0	0	1
LD	00	-	1	0	1
ST	–	-	-	1	0

最後に今までの POCO のデータパスの全体図を示す。線がごちゃごちゃしている印象を受けるかもしれないが、これでも CPU としては単純な部類に入る。実際、ごちゃごちゃしても線が全部書けるということは、普通の CPU ではあり得ないのだ。

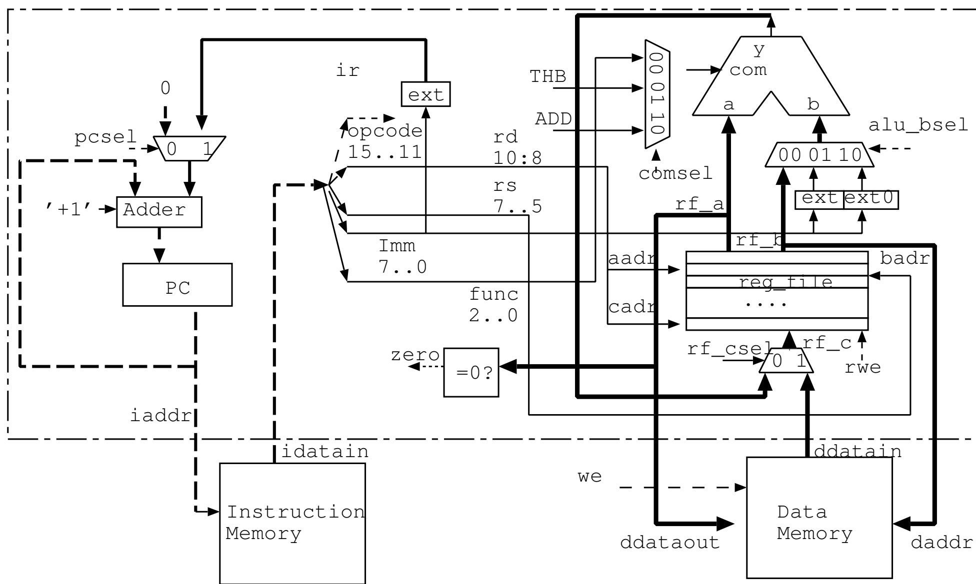


図 5.7: POCO のデータパス

演習 5.2

ADDUI, LDI を表に付け加えよ。

5.4.5　POCO の Verilog 記述

レジスタファイルの Verilog 記述

最初に図 5.3 のレジスタファイルの Verilog 記述は以下のようになる。

```
‘include "def.h"
module rfile (
 input clk,
 input [‘REG_W-1:0] aadr, badr, cadr,
 output [‘DATA_W-1:0] a, b,
 input [‘DATA_W-1:0] c,
 input we);

reg [‘DATA_W-1:0] r0, r1, r2, r3, r4, r5, r6, r7;

assign a = aadr == 0 ? r0:
aadr == 1 ? r1:
aadr == 2 ? r2:
aadr == 3 ? r3:
aadr == 4 ? r4:
aadr == 5 ? r5:
aadr == 6 ? r6: r7;
assign b = badr == 0 ? r0:
badr == 1 ? r1:
badr == 2 ? r2:
badr == 3 ? r3:
badr == 4 ? r4:
badr == 5 ? r5:
badr == 6 ? r6: r7;

always @(posedge clk) begin
if(we)
case(cadr)
0: r0 <= c;
1: r1 <= c;
2: r2 <= c;
3: r3 <= c;
4: r4 <= c;
5: r5 <= c;
```

```
6: r6 <= c;
default: r7 <= c;
endcase
end

endmodule
```

この書き方はレジスタを個別に宣言していて、あまり賢いやり方ではない。

```
reg [`DATA_W-1:0] r[0:7];
```

とメモリの形で宣言すれば、もっとずっと楽である。なぜわざわざ個別に宣言したかというと、メモリの形で宣言すると後で gtkwave でシミュレーションをする際に中身を見ることができなくなってしまうためである。通常、メモリは膨大なので、波形シミュレータでその中身をいちいち表示することはしない。そうは言っても演習時にレジスタが見えないと困るので、今回はこのようにした。あくまで演習用と考えて欲しい。

ここで always 文の中で if 文と case 文を使っている。case 文は、括弧内に示した信号について、そのデータに応じた処理をコロン：以降に書くことができる。どれも当てはまらない場合は、default：以降が実行される。大変見やすくて便利な文だが、if 文と同じく、always 文でレジスタを記述する場合およびその他の特殊な構文中でしか使えない。

we が 1 である時に、書き込みを行う cadr の値に応じて入力 c がそれぞれのレジスタに書き込まれる。もちろん書き込みが行われるのはクロックの立ち上がりに同期している。case 文は便利だが、特殊な構文中だけしか使えないため、読み出しの部分では、? : のマルチプレクサ構文を使っている。

POCO 本体の Verilog 記述

さて、このレジスタファイルを使って、図 5.7 を比較的そのまま Verilog で記述したものを示す。まず、以下のように定義をしておく。

```
`define DATA_W 16
`define SEL_W 3
`define REG 8
`define REG_W 3
`define OPCODE_W 5
`define IMM_W 8
`define DEPTH 65536
`define ENABLE 1'b1
```

```
‘define DISABLE 1’b0
‘define ENABLE_N 1’b0
‘define DISABLE_N 1’b1
‘define OP_REG ‘OPCODE_W’b00000
‘define OP_BEZ ‘OPCODE_W’b10000
‘define OP_BNZ ‘OPCODE_W’b10001
‘define OP_LDI ‘OPCODE_W’b01000
‘define OP_LDIU ‘OPCODE_W’b01001
‘define OP_ADDI ‘OPCODE_W’b01100
‘define OP_ADDIU ‘OPCODE_W’b01101
‘define F_ST ‘OPCODE_W’b01000
‘define F_LD ‘OPCODE_W’b01001
```

```
‘include "def.h"
module poco(
input clk, rst_n,
input [‘DATA_W-1:0] idatain,
input [‘DATA_W-1:0] ddatain,
output [‘DATA_W-1:0] iaddr, daddr,
output [‘DATA_W-1:0] ddataout,
output we);

reg [‘DATA_W-1:0] pc;
wire [‘DATA_W-1:0] rf_a, rf_b, rf_c;
wire [‘DATA_W-1:0] alu_b, alu_y;
wire [‘OPCODE_W-1:0] opcode;
wire [‘OPCODE_W-1:0] func;
wire [‘REG_W-1:0] rs, rd;
wire [‘SEL_W-1:0] com;
wire [‘IMM_W-1:0] imm;
wire rwe;
wire st_op, addi_op, ld_op, alu_op;
wire ldi_op, ldiu_op, addiu_op;

assign ddataout = rf_a;
assign iaddr = pc;
assign daddr = rf_b;

assign {opcode, rd, rs, func} = idatain;
```

```
assign imm = idatain[`IMM_W-1:0];

// Decorder
assign st_op = (opcode == `OP_REG) & (func == `F_ST);
assign ld_op = (opcode == `OP_REG) & (func == `F_LD);
assign alu_op = (opcode == `OP_REG) & (func[4:3] == 2'b00);
assign ldi_op = (opcode == `OP_LDI);
assign ldiu_op = (opcode == `OP_LDIU);
assign addi_op = (opcode == `OP_ADDI);
assign addiu_op = (opcode == `OP_ADDIU);

assign we = st_op;

assign alu_b = (addi_op | ldi_op) ? {{8{imm[7]}},imm} :
(addiu_op | ldiu_op) ? {8'b0,imm} : rf_b;

assign com = (addi_op | addiu_op ) ? `ALU_ADD:
(ldi_op | ldiu_op ) ? `ALU_THB: func[`SEL_W-1:0];

assign rf_c = ld_op  ? ddatain : alu_y;
assign rwe = ld_op  | alu_op | ldi_op | ldiu_op | addi_op | addiu_op ;

alu alu_1(.a(rf_a), .b(alu_b), .s(com), .y(alu_y));

rfile rfile_1(.clk(clk), .a(rf_a), .aadr(rd), .b(rf_b), .badr(rs),
.c(rf_c), .cadr(rd), .we(rwe));

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else
     pc <= pc+1;
end

endmodule
```

アキュムレータマシンと同じく、POCO は，クロック、リセットおよび命令メモリに対するアドレス iaddr, データ入力 idatain、データメモリに対するアドレス daddr, データ入力 ddatain, データ出力 ddataout、書き込み制御信号 we を入出力として持つ。

ddataout はレジスタファイルの a 出力が、daddr には b 出力が接続され、iaddr には pc が接続される。これらはもう決っているので最初に宣言する。

```
assign ddataout = rf_a;
assign iaddr = pc;
assign daddr = rf_b;
```

さて、命令メモリから読んで来た idatain を命令コード (opcode)、ディスティネーションレジスタ番号 (rd)、ソースレジスタ番号 (rs)、ファンクションフィールド (func) に分離する。

```
assign {opcode, rd, rs, func} = idatain;
```

この書き方は、複数の信号線をくっつけて束にする連結演算子 , を左辺に使っていて格好良い書き方であり、下の書き方をまとめたものとなる。

```
assign opcode = idatain[15:11];
assign rd     = idatain[10:8];
assign rs     = idatain[8:5];
assign func   = idatain[4:0];
```

このまとめた書き方は、行数が少なく、読んで分かりやすい。しかし、ビット幅がうまく合っていないと、誤った信号に分離されてしまっても、バグに気づき難い欠点もあるので利用時は注意が必要である。[1]

イミーディエイト命令時は、下位 8 ビットをイミーディエイト値として使うため、これも分離しておく。

```
assign imm = idatain[‘IMM_W-1:0];
```

もちろん、imm は fs,func と物理的には同じなのだが、意味が違うため、違った信号名にしておく。

次に opcode で命令を判別する。これが以下の部分で、これをデコーダ (解読器) と呼ぶ。デコードの結果、命令の種類毎に信号線を用意した。これで、表 5.1 の信号を簡単に発生できる。

```
// Decorder
assign st_op = (opcode == ‘OP_REG) & (func == ‘F_ST);
assign ld_op = (opcode == ‘OP_REG) & (func == ‘F_LD);
assign alu_op = (opcode == ‘OP_REG) & (func[4:3] == 2’b00);
```

[1] Verilog はシミュレーション時に左右のビット幅が合っていなくても実行されてしまい場合によっては気づかない。しかし論理合成時には Warning が出るので、この時にちゃんと見ていれば気づく

```
assign ldi_op = (opcode == `OP_LDI);
assign ldiu_op = (opcode == `OP_LDIU);
assign addi_op = (opcode == `OP_ADDI);
assign addiu_op = (opcode == `OP_ADDIU);
```

デコードしておけば、例えば ST 命令実行時にはデータメモリの書き込み信号を 1 にするという記述を以下のように簡単に記述できる。

```
assign we = st_op;
```

さて、次は ALU 周辺のマルチプレクサを記述する。alu_b は ALU の B 入力、com はコマンド入力である。A 入力は直接レジスタファイルの出力が繋がるのでマルチプレクサを記述する必要はない。

```
assign alu_b = (addi_op | ldi_op) ? {{8{imm[7]}},imm} :
(addiu_op | ldiu_op) ? {8'b0,imm} : rf_b;
```

以前から使っているように、Verilog ではマルチプレクサを ? : で表す。表 5.1 の制御信号との対応に注目されたい。3 入力以上のマルチプレクサは?を 2 回使って条件を順番にチェックしている。

ここで、まず分かり難い記述は、ALU の B 入力であろう。ここでは、ADDI と LDI の実行時には符号拡張した値を選択する。

Verilog では 8bit の imm を 16bit に符号拡張する場合、

```
{{8{imm[7]}},imm}
```

と記述する。まず imm の最上位ビットつまり 7bit 目を切り出す (imm[7])。次にこれを 8 個分並べる。これを Verilog では

```
{8{imm[7]}
```

と書くことができる。これは

```
{imm[7], imm[7], imm[7], imm[7], imm[7], imm[7], imm[7], imm[7]}
```

と同じだが、よりすっきりした書き方である。この 8bit 分を元の imm とくっつけて 16bit にするので、これはすなわち符号拡張である。一方、ADDIU, LDIU は符号拡張せず、8bit の 0 をくっつければ良いので、以下のように書けば良い。

```
{8'b0,imm}
```

これ以外は、レジスタ同士の演算ということになるので、レジスタファイルから読み出した rf_b を ALU の B 入力に入れてやる。

次に ALU の制御入力には com には、まずイミーディエイトロード命令の LDI、LDIU の場合には B 入力をそのまま素通しするため、THB(Through B) を入れる。ADDI, ADDIU の場合には加算をするので、もちろん ADD を入れる。それ以外は、レジスタ同士の演算と考えられる。この場合、アキュムレータマシン同様、命令コードの一部を入れてやる。ここでは、funct コードの下 3bit がこれに当たる。

```
assign com = (addi_op | addiu_op ) ? ‘ALU_ADD:
(ldi_op | ldiu_op) ? ‘ALU_THB: func[‘SEL_W-1:0];
```

この alu_b、com を ALU に接続する。A 入力はレジスタファイルの A ポート出力をそのまま繋ぎ、出力には alu_y という信号名を付けておく。

```
alu alu_1(.a(rf_a), .b(alu_b), .s(com), .y(alu_y));
```

次はレジスタファイル周辺の記述である。A ポートはディスティネーションレジスタを取り出すので、アドレスに rd を繋ぎ、B ポートはソースレジスタを取り出すので、アドレスに rs を繋ぐ。それぞれの出力 rf_a は直接 ALU に rf_b はマルチプレクサに接続済みである。

```
rfile rfile_1(.clk(clk), .a(rf_a), .aadr(rd), .b(rf_b), .badr(rs),
.c(rf_c), .cadr(rd), .we(rwe));
```

それではレジスタファイルに書き込む側を考えよう。書き込むレジスタ番号は rd なので、これを C ポートのアドレスに繋ぐ。次に書き込むデータは、LD 命令ではデータメモリ (ddatain) から、それ以外では ALU から来るので、これをマルチプレクサで切り替える。

```
assign rf_c = ld_op  ? ddatain : alu_y;
```

書き込み用の信号線 rwe は、結果をレジスタに書き込む全ての命令で 1 にする必要があるので、これらの論理和として表現する。

```
assign rwe = ld_op  | alu_op | ldi_op | ldiu_op | addi_op | addiu_op  ;
```

最後の部分は pc の記述である。

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else    pc <= pc+1;
end
```

ここではまだ分岐命令を導入していないので、単に毎クロックでカウントアップされる。

テストベンチ

POCO のテストベンチは、アキュムレータマシンとさほど変わっていない。

```
/* test bench */
`timescale 1ns/1ps
`include "def.h"
module test;
parameter STEP = 10;
   reg clk, rst_n;
   wire [`DATA_W-1:0] ddataout ;
   wire [`DATA_W-1:0] daddr, iaddr;
   wire we;
   reg [`DATA_W-1:0] dmem [0:`DEPTH-1];
   reg [`DATA_W-1:0] imem [0:`DEPTH-1];

   always #(STEP/2) begin
            clk <= ~clk;
   end

   poco poco_1(.clk(clk), .rst_n(rst_n), .idatain(imem[iaddr]),
               .ddatain(dmem[daddr]), .iaddr(iaddr), .daddr(daddr),
               .ddataout(ddataout), .we(we));

   always @(posedge clk)
   begin
      if(we) dmem[daddr] <= ddataout;
   end

   initial begin
      $dumpfile("poco.vcd");
      $dumpvars(0,test);
      $readmemh("dmem.dat", dmem);
      $readmemb("imem.dat", imem);
      clk <= `DISABLE;
      rst_n <= `ENABLE_N;
   #(STEP*1/4)
   #STEP
      rst_n <= `DISABLE_N;
```

```
   #(STEP*100)
   $finish;
   end

   always @(negedge clk) begin
      $display("pc:%h idatain:%b ", poco_1.pc, poco_1.idatain);
      $display("reg:%h %h %h %h %h %h %h %h",
poco_1.rfile_1.r0, poco_1.rfile_1.r1, poco_1.rfile_1.r2,
poco_1.rfile_1.r3, poco_1.rfile_1.r4, poco_1.rfile_1.r5,
poco_1.rfile_1.r6, poco_1.rfile_1.r7);
      $display("dmem:%h %h %h %h", dmem[0], dmem[1], dmem[2], dmem[3]);
   end
endmodule
```

アキュムレータマシンと同様に、

```
iverilog test_poco.v poco1.v alu.v rfile.v
vvp a.out
```

でシミュレーションの実行が可能である。
ここで、若干違っているのは、initial 文の直後に

```
      $dumpfile("poco.vcd");
      $dumpvars(0,test);
```

が入っていることである。この vcd 形式というのは、シミュレーション結果などを取っておくための標準形式の一つで、この二つの文により、波形ビュアーが利用可能になる。ここではフリーの波形ビュアー gtkwave を用いる。

```
gtkwave poco.vcd
```

で起動する。この利用法と結果表示例は web に示すが、非常に使いやすいビュアーで一通りのことならばマニュアルなしで操作できる。pc の様子、各レジスタの変化等を観察してみよ。

演習 5.3

imem.dat, dmem.dat を書き替え、A OR (A-B) の演算を行うプログラムをシミュレーションして結果を確認せよ。

演習 5.4

イミーディエイト論理演算命令

```
ORI rd,#X
```

を実装せよ。ここで、X は 0 拡張せよ。

演習 5.5

LDI, LDIU はレジスタの下位 8 ビットに数を入れるのには便利だが、これだと上位 8 ビットは、全て 1 か全て 0 が常に入ることになる。これでは困るので上位 8 ビットに、指定した値 X を入れて、下位 8 ビットは 0 にクリアするのが LDHI(Load High Immediate) である。この命令の働きを図 5.8 に示す。

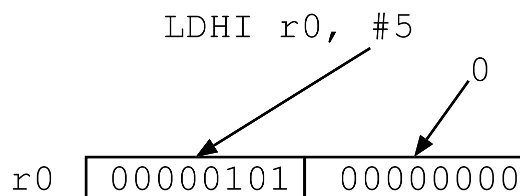


図 5.8: LDHI の操作

LDHI は以下のフォーマットとする。

```
LDHI rd,#X      rd <- X|0                          01010 ddd XXXXXXXX
```

LDHI と ADDIU の組み合わせで、16 ビットの数をレジスタ中に作ることができる。例えば、0x8090 番地をアクセスしたければ、以下のように行えば良い。

```
LDHI r0, #0x80   01010 000 10000000
ADDIU r0, #0x90  01101 000 10010000
LD r1, (r0)      00000 001 000 01001
```

さて、この LDHI(Load High Immediate) 命令を Verilog 記述に付け足して実現せよ。

演習 5.6

LDHI では下位 8bit を 0 にするが、これは時に不便な場合がある。そこで、LDHI と全く同じ操作を行うが、下位 8bit はレジスタの元の値を保持する命令 LDHI2 を実装せよ。

第 6

分岐、ジャンプ、命令の拡張

6.1 分岐命令

前の章で定義した命令はごく基本的なものだけで、これでは普通のプログラムを書くには不足する。ここではより高度な命令に拡張していく。

アキュムレータマシンと違って POCO ではレジスタ数が増えたため、アキュムレータの分岐命令はレジスタのどの中身を調べるか、を指定する必要がある。このため、レジスタ番号のフィールドを付け加えて、以下のように分岐命令を拡張する。

```
BEZ rd, X   if (rd==0) pc <- pc + X    10000 ddd XXXXXXXX
BNZ rd, X   if (rd!=0) pc <- pc + X    10001 ddd XXXXXXXX
```

ここで、もう一つ拡張の必要がある。今まではメモリのアドレス空間が狭かったため、4bit で直接指定することができた。すなわち、今までの方法は、4bit で表した X を直接 pc に入れてしまっていた。これを絶対指定:absolute(直接指定:direct) と呼ぶ。絶対指定は分かりやすいが、アドレス空間が広い場合は指定のビット数が長くなりすぎて現実的ではない。

そこで、多くのマシンで使われている方法がプログラムカウンタ (PC) 相対 (relative) 指定である。この方法では飛び先 X を直接 pc に入れるのではなく、現在の pc の値に加えた値を飛び先とする。もちろん X は符号拡張されるので、マイナス方向にも飛ぶことができる。ここで注意したいのは、多くの CPU では、分岐の起点は分岐命令ではなく、その次の命令になっている点である。跳び越す命令数は分岐命令の次の命令を起点にして数える。すなわち、X を 0 とすると、分岐しないで次の命令がそのまま実行され、-1 とすると自分自身に分岐する。

この方法では、分岐命令の飛べるアドレスの範囲は、-方向には 128、+方向には 127 に限定される。これでは困るようだが、一般的にはさほど不便ではない。これは、プログラムの動き方が極めて局所性 (Locality) が強いことから、飛び先は概ね近場に限

られているからである。とはいえ、やはり、これだけでは困るので、後でもっと別のタイプの分岐命令を導入する。

2 番地の中身と 3 番地の中身を掛け算して 0 番地に格納するプログラムは以下のようになる。

```
LDIU r0, #2
LD r1,(r0)
LDIU r0, #3
LD r2,(r0)
LDIU r3, #0
ADD r3,r1
ADDI r2, #-1
BNZ r2,-3
LDIU r0, #0
ST r3,(r0)
BEZ r2,-1
```

6.1.1　アセンブラ shapa

POCO でプログラムを書く場合、人手で機械語に直していては大変である。そこで、アセンブラというプログラムを使って、アセンブリ表記を機械語に変換してやる。POCO 用のアセンブラ shapa を web 上に置く。shapa は ruby で書かれており、web 上のプログラムは通常の Linux ではそのまま動作する。POCO.rb 中に命令の定義が書かれており、これを書き換えれば、様々な命令に対応可能である。掛け算のプログラムは以下のようにラベルを付けて書くことができる。

```
      LDIU r0, #2
      LD r1,(r0)
      LDIU r0, #3
      LD r2,(r0)
      LDIU r3, #0
loop: ADD r3,r1
      ADDI r2, #-1
      BNZ r2,loop
      LDIU r0, #0
      ST r3,(r0)
end:  BEZ r2,end
```

これを、例えば mult.prg というファイルに入れておく場合、同じディレクトリ中に shapa と poco.rb を置いておく。

そして、

```
./shapa kadai.asm -o imem.dat
```

とすると、imem.dat 中に機械語が生成される。poco1.v, test_poco.v, alu.v, rfile.v, def.h, dmem.dat をこれと同じディレクトリに置いておき、

```
iverilog test_poco.v poco1.v alu.v rfile.v
vvp a.out
```

とすればシミュレーションの実行が可能である。

6.1.2　比較と分岐

BNZ, BEZ だけでは数の大小の比較ができない。そこで、POCO では、以下の命令を設ける。

```
BPL rd, X   if (rd>=0) pc <- pc + X          10010 ddd XXXXXXXX
BMI rd, X   if (rd<0) pc <- pc + X           10011 ddd XXXXXXXX
```

これらの命令はレジスタの内容の正負で分岐するかどうかが決る。これらの命令は、レジスタの符号ビットだけを見れば良いので、実装が楽だという利点がある。一方で、大小比較を行うには、引き算を行った後に、これらの命令を使う必要が出るため、レジスタの内容を破壊してしまう欠点がある。

以下に、0 番地から並んでいる 8 つのデータの中から最大のものを選ぶプログラムを示す。r3 に最大値が格納されるが、一度引いて比較した値を再び足し戻す処理が必要となる。

```
      LDIU r0, #0x00
      LDIU r3, #0x00
      LDIU r4, #8
loop: LD r2,(r0)
      SUB r2, r3
      BMI r2, skip
      ADD r3, r2
skip: ADDI r0, #1
      ADDI r4, #-1
      BNZ r4, loop
end:  BEZ r4, end
```

6.1.3　フラグを使う方法

POCO では採用しないが、レジスタを破壊する欠点を嫌ってフラグ (Flag) を使う方法を採用するプロセッサもある。この場合、フラグとは演算の結果の性質を格納する小規模の専用レジスタを指す。良く使われるフラグは、以下の例がある。

- Zero Flag: 演算の結果が 0 になるとセットされる
- Carry Flag: 演算の結果、桁あふれがおきるとセットされる
- Sign Flag: 演算の結果、マイナスになるとセットされる

分岐命令は、このフラグがセットされていれば分岐するという形式を取る。例えば、Branch Zero ならば、Zero Flag がセットされていれば分岐する、という命令に相当する。このような分岐命令で、POCO の命令セットと同様のプログラムが書けるのは容易に理解できるだろう。

フラグは演算の結果セットやリセットされるが、フラグだけを変化させる専用の命令を設ける場合がある。これを比較命令 (Compare) と呼ぶ。例えば、Compare r0,r1 とは、r0 と r1 を比較し、等しければ Zero flag が、r0r1 が成立すれば Sign flag がセットされる。

比較命令を使えば、レジスタを破壊せずに、フラグのみをセットして条件分岐が可能である。このようにフラグを使う方法は、便利で効率が高いが、一面、命令コードの入れ替えが難しい、割り込み発生時にフラグを保存する必要がある等の問題がある。このため、POCO ではフラグを持っていない。フラグについての詳しい解説は web を参照されたい。

6.1.4　ジャンプ命令

Branch つまり分岐命令は、判断を伴うものを呼び、単純にプログラムカウンタの値を変更する命令のことをジャンプ命令と呼ぶ。ジャンプ命令は分岐命令と同様、相対指定を行うが、レジスタ指定の必要がない分遠くに飛ぶことができる。

```
JMP X       pc <- pc + X                    10100 XXXXXXXXXXX
```

POCO の場合、飛び先の番地を指定するビットを 11 ビット確保できる。しかし、この方法でも 11 ビットの範囲に入らなければ、飛べないことになり、ときに不便である。そこで、以下の最終手段を用意しておく。

```
JR  rd                pc <- rd                00000 ddd --- 01010
```

この命令は、レジスタ間接ジャンプと呼び、レジスタの値をそのままプログラムカウンタにセットする。レジスタは 16bit なので、アドレス空間のどこにでも飛ぶこと

ができる。一種の絶対ジャンプだが、レジスタの値はプログラムにより制御できるため、結果によってジャンプ先を変えたり、後に述べるようにサブルーチンコールからのリターン命令として利用可能である。

演習 6.1

アキュムレータマシンでやった演習と同じく、1 番地に X が入っている時、1+2+...+(X-1)+X を計算するプログラムを POCO のアセンブラで書け。

演習 6.2

メモリの 0 番地から並んだ 8 つの数のうち、0 の個数を数えるプログラムを書け。

サブルーチンコール

JAL と JALR

ジャンプ命令は一度飛んでいったら、戻って来ない。しかし、プログラムは一定の処理をサブルーチンとして定義し、これを様々な場所から呼び出して利用する命令が必要である。これをサブルーチンコールと呼ぶ。サブルーチンコールは、図 6.1 に示すように、呼び出して、戻ってくる機能が必要になり、このため、プログラムカウンタをどこかに保存しておく必要がある。保存場所としては特定のレジスタを利用するのが一般的である。多くの場合、最大番号のレジスタを用いるので、POCO では r7 を保存場所として定める。

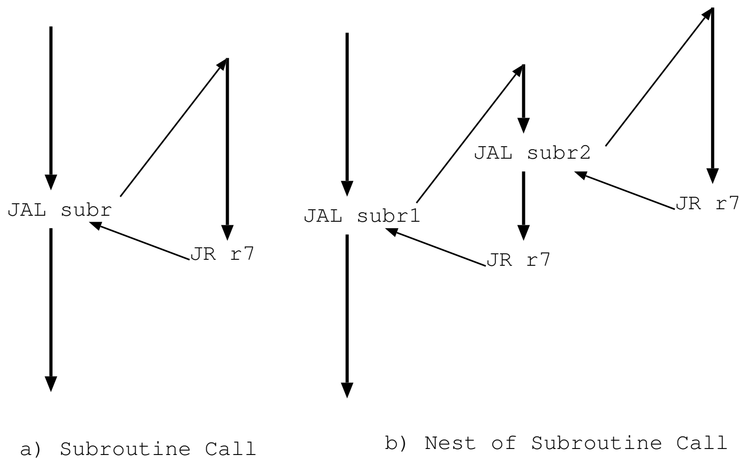


図 6.1: サブルーチンコール

```
JAL X       r7 <- pc, pc<- pc + X          10101 XXXXXXXXXXX
JALR rd     r7 <- pc,  pc <- rd            00000 ddd --- 11000
```

飛び先は JMP 命令と同様に 11 ビットを指定することができる。11 ビットではアドレス空間全体をカバーできないため、JR と同様にレジスタ間接指定を使えるようにしている。これが JALR である。この命令では飛び先をレジスタの内容で指定することができる。PC の格納場所としては、同様に r7 である。

戻り番地は r7 に格納されるため、リターン命令は JR r7 となる。下のプログラムは 0 番地のデータの 2 乗を計算するものである。掛け算のサブルーチン Mult をコールしている。このサブルーチンは r1 X r2 の結果が r3 に格納される。r1 は破壊されないが、r2 は破壊されてしまう点に注意。なお、ここではプログラムを見やすくするためにラベルを使っている。通常、ラベルはアセンブラにより、自動的に相対アドレスに変換される。

```
      LDIU r0, #0x00
      LD r1, (r0)
      MV r2, r1
      JAL mult
      ST r3, (r0)
end:  JMP end
//Subroutine Mult
mult: LDIU r3, #0x00
loop: ADD r3, r1
      ADDI r2, #-1
      BNZ r2, loop
      JR r7
```

スタック

サブルーチンは、プログラムの各所から呼び出せるという利点がある。しかし、ここで問題がある。呼び出すプログラム（メインルーチン）とサブルーチンで利用するレジスタが重なってしまうと、サブルーチンを呼び出してリターンしてきた時に、サブルーチン内でレジスタが破壊されて、プログラム実行上の情報が失われる可能性がある。そこで、サブルーチンは、呼び出された時に、利用するレジスタをどこかにしまって置く必要がある。これを実現するのがスタック (Stack) と呼ぶ。スタックは棚であり、先に積んだものを後で取り出し、後で積んだものを先で取り出すことから Last In First Out (LIFO) または First In Last Out (FILO) とも呼ぶ。スタックにデータを積む操作を push、取り出す操作を pop と呼ぶ。RISC では通常スタックは、ソフトウェアで実現する。このためには、スタックとして利用するメモリ領域を指すレジスタが必要

になる。これをスタックポインタと呼ぶ。図 6.2 の例では r6 がスタックポインタであり、プログラムをスタートする場合これをスタックとして使う領域の最後に設定しておく必要がある。

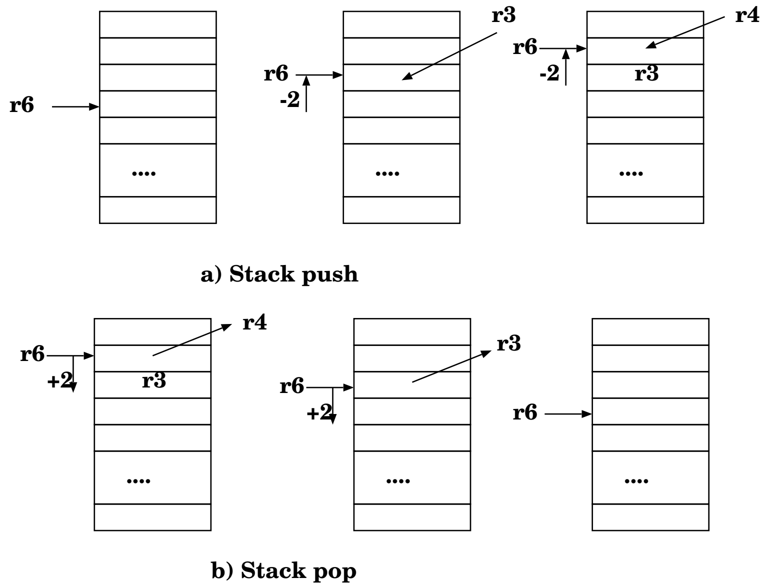


図 6.2: スタックプッシュとポップ

r3,r4 を順にスタックに push する操作は以下のようになる。

```
ADDI r6,#-1
ST r3,(r6)
ADDI r6,#-1
ST r4,(r6)
```

すなわち、あらかじめスタックポインタを減らして領域を確保し、そこにデータを格納する。逆に格納したデータを元に戻す pop 操作は以下のようになる。

```
LD r4,(r6)
ADDI r6,#1
LD r3,(r6)
ADDI r6,#1
```

スタック操作の優れた点は、サブルーチンの入れ子（ネスト）に対応可能な点である。サブルーチンの中でサブルーチンを呼んでも、呼ばれたサブルーチンで利用する

レジスタが順にスタックに積まれ、逆順で戻るため、スタック領域を使いつくすまでいくらでもサブルーチンを呼ぶことが可能である。実際、再帰呼び出し (recursive call) を行う場合、多くのデータがスタックに積まれることになる。ここで、戻り番地は r7 に格納されていることから、忘れずに r7 をスタックに push し、リターン操作である JR r7 を実行する前に pop して元に戻しておくことが必要である。

6.2　BNZ/BEZ 命令の実行

分岐命令を実装するために、フロントエンド部分に改造を加え、命令の下位 8 ビットと PC+1 を加算して飛び先番地を計算する回路を付け加える。さらに、マルチプレクサを取り付け、分岐が成立した場合に、飛び先番地を PC に入れてやる。

BEZ 命令は、a ポートから読み出された rd が 0 かどうかチェックする。条件が成立、つまりこの場合 0 だった時に pcsel=1 として、命令の下位 8 ビットを符号拡張して加算器に入れてやる。条件が成立するかどうかは制御回路で簡単に判断できる。図 6.3 にこの様子を示す。

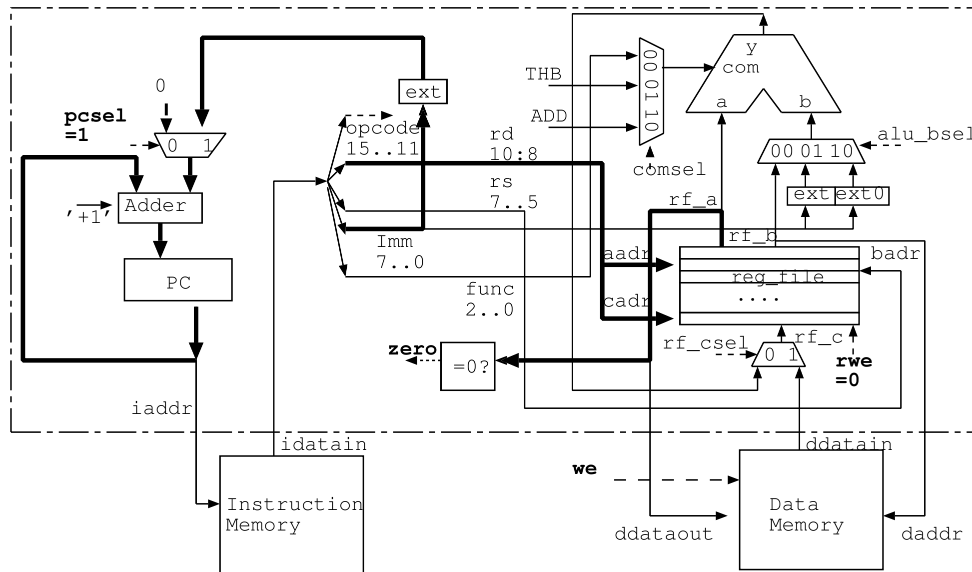


図 6.3: BEZ 命令実行のデータパス

JMP 命令は、符号拡張が 11 ビット範囲であることを除けば分岐命令とほぼ同じ回路で実現できる。JR 命令は、飛び先番地がレジスタファイルの a 出力から表われるので、これを直接 pc に入れてやる。この辺は簡単だが、JAL 命令は面倒である。こ

の命令は、ジャンプする部分は JMP 命令と全く同じ回路で良いが、その他に pc+1 を r7 に書き込まなければならない。今まで、レジスタファイルの a ポートアドレスと c ポートアドレスはいつも同じで、rd を示す命令の 10-8bit を入れていたが、JAL に限っては c ポートアドレスに 7 を入れてやる。さらに、レジスタファイルへの書き込み用のマルチプレクサを拡張して、pc+1 を入れてやる必要がある。

これらの拡張を全て行った構成を図 6.4 に示す。

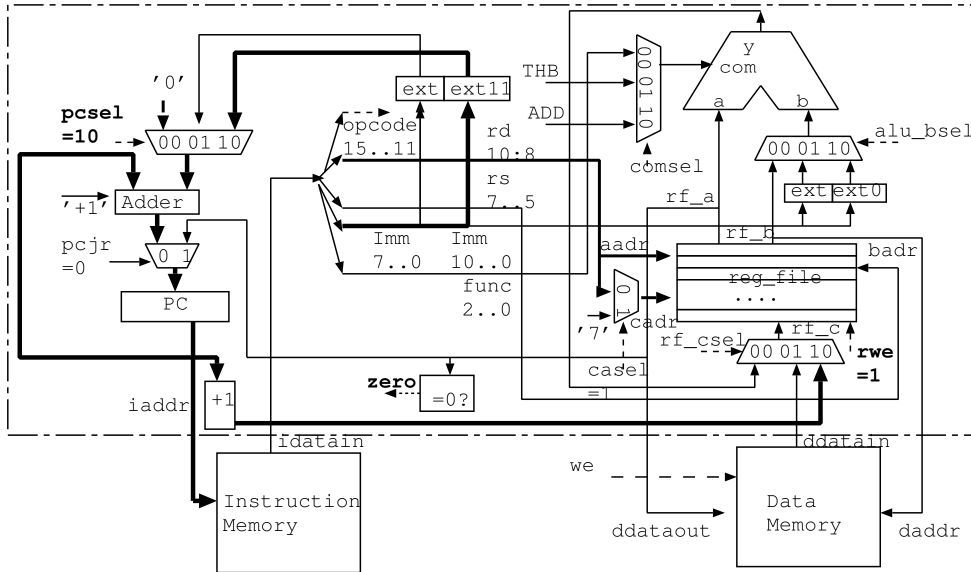


図 6.4: JAL が実行可能なデータパス

さまざまな線が増えたため、かなりごちゃごちゃした印象を与えるが、これで拡張は終わりである。あまり細部にこだわらず、今度は Verilog の記述を見てみよう。

6.2.1 Verilog の変更

図 6.4 に示す分岐命令用のハードウェアの変更はほとんどそのまま Verilog 記述に反映することができる。ここでは変更箇所のみ示す。まず wire 文で、各命令に対応する信号を宣言してから、デコーダに以下の部分を付け加える。これで、それぞれの命令に対応した信号が利用可能となる。

```
assign jr_op = (opcode == ‘OP_REG) & (func == ‘F_JR);
assign bez_op = (opcode == ‘OP_BEZ);
```

```
assign bnz_op = (opcode == `OP_BNZ);
assign bpl_op = (opcode == `OP_BPL);
assign bmi_op = (opcode == `OP_BMI);
assign jmp_op = (opcode == `OP_JMP);
assign jal_op = (opcode == `OP_JAL);
```

まず、JAL 命令用にレジスタファイルの c ポートの入力とアドレスの拡張が必要となる。そこで、以下のようにする。

```
assign rf_c = ld_op  ? ddatain : jal_op ? pc+1 : alu_y;
assign rwe = ld_op  | alu_op | ldi_op | ldiu_op | addi_op |
           addiu_op | ldhi_op     | jal_op ;
assign cadr = jal_op ? 3'b111 : rd;
rfile rfile_1(.clk(clk), .a(rf_a), .aadr(rd), .b(rf_b), .badr(rs),
        .c(rf_c), .cadr(cadr), .we(rwe));
```

すなわち、JAL 命令の際は、c ポートのアドレス cadr が 7 となり、入力には pc+1 が設定される。次に、pc については以下のように修正する。分岐命令はそれぞれの条件をチェックする。BMI や BPL は読み出してきたレジスタの符号ビット (MSB) をチェックすれば良い。

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if ((bez_op & rf_a == 16'b0 ) | (bnz_op & rf_a != 16'b0) |
             (bpl_op & ~rf_a[15]) | (bmi_op & rf_a[15]))
     pc <= pc +{{8{imm[7]}},imm}+1 ;
   else if (jmp_op | jal_op)
    pc <= pc + {{5{idatain[10]}},idatain[10:0]}+1;
   else if(jr_op)
    pc <= rf_a;
   else
     pc <= pc+1;
end
```

ここで、JMP と JAL では ir の下位 11 ビットを符号拡張している点に注意されたい。また、JR では直接レジスタファイルの a ポートから読み出されたレジスタが設定されている。always 文中であるため、if 文が使えるため、条件は見やすいと思う。

POCO の基本命令

今回で、POCO の命令上の拡張は打ち止めである。最後にまとめておく。

```
NOP                                            00000 --- --- 00000
MV  rd,rs    rd <- rs                          00000 ddd sss 00001
AND rd,rs    rd <- rd AND rs                   00000 ddd sss 00010
OR  rd,rs    rd <- rd OR rs                    00000 ddd sss 00011
SL  rd       rd <- rd<<1                       00000 ddd --- 00100
SR  rd       rd <- rd>>1                       00000 ddd --- 00101
ADD rd,rs    rd <- rd + rs                     00000 ddd sss 00110
SUB rd,rs    rd <- rd - rs                     00000 ddd sss 00111
ST rs, (ra)     rs -> (ra)                     00000 sss aaa 01000
LD rd, (ra)     rd <- (ra)                     00000 ddd aaa 01001
LDI rd,#X       rd <- X (符号拡張)             01000 ddd XXXXXXXX
LDIU rd,#X      rd <- X (符号拡張なし)         01001 ddd XXXXXXXX
ADDI rd,#X      rd <- rd + X (符号拡張)        01100 ddd XXXXXXXX
ADDIU rd,#X     rd <- rd + X (符号拡張なし)    01101 ddd XXXXXXXX
LDHI rd,#X      rd <- X|0                      01010 ddd XXXXXXXX
BEZ rd, X   if (rd==0) pc <- pc + X            10000 ddd XXXXXXXX
BNZ rd, X   if (rd!=0) pc <- pc + X            10001 ddd XXXXXXXX
BPL rd, X   if (rd>=0) pc <- pc + X            10010 ddd XXXXXXXX
BMI rd, X   if (rd<0) pc <- pc + X             10011 ddd XXXXXXXX
JMP X       pc <- PC + X                       10100 XXXXXXXXXXX
JAL X       r7 <- pc, pc<- pc + X              10101 XXXXXXXXXXX
JR  rd                pc <- rd                 00000 ddd --- 01010
JALR rd     r7 <- pc, pc <- rd                 00000 ddd --- 11000
```

演習 6.3

例題の乗算ルーティンを使って、0 番地に格納されているデータの階乗を計算するプログラムを書け。(まずは値を小さくしてテストすることをお勧めする。アセンブラ shapa を利用せよ。)

演習 6.4

JALR を poco1.v に付け加え、その機能をテストせよ。

第 7

CPU の性能とコストの評価

7.1　CPU の性能評価式

7.1.1　CPI とクロック周期

CPU の性能は当然のことながら、プログラムの実行時間により評価する。OS 等のロスを除いた純粋にユーザプログラムの実行時間は以下の式により表される。

$$CPUtime = \text{プログラムの } CPU \text{ クロックサイクル数} \times \text{クロック周期}$$

ここで、実行された命令数をカウントし、一命令あたりの平均クロック数を CPI: Clocks Per Instruction とすると、CPU 実行時間は、以下の式で表わすことができる。

$$CPUTime = \text{命令数} \times CPI \times \text{クロック周期}$$

CPI は、CPU に固有のものでなく、実行するプログラムによって変化することに注意されたい。つまり、実行時間の長い命令をたくさん用いるプログラムでは CPI は長くなる。CPI がどのようになるかは、一つのプログラム中での命令の利用頻度に依存する。

POCO の場合、全ての命令は 1 クロックで終了するので、CPI は単純に 1 となるが、通常の CPU は命令毎に実行クロック数が異なるので、どのようなプログラムで動かすかによって CPI は変わってくる。

クロック周期は、マイクロアーキテクチャの動作速度で決る。これは、実際にゲートレベルで設計して、遅延時間を見積ることで評価することができる。

例題 7.1：　CPU の性能の尺度に MIPS(Million Instructions Per Second) というのがある。これは、１秒間に何百万回命令を実行可能か、という数値である。POCO が 100MHz で動くとすると何 MIPS になるか？

答：　MIPS 値は実行周波数のメガ (Mega:10 の 6 乗) Herz 表示を平均 CPI で割れば良いので、100/1 = 100MIPS となる。ただし、この MIPS という尺度は、一定時間で実行できる命令数のみを考え、個々の命令の能力が考慮されていないため、命令セットが異なるマシン間の比較には目安程度にしか使えない。

では、POCO はいったいどの程度の周波数で動くのだろうか？周波数は回路の遅延時間の逆数になる。6 章の POCO のマイクロアーキテクチャでこれを検討してみよう。

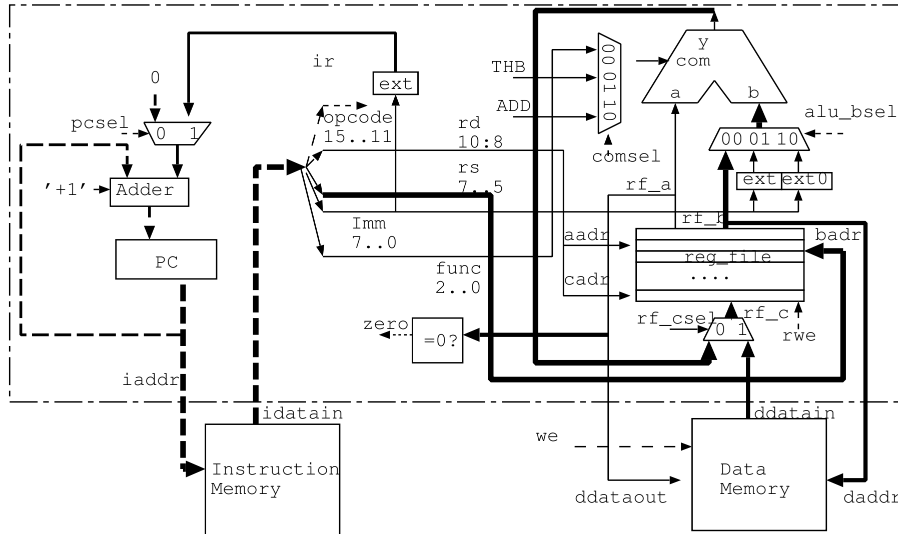


図 7.1: POCO のクリティカルパス

図 7.1 の太線が、POCO の中で、もっとも遅延時間の長い信号伝搬経路 (パス) である。このようなパスをクリティカルパスと呼ぶ。

まず、命令を命令メモリから読み出すために時間が掛かる。これを命令メモリのアクセスタイム t_{inst} と呼ぶことにする。次は読んで来た命令のレジスタ番号からレジスタファイルを読み出すのに必要な時間が加わる。これがレジスタファイルのアクセスタイム t_{rfile} である。ここから経路が二つに分かれる。

一つは、データメモリを読み出す経路である。これは、データメモリのアクセスタイム t_{data}+2 入力マルチプレクサの遅延 t_{2mux} +レジスタファイルへの書き込み時間 t_{setup} となる。

もう一つは、ALU で計算を行う経路である。これは、3 入力マルチプレクサの遅延 t_{3mux}+ALU での遅延 t_{alu}+ 2 入力マルチプレクサの遅延 t_{2mux} となる。これの大きい方を取れば良い。

例題 7.2：　t_{inst} = t_{data}= 2nsec、t_{rfile} = 1nsec、t_{2mux}=0.3nsec、t_{3mux}=0.5nsec、t_{setup} = 0.2nsec、t_{alu} = 3nsec と仮定すると、POCO の動作周波数はどのようになるか？ちなみ

に、nsec(ナノ秒) は 10 のマイナス 9 乗である。

答： データを読み出す経路は、2 + 1 + 2 + 0.3 + 0.2 =4.5nsec、ALU で計算を行う経路は、2 + 1 + 0.5 + 3 + 0.3 + 0.2 = 7.0nsec である。このため、ALU で計算を行う経路がクリティカルパスになる。動作周波数は、1/7 GHz = 0.143 GHz = 143MHz である。ちなみに、G(ギカ) は 10 の 9 乗である。

上記の演習は、適当に各部の遅延を見積った概略のものである。本当に遅延を評価するには、実装するチップを想定して論理合成および圧縮を行う必要がある。4 章で紹介した論理合成、圧縮ツールを使って POCO を合成、圧縮してみよう。合成、圧縮ツールには STA(Static Timing Analysis) 機能が付いているので、クリティカルパスをレポートしてくれる。

POCO のクリティカルパスが本質的に長いのは、全命令を 1 サイクルで終わらせてしまうためである。そこで、マルチサイクル化する方法とその Verilog 記述について web 上に示す。実際はマルチサイクル化しても性能の向上には中々結び付かないが、メモリその他の資源を共有することができるため、コストを削減することは可能である。最適化には論理合成圧縮ツールが必須であるので、この辺に興味をお持ちの読者は是非 web を参照されたい。

7.1.2 MIPS、MFLOPS、MOPS

POCO の CPI は 1 であり、動作周波数を MHz で表したものが、一秒間で何百万回命令を実行可能か、すなわち、MIPS 値と同じになる。先の例では 143MIPS ということになる。MIPS 値は、あるプログラムを実行するのに何命令必要か、という命令数についてのファクタが入らないため、評価基準として、あまり正しくない。しかし、同じシリーズの命令同士の比較には使えるため、Intel の 86 系の命令セットが主流のデスクトップやラップトップコンピュータの評価には利用される場合がある。

また、浮動小数点演算能力を持った科学技術用のマシンの性能尺度として MFLOPS(Million Floating Point OPerations Per Second) がある。これは 1 秒間に何百万回浮動小数点演算を行うことができるかを示す。決まった数値計算アルゴリズムを実行するのに要する演算数（命令実行数ではない）は、マシンの命令セットアーキテクチャに依らず一定であるので、この数値は MIPS よりは信頼できるのではないか、というのが MFLOPS の根拠である。しかし、平方根や指数関数など関数演算が入るとこの命令を持つマシンと持たないマシンが出てくるため、話が面倒になる。この問題を解決するため、関数演算を複雑さによって、浮動小数演算の数命令分としてカウントする正規化 MFLOPS という方法も用いられる。このように MFLOPS という尺度は、本来は MIPS 値よりも根拠があるのだが、スーパコンピュータなどで良く用いられるピーク MFLOPS は、そのコンピュータ出すことのできる瞬間最大性能 (逆に考えるとどのような瞬間もこれ以上の性能は出すことができない数値) で実効性能ではないので、性能尺度として

は全く信用してはならない。

さらに、同様の考え方として、信号処理用の専用プロセッサである DSP(Digital Signal Processor) などでは、1 秒間に何百万回、所定の演算を実行できるかという性能尺度として、MOPS(Million Operations Per Second) を用いる場合がある。

7.1.3　性能評価手法

POCO の場合、まだキャッシュも装備しておらず、主記憶もシミュレーションモデルなので、評価するのはある意味で簡単だが、一般的なコンピュータシステムの性能は、CPU だけでなく、搭載しているメモリ、キャッシュ、あるいは OS、コンパイラなどのソフトウェアの影響も受ける。ここで、CPU 性能をきちんと評価する方法を解説しておく。

評価用のプログラム

CPU の性能は基本的にプログラムを実行して、その時間を測ることで評価する。性能の測定に使われるプログラムは以下のようなものがある。

- 実プログラムに基づくベンチマーク集: 実際のプログラムと、その入力をセットにしたベンチマーク集。一般的な WS/PC の評価には SPEC ベンチマークが用いられる。これには非数値系の SPECint と数値系の SPECfp に分けられる。SPECint は、主として C 言語で記述されており、GCC, Latex, 表計算プログラム、CAD の最適化問題等から成り、SPECfp は FORTRAN が中心で、SPICE などの科学技術計算から成っている。一定の金額を払えば誰でも利用可能である。他にスーパコンピュータ評価用の Perfect Club、マルチプロセッサ用の SPLASH-II などがある。

- 簡単なトイプログラム: ソートや、エラトステネスのふるい、8-queen など。アセンブラプログラムで記述できるし、アルゴリズムは誰でも知っているので、ある程度再現性もある。POCO を今評価するならこの方法しか手がないが、一般的にはこの程度の簡単なプログラムでは、特にメモリシステムの挙動がからむ正確な評価を取ることはできない。

- プログラムカーネル: 実プログラムに基づく評価はリアルだが、内部処理が何をやっているのか見にくい。このため、数値計算用のマシンの評価では、様々なプログラムから繰り返し部分のみを抜き出したカーネルが使われる場合も多い。LLL(Lourence Livermore Loops)、Linpack が有名。数値計算のプロはこのループの番号を聞いただけで、その内部の処理の性質を理解することができる。スーパーコンピュータの Top500 を決めるのは Linpack を用いている。また組み

込み CPU の評価用の EEMBC や MiBench などのプログラム集もカーネルに分類される。

- 評価専用プログラム: ベンチマークやカーネルは、いくつものプログラムを動かさねばならず、しかもプログラムによって結果の順位が逆転したりして、どちらが高速が判断することが難しい。このため、様々な振る舞いを一つのプログラムに押し込んだ評価専用プログラムを作り、これを一つ動かせば評価を取れるようにしたもの。数値処理用の Whetstone、非数値用の Dhrystone が有名で、かつては評価に頻繁に利用された。しかし、これらのプログラムはコンパイラの最適化を効かなくするために、実際のプログラムではほとんど考えられないような振る舞いをする上、これらの評価用プログラムに特化した最適化を行うコンパイラまで現れた。このため、Hennessy&Patterson のテキスト [2] で厳しくこの欠点が指摘された結果、現在はほとんど使われなくなった。

7.1.4 評価結果の整理と報告

評価専用プログラムが使われなくなった現在、評価を行うためには、複数のプログラムを稼働させるのが普通である。この場合は、結果が複数得られることになるが、比較を行う場合は、どちらかのマシンに正規化して、実行したプログラムの相乗平均を取って比較するのが一般的である。これは、ただの相加平均だと基準に取るマシンを変えると比率が代わってしまうことを避けるためである。しかし、相乗平均は非線形性を持つので、出てきた結果がそのまま性能比となるわけではなく、この点には注意しなければならない。

また、評価を取ってこれを報告する場合は、ちょうど自然科学実験のレポートで、実験器具や方法を細かく記述するのと同様に、以下の点を細かく報告しなければならない。この辺は計算機アーキテクチャの世界といえどもきちんとしなければならない。

- ハードウェア:
  - 動作周波数
  - キャッシュ容量
  - メモリ容量
  - Disk 容量
- ソフトウェア:
  - OS の種類、バージョン
  - コンパイラの種類、オプション

演習 7.1　例題と同じ条件だが、メモリが遅く 4nsec であった場合、POCO の動作周波数はどうなるか？また、MIPS 値はどうなるか？

7.2　コストの評価

CPU チップとしてのコストは基本的にはゲート数（面積）により決る。次に価格は、ゲート数あるいは面積によって支配される。一般論では、半導体のコストは、面積の 4 乗に比例する。これは、面積が大きくなればなるほど、一つの wafer（半導体を作成するための板）から取れる die(チップに格納される半導体 1 個分の切片) の数が減ることと、埃等による欠損から良品が取れる割合（歩留りという）が悪化するためである。

とはいえ、新しい半導体チップは開発費およびマスク代が占める割合が大きいため、チップのコストを直接評価するのは、きわめて難しい。論理合成ツールには合成後、ゲート数 (FPGA の場合は Slice 数などの内部基本論理素子数) をレポートしてくれるので、これを基準にコストを見積る。

7.3　消費電力

次に、最近特に重要になってきた可搬型のシステム用には、消費電力も重要なファクタである。一般に性能を上げるためには多くのゲートを高速に動作させる必要があるため、消費電力が大きくなってしまう。CMOS デバイスにおいて消費電力は、

動的な消費電力＋漏れ電流による消費電力

に分けられる。

前者は、電圧の 2 乗に比例し、CMOS のゲートが ON/OFF する周波数に比例する。すなわち、ゲート数が多く、クロック周波数が大きければその分電力が大きくなる。しかし、全てのゲートが動作しているわけではないので、論理シミュレーションを行ってどの程度のゲートが動作するかを測定する必要がある。最近は、CPU の負荷が低い場合は電圧とクロック周波数を落とし、電力を節約し、高い性能が必要な時だけ電圧とクロック周波数を上げる DVFS(Dynamic Voltage and Frequency Scaling) という手法が普及している。

後者の漏れ電流は、90nm 以降のプロセスの微細化に伴ってその割合が大きくなってきた。動作をさせなくても電力を消費してしまう点でバッテリー駆動の製品では深刻な問題である。これを防ぐためには必要に応じて電力供給を切断する Power Gating や、遅いが漏れ電流の小さいゲートと速いが漏れ電流の大きいゲートを組みあわせる設計技術が利用される。

消費電力の削減は最近の CPU にとって大きな課題であり、低電力用のアーキテクチャの研究開発が盛んである。

第 8

入出力：I/O

8.1 I/O とは

今までこのテキストはCPU のみのシステムを扱っていた。しかし、実際のコンピュータシステムは、I/O すなわち、入出力なしには意味のある作業を行うことができない。

簡単なコンピュータシステムでは、I/O は図 8.1a) に示すように、CPU- メモリを結ぶバス上に設ける。高性能のシステムでは図 8.1b) のように、プロセッサ-メモリ間のバスの性能を阻害しないためと、標準化されたインタフェースを用いるために、標準I/O バスをブリッジを介して接続する方法もある。パーソナルコンピュータにおける代表的な I/O バスは PCI バスで、様々な機能の I/O ボードを接続することができる。代表的な I/O は、キーボード、ビットマップディスプレイ、ディスク、モデムインタフェース（RS232C）、Ethernet、A/D 変換器等を接続するパラレルインタフェースなどがある。ここでは、PICO にごく簡単な I/O モデルを接続することで、I/O 接続の基本を紹介する。実際は、I/O は接続する機器によって性質が異なり、構造、動作も奥が深い。

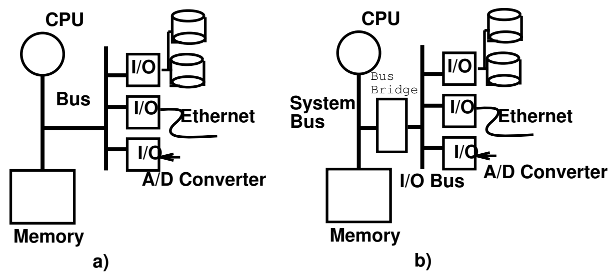


図 8.1: I/O の構成例

8.1.1　I/O モジュールの接続法

I/O の接続法は、大きく以下の２つに分けることができる。

- メモリマップト I/O: メモリと同一アドレス空間で I/O をアクセスする。このため、I/O アクセスのための命令を専用に設ける必要はない。
- プログラムド I/O(I/O マップド I/O): I/O 独自のバスあるいは I/O アドレス空間を持ち、専用の I/O 命令 (IN 命令、OUT 命令) の実行によって I/O アクセスを行う。I/O とメモリの空間が分かれていることから、セパレート I/O と呼ばれる場合もある。

前者の方法は簡単だが、広大なメモリの空間に比較的小規模な I/O をマップするため、メモリ空間に虫食いができる問題がある。これに対し後者の方法は、専用の命令、バス等を必要とする。今回は付加命令の必要のない前者の方法を用いる。

8.1.2　I/O モジュールのモデル

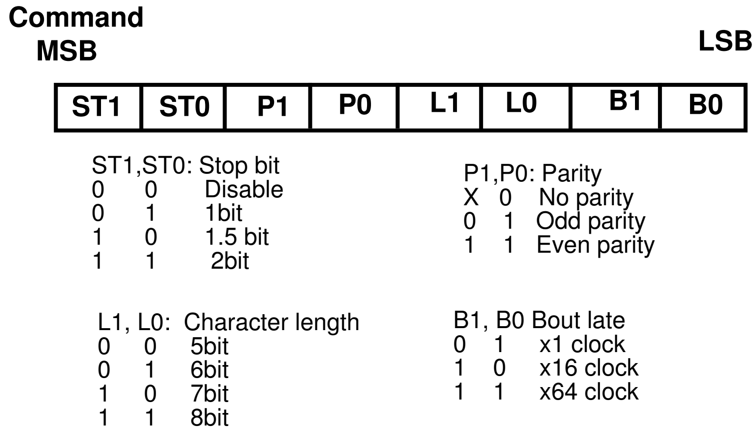
簡単な I/O モジュールのモデルを示す。I/O モジュールは、多くは以下のレジスタを持つ。

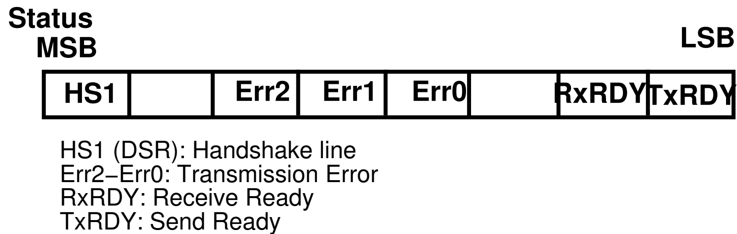
- コントロール（コマンド）レジスタ: 多くの I/O は、切り替え機能やパラメータを持っており、プログラム可能になっている。このプログラム用に、設定データを書き込むためのレジスタ。通常は、書き込み専用である。
- ステータスレジスタ : その I/O の状態を CPU が知るためのレジスタ。例えばデータが到着したかどうか、などは、このレジスタをアクセスすることにより知ることができる。データが到着したか、データの転送が終了するかなどを示す小規模なレジスタをフラグと呼ぶ。これは分岐命令の所で軽く紹介したフラグと同じ意味である。読み出し専用。
- データレジスタ: 実際にその I/O との間で交換するデータを受け渡すレジスタ。I/O の種類によってビット幅も異なる。通常は双方向。

これらのレジスタは、異なったアドレス空間に割り付けられ、CPU からはアドレスによって識別される。I/O モジュールによっては、コマンドレジスタとステータスレジスタは同一番地を共有し、書くとコマンド、読むとステータスになることもある。図 8.2 に、古典的なシリアルインタフェース（モデム用:RS232C) のコントローラのコマンドとステータスの例 (Intel 社 8251 の一部) を示す。モデムは音声信号にデータを変換し、電話線を用いてデータを送る装置であり、このためには、データを 1bit ずつ転送するシリアルインタフェースが必要である。この規格が RS232C であり、図中に示すタイミング図のように、1bit 分のスタートビット (L レベル) の後、データ（この

場合はキャラクタ：最大 8bit) を順に転送し、最後にストップビット (H レベル) を一定 bit 分転送する。

このコントローラでは、コマンドによって、ストップビットの長さ (ST1,ST0)、転送データ (キャラクタ) の長さ（L1,L0)、パリティの有無、種類 (P1,P0)、転送周波数 (ボーレート)(B1,B0) の設定が可能である。一方、同じ番地を読み出すと、ステータスレジスタを読むことができ、これにより、データ到着 (RxRDY)、送信終了 (TxRDY)、エラー (Err2-0) を知ることができる。





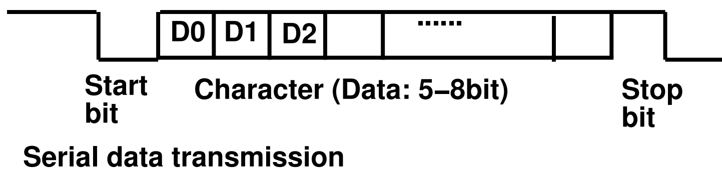


図 8.2: シリアル I/O の例

この I/O モジュールのプログラムは、以下のようにして行う。

- データの受信

  - ステータスレジスタを読み込むことにより、データの到着を知る。
  - データレジスタからデータを読み込む。これと共に次のデータが到着可能となる。これを繰り返す。

- データの送信
  - ステータスレジスタを読み込むことにより、バッファが空で次のデータが送信可能かどうかを確認する。
  - データレジスタにデータを書き込む。これによりデータの送信が起動される。

8.1.3 演習用のディスプレイ

さて、次は今回演習に用いる I/O モデルを紹介する。これは極めて単純なものとした。

- 0x8000 番地に ASCII コードのデータを書き込み、一定の時間が立つとディスプレイに表示される。
- 表示されるまでの間、次のデータの書き込みはできない。これを示すステータスレジスタ (フラグ) がやはり 0x8000 番地に割り当てられている。すなわち、この I/O は、データとステータスが同じ番地に割り当てられている。ここでは、0x8000 番地の一番下のビットがフラグであり、これが 0 になっていれば出力中で、1 ならば書き込み可能である。

さて、ASCII コードとは、8(7) ビットで表される文字コードで、英文字と数字を表すのに利用される。man ascii と打ち込んでみよう。対応が表示される。
以下のプログラムを実行すれば、A と B が出力されるはずである。

```
        LDHI r0,#0x80
LOOP:   LD r2,(r0)
        BEZ r2,LOOP
        LDIU r3,#0x41
        ST r3,(r0)
LOOP2:  LD r2,(r0)
        BEZ r2,LOOP2
        LDIU r3,#0x42
        ST r3,(r0)
END:    JMP END
```

このプログラムでは最初の LOOP でフラグをチェックし、1 ならば A(0x41) を、次の LOOP2 でもう一度フラグをチェックして、1 ならば B(0x42) を出力している。ディスプレイを摸擬するために、disp.v を使っている。このため、iverilog でシミュレーションを実行するには、web から test_poco.v と disp.v を取って来て、

```
./shapa ex1.prg -o imem.dat
iverilog poco_test.v poco1.v alu.v rfile.v disp.v
vvp a.out
```

やれば良い。

演習 8.1

メモリの 0 番地から順に格納された文字コードを 5 つ、ディスプレイに出力するプログラムを書け。プログラムは大したことはないが、shapa を使うことをおすすめする。表示する文字は何でも良いが、面倒ならば web 上の dmem.dat を使うと良い。

8.2 バイトアドレッシング

8.2.1 LB と SB

前回のプログラムは、8bit=1byte の I/O をアクセスするのに、ワード単位の LD、ST 命令を用いている。先に挙げた ASCII コードの文字列や I/O データなどは 8bit が多く、これをいちいち 16bit(あるいは 32bit) のワードで扱うとメモリの無駄が多い。そこで、一般的な計算機ではバイト単位のデータの読みだし、書き込みを行なう命令を持つ。POCO でも以下の命令を用意する。

00000dddaaa01110	LB rd,(ra)	Load Byte	8bit データの読みだし
00000sssaaa01011	SB rs,(ra)	Store Byte	8bit データの書き込み

表 8.1: バイトアクセス命令

8.2.2 メモリのアドレッシング

今まで POCO は、データも命令も 16bit を 1 ワードとして、ワード単位にアドレスを割り当ててきた。しかし、実はこれは一般的な方法ではない。世の中には ASCII コードをはじめとして 8 ビットのデータを扱う機会も多く、ワード単位にアドレスを付けると効率が悪いためである。

そこで、今回 POCO のデータメモリのアドレスをバイトアドレッシング方式に変更しよう。この方式は、8bit 単位にアドレスが振る方法で、汎用プロセッサでは一般的に使われる。さて、8 ビット単位にアドレスを降っても、データメモリの幅自体は 16 ビットなのでこれを二つに分離して順序付けを行う必要がある。メモリ中のデータの順序付けの方法には、図 8.3 に示すように以下の二つがある。

- Big Endian: 2 進数として見た場合の最上位 bit(MSB:Most Significant Bit) を 0 番地と考え、最下位 bit: (LSB:Least Significant Bit) を大きい方の番地にする。大きい方の番地が LSB 側であることから Big Endian と呼ぶ。
- Little Endian: 2 進数として見た場合の最下位 bit(LSB) を 0 番地と考え、大きい方の番地を MSB 側に持ってくる方法。小さい方の番地が MSB 側であることから Little Endian と呼ぶ。

図は 16 ビットなので右と左を入れ替えただけだが、32 ビットメモリ、64 ビットメモリの場合を考えると順番の問題が理解できると思う。この二つの方法は、市販の CPU で統一されていないため、しばしば混乱を引き起こす。違った Endian 同士でデータを交換すると、桁がひっくり返ってしまう可能性がある。混乱を嫌って、立ち上げ時に指定することで、両方用いられるようにしている CPU もある。POCO では Big Endian を用いることにする。この方法では、16bit データを扱う場合は、{ 偶数番地 8bit、奇数番地 8bit} の組を一つの 16bit データとして偶数番地で代表させる。この時、奇数番地から 16bit をアクセスしようとすると、メモリを 2 回読み出す必要が生じてしまう。このように、メモリの境界に合わないデータ配置のことをミスアラインメントと呼ぶ。ミスアラインメントを許すかどうかは、CPU 毎に異なるが、POCO では実装が面倒になるため、許さないこととする。このため、以後、16 ビットデータをアクセスする場合は、偶数番地しか許さなくなる。これは注意が必要である。

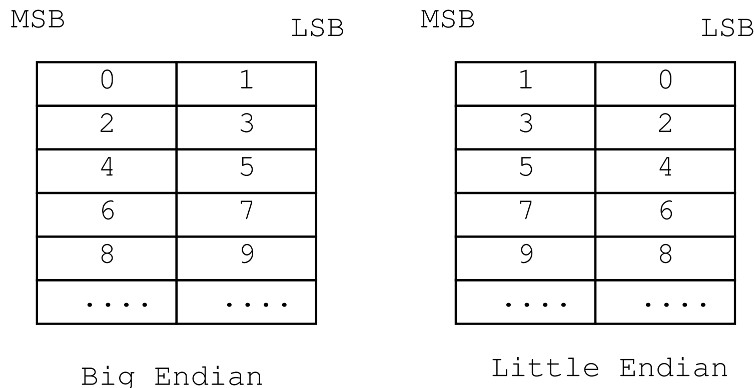


図 8.3: メモリの番地付け

もう一つ、問題がある。SB の場合は格納されないレジスタの上位 8 ビットは単純に切り落としてしまえば良い。しかし、LB の場合、意味のあるデータは 8 ビットなのにレジスタは 16 ビットである。上位 8 ビットはどうすれば良いだろうか？通常は、イミーディエイト命令同様、符号拡張とゼロ拡張を用意しておく。すなわち LB 命令は符号拡張し、他に符号拡張しない LBU(Load Byte Unsigned) を設けておくことにしよう。ここでは以下のように定義しよう。

```
00000dddaaa01101 // LBU rd,(ra):  Load Byte Unsigned
```

ちなみに、この改造で POCO のデータメモリ空間は 2^{16}=64K × 8bit となり、メモリの実体は 64K × 16bit の半分となる。さらに、やや変だが、命令メモリはワードアドレッシングのままとしておく。これは、命令は全て 16 ビットなのでバイトアドレッシングを導入しても意味がないためである。

LB、SB 命令を用いて前回用いたの出力装置への出力を行ってみよう。メモリマップト I/O では、I/O もバイトアドレッシングで番地が割り当てられる。ここでは LSB 側の 0x8001 番地に割り当てられているとしよう。

メモリ中の 5 個のデータ（バイトデータ）を出力するプログラムは以下のようになる。試してみよう。

```
        LDHI r0,#0x80
        ADDI r0,#1
        LDIU r1,#0
        LDIU r4,#5
LOOP:   LB r2,(r0)
        BEZ r2,LOOP
        LB r3,(r1)
        SB r3,(r0)
        ADDIU r1,#1
        ADDI r4,#-1
        BNZ r4,LOOP
END:    JMP END
```

8.2.3 LB/SB の実装

LB/SB の実装は案外面倒である。これは、以下の問題があるからだ。

- SB で偶数番地に書き込む場合は奇数番地に書き込んではならず、奇数番地に書き込む場合は偶数番地に書き込んではならない。

- SB で偶数番地に書き込む場合、メモリの 15bit-8bit にレジスタの下位 8 ビットの値を格納しなければならない。これは今までの ST 命令や奇数番地への SB と違った場所である。
- LB で偶数番地を読む場合、メモリの 15bit-8bit をレジスタの下位 8 ビットに格納しなければならない。これは、今までの LD 命令や奇数番地からの LB と違った場所である。

このため、まず、データメモリを 8 ビット単位に分割し、それぞれへの書き込み制御信号を別々に持たせる。すなわち、we0(偶数番地：上位 8 ビット用) と we1(奇数番地:下位 8 ビット用) が H レベルの場合にそれぞれのメモリにデータが書き込まれるとする。また、偶数番地の読み書き用にはビットをシフトする必要がある。この辺は、16 ビットなので楽だが、32 ビット以上ではさらに複雑（というか面倒）になる。
以下、関連の記述を示す。

```
..
wire lb_op, lbu_op, sb_op;
assign sb_op = (opcode == ‘OP_REG) & (func == ‘F_SB);
assign lb_op = (opcode == ‘OP_REG) & (func == ‘F_LB);

assign ddataout = (sb_op & ˜rf_b[0]) ? {rf_a[7:0],8’b0}: rf_a;
assign we0 = st_op | sb_op & ˜rf_b[0] ;
assign we1 = st_op | sb_op & rf_b[0] ;
..
assign rf_c = ld_op  ? ddatain :
        lb_op & ˜rf_b[0] ? {{8{ddatain[15]}},ddatain[15:8]}:
        lb_op & rf_b[0] ? {{8{ddatain[7]}},ddatain[7:0]}:
        jal_op ? pc+1 : alu_y;

assign rwe = ld_op  | alu_op | ldi_op | ldiu_op | addi_op | addiu_op | ldhi_op
                | lb_op | jal_op ;
```

さらに、データメモリも 8 ビットのメモリ×2 に変更する必要があるため、test_poco.v も変更する必要があるのだが、これは各自チェックされたい。

演習 8.2

POCO1.v を改造し、LBU を実装せよ。テストは imem_enshu_test.dat を imem.dat に、test_enchu_dmem0.dat を dmem0.dat に名前を変えて行え。r3 が 00ff になれば正解である。

8.3 割り込み

8.3.1 割り込みとは？

今までのプログラムでは、I/O モジュールのデータ送信が終了したかどうかを、モジュールのコントロールレジスタを常にチェックすることで検出した。このような操作をポーリング (Polling) と呼ぶ。しかし、これでは、CPU が入出力を行うためには、常に I/O モジュールのコントロールレジスタを定期的にチェックしなければならないことになる。適切な時期にチェックに行くためのプログラムを書くのは大変であるし、しょっちゅうチェックに行くと性能が低下してしまう。そこで、このことを防ぐためには、I/O の方から、入出力を行う必要があることを CPU に対して積極的に教える機構が必要になる。これが割り込み (Interrupt) である。普通、CPU は Operating System か、少なくとも簡単なモニタプログラムの下でユーザプログラムを動かしており、割り込みが発生すると、通常、OS またはモニタプログラムが呼び出されて、必要な I/O 処理を行う。割り込みは、計算機の歴史の中でもかなり初期の段階で導入された概念だが、後に必要性によって、以下の類似の機能が装備されるようになった。これらは、ひっくるめて例外処理 (Exception) として扱われる。

- I/O 割り込み: I/O モジュールからの要求によって CPU の動作を変える。
- トラップ／割り出し: プログラムから割り込みをかけて、Operating System を呼び出す。
- Page Fault: Memory Management Unit を用いて仮想記憶に基づく記憶管理を行っている際に、実メモリ上に存在しない物理アドレスをアクセスした。この場合、計算機の管理をする基本ソフトウェアである OS(Operating System) が、ディスクなどの補助記憶から必要なページを持ってくる必要がある。この機構は、キャッシュの記憶階層を紹介する部分で触れるが、本格的な理解のためには、OS を学ぶ必要がある。
- Segmentation Fault: OS で管理されている、そのユーザが使ってよいメモリ領域以外をアクセスした場合に生じる。UNIX 上で C 言語で誤ったプログラムを書くと往々にしてこのメッセージが表れる。
- Break Point: 特定の番地にしかけておき、そこをアクセスすると割り込みが生ずる。デバッグ時に利用する。
- Trace: 命令を一つ実行する度に割り込みがかかる。

これらの例外処理は基本的に同様な機構で扱うことができる（厳密に言うと、Page Fault などの実行中に生じる例外処理は、I/O 割り込みとは性質が違う）ので、ここでは I/O 割り込みについてのみ実例を示す。

8.3.2　割り込みの方式

割り込みをかけるためには、I/O モジュールから CPU への割り込み要求線が必要である。I/O モジュールは、この要求線に信号を送る（例えば L レベルにする）ことにより、CPU に割り込みの存在を知らせる。図 8.4 は、最も単純な実装例である。この場合、複数の I/O モジュールは一本のオープンドレイン（コレクタ）線を共有している。オープンドレイン（コレクタ）はディジタル回路の出力の構成法の一つで、複数の出力を一つにまとめることができ、どれかが一つの出力でも L にすると全体が L になる。

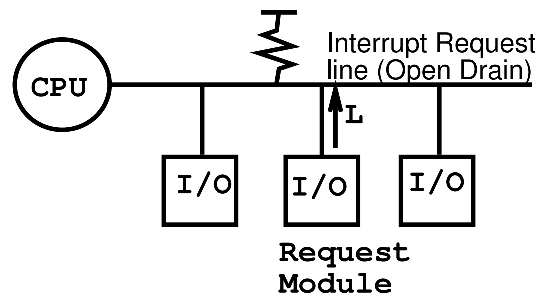


図 8.4: 割り込み要求線

CPU は、割り込み信号線が L になっており、内部状態として割り込みが許可されていれば、戻り番地と分岐判定用のフラグをどこかに保存してから、強制的に割り込み処理ルーチンにジャンプする。割り込みは、サブルーチンコールと違ってどこでかかるかわからない。分岐判定直前にかかるかもしれないので、分岐判定にフラグを用いる場合は、この情報も保存する必要がある点に注意しなければならない。幸いにして今回の PICO の実装は、比較結果はレジスタに反映されるので、そのレジスタを保存すれば済む。

メインルーチン実行中に割り込み要求が発生した場合の処理の様子を図 8.5 に示す。

割り込み処理ルーチンにジャンプすると、普通は、割り込みは強制的に禁止 (Disable) される。これは禁止しないと、割り込み処理ルーチンに飛んだと思ったら次の命令で再び割り込みがかかってしまうためである。複数の I/O モジュールが存在する場合は、割り込み処理ルーチン内でどのモジュールが割り込みをかけたかを知るために、CPU は I/O モジュールのコントロールレジスタを順に探して行く。この辺は割り込みを持たない演習プログラム同様であるが、割り込みがかかったのだから、必ずどこかのモジュールが要求しているはずなので、そこで当たるはずである。この点が、要求が存在しないかもしれないのにチェックを繰り返す単純なポーリングとは本質的に異なっている。

要求を出したモジュールを見つけたら、CPU は、その要求を満足するアクセスを

I/O モジュールに対して行う。このことにより I/O モジュールは割り込み要求を引っ込めるはずなので、CPU は、割り込みから復帰後に、元々行っていたプログラムを再実行することができる（もちろん割り込み処理中に別のモジュールがまた要求を出すこともある。この場合は復帰したらすぐまた割り込み処理ルーチンに飛ぶことになる。)。

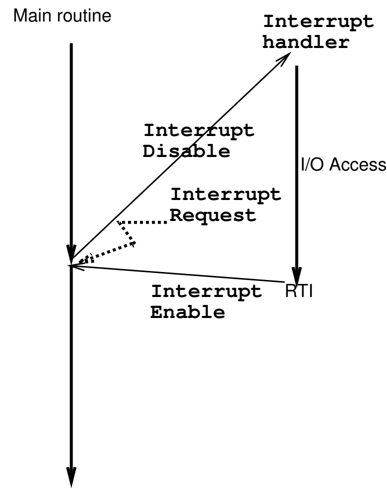


図 8.5: 割り込み処理の様子

戻り番地とフラグの保存場所は以下の２つの方法が一般的である。

- 専用のレジスタ IAR(Interrupt Address Register)、IFR(Interrupt Flag Resiter) などに格納する。POCO は分岐にフラグを用いないため、IFR は不要である。
- システムスタックに格納する。

サブルーチンコールの戻り番地をレジスタに格納する方法を取っている CPU は前者、サブルーチンコールにシステムスタックを用いる方法の CPU は後者を用いる場合が多い。ただし、汎用のレジスタを、戻り番地の保存に使うことはしない。これは、サブルーチンコールと違って、割り込みはいつかかってくるかプログラマの予想がつかないため、戻り番地の保存に使うことを決めたら、そのレジスタはそれ以外の用途には全く使えなくなってしまうためである。

次に割り込み処理ルーチンへのジャンプの方法も様々である。

- 固定番地にジャンプする。
- 特定のレジスタ又はメモリの番地に書かれている番地にジャンプする。

後者の方法は、割り込み処理ルーチンの配置番地の柔軟性が高くなるが、割り込み処理の実装がめんどうである。

割り込み処理からの復帰は、専用の命令（例えば RTI:ReTurn from Interrupt) を用いる場合が多い。これは、割り込み処理からの復帰は、PC を (分岐にフラグを用いる場合フラグも) 元に戻し、次の割り込みを許可する複合操作が必要とされるためである。

さて、今回 POCO では、最も単純な割り込みの実装法を採用する。すなわち、(1) 戻り番地は専用レジスタの IAR に格納し、(2) 割り込みがかかった際には固定番地に飛ぶ。

先に述べたように、POCO では分岐にフラグを用いていないため、フラグを保存する必要がなく、この点は非常に楽である。割り込みがかかったときの飛び先番地は、ここではなんとなく 0xff00 番地とした。したがって、割り込み処理プログラムの先頭は必ず 0xff00 番地となる。また、一般的に CPU は、リセット時には割り込み禁止状態にしておく。これは、スタート時にスタックや I/O モジュールの初期化を行う必要があるが、この際には割り込みがかかっては困るためである。そこで、ある時点で割り込みを許可するための命令 EINT(Enable Interrupt) が必要である。ここでは以下の命令を設ける。

00000xxxxxx01110	EINT	Enable Interrupt	割り込みを許可
00000xxxxxx01111	RTI	ReTurn from Interrupt	割り込みプログラムから復帰

表 8.2: 割り込み命令

これらは、レジスタ指定の必要はないが、R 形式を用いることにする。さて、割り込み処理の Verilog での記述の解説は後に回して、まず、割り込み機構の働きを観てみよう。

割り込みはもちろん出力装置についても可能だが、ここではより入力時刻が予想しにくいキー入力装置を考える。

この装置は、0x9001 番地に 1 バイト分のデータレジスタが割り当てられており、人がキーを押すと、割り込み信号 intreq が H レベルになると共に、データレジスタに 8 ビットの文字列がセットされる。0x9001 番地からデータを読み出すと、割り込み信号はクリアされ、次のキー入力が可能になる。上記は典型的な入力装置の動きである。

まず、分かりやすさのため、メインルーティンは、単純に無限ループとする。

```
00000_000_000_01110 // EINT
10100_11111111111   // END:     JMP END
```

今回は何も行わないが、もちろん本来は無限ループの代わりに何か意味のある処理をやっているはずである。

次に割り込み処理ルーチンは下のようになる。これを 0xff00 番地に置く。

```
01010_000_10000000   //  :      LDHI r0,#0x80
01101_000_00000001   //  :      ADDIU r0,#1
```

```
00000_010_000_01100  //  : LOOP:LB r2,(r0)
10000_010_11111110   //  :      BEZ r2,LOOP
01010_100_10010000   //  :      LDHI r4,#0x90
01101_100_00000001   //  :      ADDIU r4,#1
00000_101_100_01100  //  :      LB r5,(r4)
00000_101_000_01011  //  :      SB r5,(r0)
00000_000_000_01111    //       RTI
```

さて、割り込みが掛かったということは、キー入力が行われたことは間違いない。しかし、ディスプレイについては出力が可能かどうかわからないので、LOOP で確認し、可能となった後にキー入力装置から読み出してきたものを書き込んで、表示してメインルーティンに戻る。

演習 8.3

例題でやった処理は、割り込み処理ルーチン中でディスプレイの表示が終わるのを待つため、表示がキー入力よりも遅いと（そんなことは普通ないと思うが)、入力を表示し損ねる可能性がある。そこで、以下の方法を考える。

- レジスタ r3 を入力ポインタとし、r4 を出力ポインタとする。
- メインルーチンでは r3≠r4 の間は常に r4 の指すメモリの文字コードを出力し、一つ出力したら r4 をカウントアップする。
- 割り込み処理ルーチンでは、入力した文字コードを r3 の示すメモリの番地に格納し、終わったら r3 をカウントアップする。

この方法は通信用バッファで用いられる典型的な方法である。

メインルーチンは例えば以下のようになる。

```
01010_000_10000000   // 0  :    LDHI r0,#0x80
01101_000_00000001   // 1  :    ADDIU r0,#1
01010_001_10010000   // 2  :    LDHI r1,#0x90
01101_001_00000001   // 3  :    ADDIU r1,#1
01000_011_00000000   // 4  :    LDI r3,#0
01000_100_00000000   // 5  :    LDI r4,#0
00000_000_000_01110 // EINT
00000_101_011_00001  // 6  : LOOP:      MV r5,r3
00000_101_100_00111  // 7  :    SUB r5,r4
10000_101_11111101   // 8  :    BEZ r5,LOOP
00000_010_000_01100  // 9  : LOOP2:     LB r2,(r0)
10000_010_11111110   // 10 :    BEZ r2,LOOP2
```

```
00000_010_100_01100  // 11 :    LB r2,(r4)
00000_010_000_01011  // 12 :    SB r2,(r0)
01101_100_00000001   // 13 :    ADDIU r4,#1
10100_11111110111    // 14 :    JMP LOOP
```

ここでは、r0 にディスプレイの番地 0x8001 を、r1 にキー入力の番地 0x9001 を設定して初期化しておく。

割り込みルーチンを書いて、このバッファシステムをシミュレーションして見よ。割り込みルーチンでは r1 はすでに設定されているものとして良い。また、割り込みルーチンで使えるレジスタはメインルーチンと重なってはならない。

8.3.3　割り込み時のレジスタ保存

前回の課題のメインルーチンは以下のようになっていた。

```
01010_000_10000000   // 0  :    LDHI r0,#0x80
01101_000_00000001   // 1  :    ADDIU r0,#1
01010_001_10010000   // 2  :    LDHI r1,#0x90
01101_001_00000001   // 3  :    ADDIU r1,#1
01000_011_00000000   // 4  :    LDI r3,#0
01000_100_00000000   // 5  :    LDI r4,#0
00000_000_000_01110 // EINT
00000_101_011_00001  // 6  : LOOP:      MV r5,r3
00000_101_100_00111  // 7  :    SUB r5,r4
10000_101_11111101   // 8  :    BEZ r5,LOOP
00000_010_000_01100  // 9  : LOOP2:     LB r2,(r0)
10000_010_11111110   // 10 :    BEZ r2,LOOP2
00000_010_100_01100  // 11 :    LB r2,(r4)
00000_010_000_01011  // 12 :    SB r2,(r0)
01101_100_00000001   // 13 :    ADDIU r4,#1
10100_11111110111    // 14 :    JMP LOOP
```

このプログラムはメインルーチンと割り込みルーチンが協調して処理を行っている。r0 はディスプレイの番地 0x8001、r1 はキー入力の番地 0x9001 に初期化されている。また、r3 は割り込み処理ルーチンが、r4 はメインルーチンがポインタとして利用することがあらかじめ決っている。しかし、他のレジスタ、例えば r2,r5 についてはどうだろう？サブルーチンコールと違って割り込みはいつ生じるか予測が付かない。したがって r2,r5 の利用中に割り込みが発生し、割り込みルーチンでこのレジスタを利用すると内容が破壊されて、正常にプログラムが動かなくなる。前回の演習では割り込

みルーチンとメインルーチンではレジスタの番号が重ならないようにした。しかし、一般的にはこれではレジスタの数が不足する可能性がある。

ここでサブルーチンコールの際に学んだスタックを思い出して欲しい。割り込み処理ルーチンでも全く同じようにスタックを使ってレジスタを退避し、復帰する。このためにはスタックポインタをメインルーチンの開始時に設定しておく必要がある。ここではスタックポインタを r6 として、スタック領域を 7ffe 番地から下の方に向かって伸びるように設定しよう。

```
LDHI r6,#0x80
...
EINT
...
```

割り込みルーチンは、最初にルーチン内で利用するレジスタ (例えば r2 と r5 を使うとする) をスタックに退避し、RTI 前にこれを元に復帰する。

```
ADDI r6,#-2
ST r2,(r6)
ADDI r6,#-2
ST r5,(r6)
...
...
LD r5,(r6)
ADDI r6,#2
LD r2,(r6)
ADDI r6,#2
RTI
```

退避する際には 2 を（バイトアドレッシングにしたので、、）引いてから ST し、復帰の際は LD しから 2 を足す点に注意。スタックなので、積んだ逆順で戻す必要がある。

繰り返すが、割り込みはメインルーチンのどこで発生するか分からないので、割り込み処理ルーチン中では必ずこの操作を行う必要がある点に注意されたい。

8.3.4 割り込みの実装

I/O からの割り込みだけならば、実装はさほど難しくはない。命令読み出し状態 (IF) で、割り込み要求線 (intreq) がアクティブになっているかを検出し、割り込み可能状態 (int_enable が H) であれば、pc を割り込みアドレスレジスタ (iar) に保存すると共に、割り込み番地を設定する。RTI 命令を検出すると、iar に保存してあった番地を pc

に設定することでメインルーチンに復帰する。この部分について、まず pc については以下のよう拡張する。

```
wire rti_op;
assign rti_op = stat[`EX] & (opcode == `OP_REG) & (func == `F_RTI);
...
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if(jmp_op | jal_op)
     pc <= pc +{{5{ir[10]}},ir[10:0]} ;
   else if(pcsel)
     pc <= pc +{{8{imm[7]}},imm} ;
   else if(jr_op)
     pc <= rf_a;
   else if(rti_op)
        pc <= iar;
   else if(stat[`IF]) begin
        if(int_enable & intreq)
                pc <= `INTAD;
        else pc <= pc+1; end
end
```

割り込みがかかったら割り込み番地から命令をフェッチしなおすため、IF を 2 回続けて実行することになる。この制御は以下のようにする。

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) stat <= `STAT_IF;
   else
    case (stat)
      `STAT_IF: if(!(int_enable & intreq)) stat <= `STAT_EX;
      `STAT_EX: stat <= `STAT_IF;
    endcase
end
```

iar については割り込みがかかったら pc を設定すれば良いので簡単である。

```
reg [`DATA_W-1:0] iar;
...
```

```
always @(posedge clk)
   if(stat[`IF] & int_enable & intreq) iar <= pc;
```

割り込み許可フラグ int_enable は、EINT 命令で H となり、割り込みを受け付けると L となって割り込み禁止となる。RTI 命令が実行されると再び H となる。したがって以下の記述で制御する。

```
reg int_enable;
wire eint_op;
assign eint_op = stat[`EX] & (opcode == `OP_REG) & (func == `F_EINT);
...
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) int_enable <= `DISABLE;
   else if(eint_op | rti_op) int_enable <= `ENABLE;
   else if(stat[`IF] & int_enable & intreq)
                int_enable <= `DISABLE;
end
```

例外処理で面倒なのは、命令実行中に生じるページフォルトである。しかし、この例外処理については、メモリ階層の所で触れることにする。

8.3.5 多重割り込み

前回のプログラムでは、割り込み処理ルーチンはきわめて簡単なものであったので、ずっと割り込み禁止の状態で走った。しかし、場合によっては割り込み処理ルーティンが長くなり、処理を実行中にも別の割り込みを受け付ける必要がある場合もある。これを多重割り込みと呼ぶ。多重割り込みを実現するためには、CPU に割り込みレベルを設けるのが最も都合が良い。

この場合、CPU に対する割り込み線を拡張して複数持たせ、信号線によって、割り込みレベルを区別する。さらに、CPU に割り込みレベルを持たせる。メインルーチンの割り込みレベルを 0 とし、現在実行中の割り込みレベルに比べて、高いレベルの割り込み要求を検出したらその割り込みを受け付け、割り込み処理ルーティンに飛ぶ。この際に CPU の割り込みレベルは、受け付けた要求と同じレベルまで自動的に高くなる。現在実行中の割り込みレベルより低い割り込み要求は受け付けられず待たされる。

しかし、このような割り込みレベルを持っていない場合、ソフトウェアで、割り込み処理中に割り込みを許可し、疑似的に多重割り込みを許してやる方法もある。この場合、戻り番地やフラグをスタックに待避する必要があるので、IAR と（分岐にフラ

グを用いる場合はフラグ保存用レジスタも) 汎用レジスタ間の転送命令が必要になる。また、割り込み許可／禁止をプログラムで扱う必要があるため、以下の命令が必要になる。

```
00000xxxxxx10000  DINT  Disable Interrupt 割り込み禁止
00000dddxxx10001  MFIAR Move From IAR     d <- IAR
00000xxxsss10010  MTIAR Move To IAR       s -> IAR
```

とはいえ、このような命令を用いてソフトウェアで多重割り込みを実現する方法は、かなりのプログラミング能力を要する。

演習 8.4

DINT, MFIAR, MTIAR 命令を poco.v に実装せよ。

8.3.6　DMA(Direct Memory Access)

割り込みを用いて入出力プログラムを起動することにより、他の作業を行ないながら、I/O デバイスとデータのやりとりをすることができる。先に紹介したシリアルインタフェースなどの遅い I/O に対しては、プログラムでデータの転送を行なっても十分である。データ出力を例にとると、図 8.6a) に示すように、メモリの内容を LB 命令で読み出し、I/O の番地に SB 命令で書き込むことで転送を行う。ところが、ネットワークやディスクなどの高速な I/O に対しては、このように CPU を介して LB/SB 命令を用いてデータの授受を行なうのでは、デバイスの速度について行くことができない。

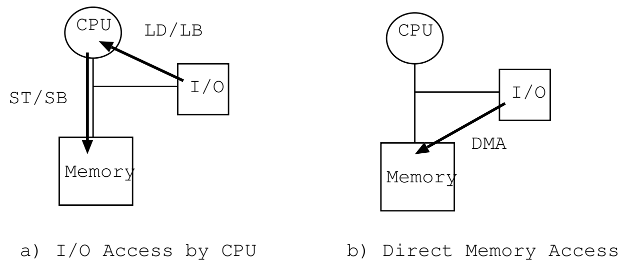


図 8.6: DMA 転送による出力

このような場合、I/O デバイスが CPU からバスの利用権を一時的に借り受け、図 8.6b) に示すように、直接メモリとの間でデータの連続転送を行なう。このような操作を DMA(Direct Memory Access) と呼ぶ。DMA は以下の手順で行なう。

1. 転送を行ないたいデバイスは、CPU に対してバスの利用権を請求する。

2. CPU は命令の区切りでバスを解放してデバイスに明け渡す。転送を行ないたいデバイスが複数存在する場合は、どれか一つを選ぶ。この操作をバスアービトレーションと呼び、選択回路をアービタと呼ぶ。

3. 利用権を獲得したデバイスは、直接メモリとの間でデータの転送を行なう。この間、CPU はバスを利用できないため、メモリ中の命令をフェッチできず、休止状態となる。

4. データ転送を終了したデバイスは、バス利用権を CPU に戻す。CPU はプログラムの実行を再開する。

5. DMA は CPU がバスを明け渡した際に行なうため、CPU はデータの転送がどうなったか全く検知することができない。したがって、一連のデータ転送処理が完全に終了した場合、データ転送を行なったデバイスは、CPU に対して割り込みをかけて、転送処理の終了を知らせる。

DMA は、転送を開始するまでの手続きに時間を要するため、ある程度まとまった大きさのデータをまとめて転送するのに用いる場合が多い。ただし、DMA を行なうためにバスを長期間占有すると、その間 CPU はまったくプログラムを実行することができないので注意が必要である。

最近のシステム LSI では、CPU と複数の I/O デバイスが同一チップ上に実装されており、これらの間で DMA を効率良く行なうために、専用の DMA コントローラを持つ。DMA コントローラの設計は、個々の I/O デバイスの性質に依存する点が大きい。

第 9
パイプライン処理

9.1　パイプライン処理の基本

POCO は 1 サイクル CPU で全ての命令を 1 サイクルで実行する (CPI=1) 代わりにクリティカルパスが長くなり、動作周波数が落ちる。web 上で検討しているが、マルチサイクル化することにより、動作周波数を改善することは可能であるが、その分 CPI が増えてしまって全体の性能向上には結びつかない。

CPI を増やさずに、動作周波数を大きく改善するためには、パイプライン処理を取り入れる必要がある。最近の CPU のほとんど全ては、パイプライン処理を基本とした構造を持っている。特に RISC の命令セットはもともとパイプライン処理の効率化を念頭に置いて作られており、80 年代中頃から 90 年代にかけてマイクロプロセッサの性能向上に最も寄与した技術の一つである。

パイプライン処理は、作業の効率を上げるために工場などで用いる流れ作業と同じ考え方である。まず作業全体をいくつかの段階（ステージ）に分割し、それぞれを担当が受持ち、前段階の処理を終えた途中結果（情報）をもらって、自分の作業を行い、次のステージの担当に送る。流れ作業の流れの速度は、最も作業時間の長いステージによって決まる。最も作業時間の長いステージの直後のステージは、そのステージから途中結果を貰えないため、作業を開始できない。一方、最も作業時間の長いステージの直前のステージでは、そのステージに結果を送ることができず、待ち状態になる。これが全てのステージに伝搬することから、結局最も作業時間の長いステージに合わせて全体が動くことになる。

すなわち、流れ作業による性能向上は、

（全体の作業時間）／（最も長いステージの作業時間）

ということになる。したがって、効率を上げるためには、各ステージの作業時間をなるべく等しくすることが重要である。全体の作業が完全に等しいステージに分割でき

れば、性能はステージ数倍になる。CPU の場合、理想的には、クロック周期をステージ数分の 1 にすることが可能である。
しかし、パイプライン処理は以下のオーバヘッドを考える必要がある。

- ステージ間のデータの受渡しにはレジスタが必要になる。このため全体のハードウェアが増え、レジスタにデータを取り込む時間が必要になる。
- 各ステージはそのステージで専用に使える資源が必要なのでハードウェア量が増大する。しかし、POCO はもともと 1 サイクル CPU で、資源の共有を行っていないため、このためのハードウェア量の増加は考えなくて良い。

しかし、通常、上記のオーバヘッドが問題ならない程の性能向上を実現することができる。

9.2　POCO のパイプライン処理

9.2.1　パイプライン構成

POCO の全体構成は、大きく考えて、命令フェッチを行うフロントエンドと実行を行うバックエンドの二つの部分に分けられる。そこで、これを素直にパイプライン化すると、図 9.1a) のように、命令フェッチ部 (IF) と実行部 (EX) をそれぞれステージと見なして、パイプライン処理する方法が考えられる。POCO では、元々pc, pc 用の加算器、命令メモリと、ALU、データメモリは独立しているため、ほとんど苦労することなしにパイプライン構造が実現できる。一点重要なのは、今までと違って取ってきた命令を蓄えて置くレジスタが必要になる。これを、命令レジスタ (instruction resgister:ir) と呼ぶ。
しかし、良く考えてみると、これは両者のバランスが悪い。IF ステージは、命令メモリを読み出し、pc に 1 あるいは飛び先を加える操作を行うだけである。これに対して EX ステージは、レジスタ間演算命令を例に取ってみると以下の操作が必要になる。(1) レジスタファイルから rs, rd で示したレジスタを読み出し、(2) ごちゃごちゃしている入力のマルチプレクサを経由し、(3)ALU で演算し、(4) 答えをメモリからのデータと切り替えるためのマルチプレクサを経由してレジスタファイルの入力にデータを与え (5) 書き込みを行う。
命令メモリとデータメモリの読み書き時間が十分高速とすると (この仮定はきちんとしたキャッシュを使わないと成り立たず、これは後で紹介する。) IF よりも EX は 2 倍から 3 倍の時間が掛かることが予想される。このため、パイプライン処理は図 9.1b) に示すようにうまく働かない。
そこで、EX ステージを三分割して、図 9.1c) のように動作させることを考える。どのように分割すれば良いだろう? 通常、EX ステージで実行する処理で最も時間を食うと考えられるのは ALU における演算である。そこで、図 9.2 に示すように ALU の

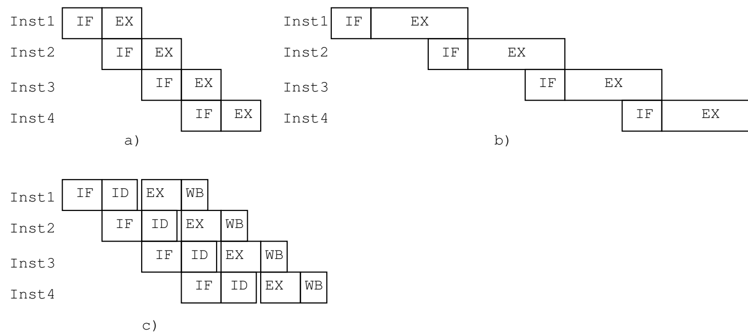


図 9.1: POCO のパイプライン処理

演算の直前 (areg, breg) と、直後にレジスタ (creg) を挿入する。そして、演算のパスを以下のように分割する。

- ID(Instruction Decode) ステージ: レジスタを読み出して ALU の入力マルチプレクサを経由してレジスタ areg, breg に値をセットする。同時に ir 中の命令コードをデコードして、ALU の com を作り、これを com_id にセットする。
- EX(Execution) ステージ: ALU は com_id に従って areg, breg の値を演算し、答えを creg にしまう。LD 命令の場合は、データレジスタからデータを読み出してこれを dreg にしまう。
- WB(Write Back) ステージ: LD 命令ならば dreg の値、それ以外ならば creg の値をレジスタファイルに書き込む。

この方法のミソは、最も時間の掛かる ALU の前後の処理を EX ステージから分離し ID ステージと WB ステージとして独立させた点にあり、CPU 全体が純粋に ALU 内の遅延時間で動作可能となる。

9.2.2 各ステージの処理

図 9.2 は、今まで解説した POCO にパイプライン用のレジスタを挿入した図であるが、パイプラインに命令が流れていく様子が掴み難い。そこで、データパスを横にしてみたのが図 9.16 である。この図は、LD,ST,LDI,LDIU,ADDI,ADDIU のみから成る単純な構造のパイプラインで、IF, ID, EX, WB の 4 つのステージが左から右に並んでいる。すなわち、IF でフェッチされた命令は左から右方向に流れていく。

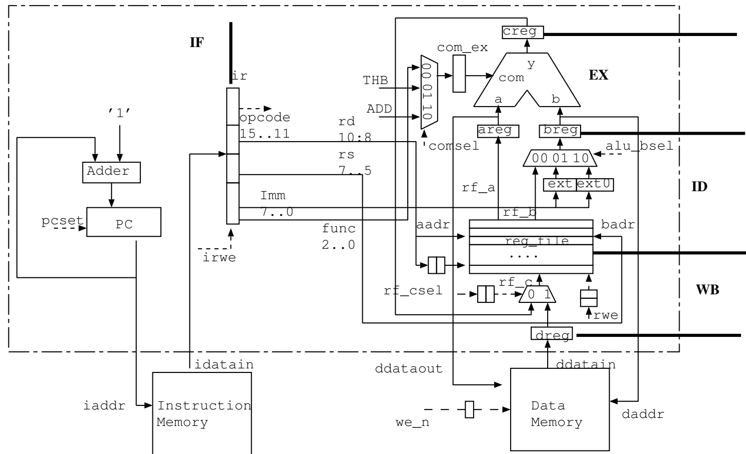


図 9.2: パイプライン化されたデータパス

IF ステージ

pc に従って命令メモリから命令をフェッチし、ir に格納する。今までとの違いは、これを毎クロック行う点にある。このため、pc は分岐命令以外は毎クロックカウントアップされる。つまり常に pcset=1, irset=1 であるので、図中では省略する。

では、分岐命令はどうだろう？実は分岐命令は次に実行する命令が分岐する場合としない場合で違ってくるので、パイプラインの動きが乱れてしまう。このため、ちょっと棚上げしておいて、後で検討することにしよう。分岐命令を考えなければ IF は非常にシンプルである。

ID ステージ

実行する命令は ir に保存されている。ID(Instruction Decode) ステージの役目は、この命令に従って、レジスタファイルからデータを読み出すと共に、ALU の com やレジスタファイルへの書き込み用マルチプレクサの制御に至るまで、すべての制御信号を生成してやることである。この処理のことを解読：デコードと呼ぶ。

基本的には今までの POCO と同様に、rd, rs に従ってレジスタファイルからレジスタを読み出すと共に、イミーディエイトフィールドの拡張を行う。レジスタの rs に従って読み出された値は、マルチプレクサを経由して breg にセットする。イミーディエイトデータが利用される場合は、符号拡張してから同様に breg にセットする。a 入力についてはレジスタファイルから読み出した値を直接 areg にセットする。ALU の com 入力についても、EX ステージですぐに演算ができるように、このステージで値を決めておき、これを次の EX ステージで使えるように保存する。

このように、デコードした結果は、今このステージで使わず、EX ステージや WB

ステージで利用する場合もある。このため、ID ステージでは様々な信号をレジスタ (1 ビットのものは Flip Flop) に格納しておく必要がある。このようにパイプライン間のデータの受け渡しに使うレジスタのことをパイプラインレジスタと呼ぶ。

ID ステージで生成した信号を後のステージに送るためのパイプラインレジスタに、ここでは_id を付けて表す。例えば ALU のコマンドは com_id という名前のレジスタに格納して EX ステージに送る。同じように命令の解読信号 (例えば rwe、st_op、ld_op など) も、全てレジスタに格納して EX ステージに渡してやる。[1]

EX ステージ

計算のためのお膳立ては全て ID ステージで整っているため、ここでは純粋に演算だけを行って結果を creg に格納する。メモリのアクセス命令の場合を考えて、breg の値をデータメモリのアドレスへ、areg の値をデータへ送ってやる。ST 命令では、メモリの書き込みを行わなければならないため、ID ステージで作られた st_op_id 信号が 1 ならば、we_n を L にしてやる。LD 命令では、読み出したデータを専用レジスタ dreg に格納する。レジスタファイルに書き込む命令では書き込むレジスタ番号の入った rd_id、書き込み信号 rwe_id をそのままここでもレジスタに格納して、id_ed、rwe_ex として次の WB ステージに送ってやる。また LD 命令かどうかを示す ld_op_id もレジスタに格納して、ld_op_ex として WB ステージに渡してやる。

WB ステージ

レジスタファイルに書き込むべき答えは、LD 命令ならば dreg に、ALU 命令ならば creg に入っている。これを ld_op_ex 信号で選んで、レジスタファイルに送ってやる。レジスタファイルは便宜上 ID ステージに書いてあるが、書き込み部分については、WB ステージから送った信号で制御する。すなわち、選んだ書き込みデータ rf_c を rwe_ex が 1 の時だけ番号が rd_ex で示されるレジスタに書き込んでやる。

9.2.3 パイプラインの動作

以下の命令がパイプライン中を流れて実行される様子を示す。太字で示す部分に有効なデータが流れている。

```
01000_000_00000000   // 0 :  LDI  r0,#0
01000_001_00000001   // 1 :  LDI  r1,#1
01000_010_00000010   // 2 :  LDI  r2,#2
00000_011_000_01001  // 3 :  LD  r3,(r0)
```

[1] この考え方で行くと areg, breg も alu_a_id や alu_b_id と呼ぶべきかもしれないが、これらは重要なのでもっと簡単に呼べるように特別な名前とした.

```
00000_000_001_00110  // 4 :  ADD r0,r1
00000_010_010_00110  // 5 :  ADD r2,r2
01100_011_00000011   // 6 :  ADDI r3,#3
```

最初の図 9.3 は命令がスタートしてから 3 クロック目のパイプラインの様子を示す。ここでは、LDI r0,#0 が EX, LDI r1,#1 が ID、LDI r2,#2 が IF で実行されている。IF では、LDI r2,#2 が命令メモリから読み出されている。次のクロックの立ち上がりで、この命令は ir に格納され、ID ステージに渡される。

ID では LDI r1,#1 の制御信号が Decorder で生成されると共に、レジスタファイルが読み出される。ただし、この場合、読み出された r1 の値は使われず、ir の下 8 ビットを符号拡張された値が breg に格納される。同時に THB を示す ALU コマンドもレジスタに格納されている点に注意されたい。

EX では LDI r0,#0 が実行されている。ここでは breg の値が、前のクロックで設定された THB 操作に従って creg に格納される。

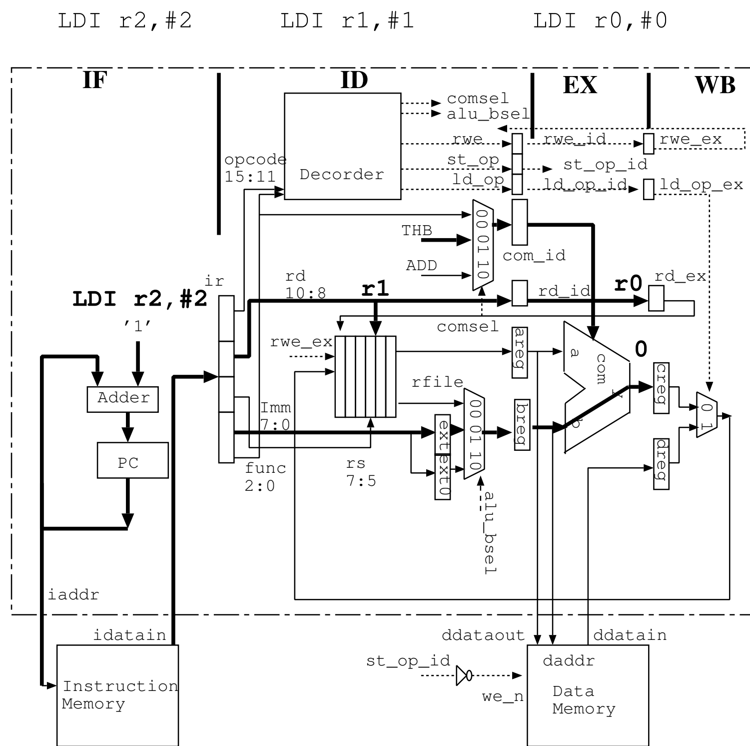


図 9.3: 3 クロック目の様子

次の図 9.4 は 4 クロック目である。ここでは、LDI r0,#0 が WB, LDI r1,#1 が EX、LDI r2,#2 が ID で実行され、LD r3,(r0) が IF で命令メモリから読み出されている。

WB では LDI r0,#0 が実行されている。ここではすでに前のクロックで結果が出た creg の値が選ばれて、レジスタファイルの r0 に格納される。EX, ID, IF の操作は基本的に前のクロックと同じである。

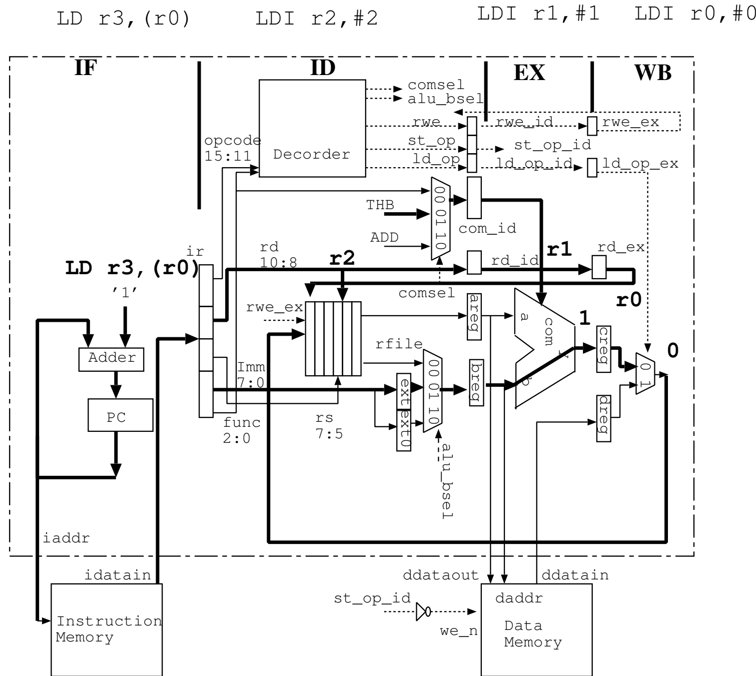


図 9.4: 4 クロック目の様子

次の図 9.5 は 5 クロック目である。ここでは、LDI r1,#1 が WB, LDI r2,#2 が EX、LD r3,(r0) が ID で実行され、ADD r0,r1 が IF で命令メモリから読み出されている。

WB では LDI r1,#1 が実行されている。ここでは先のクロック同様に、creg の値が選ばれて、レジスタファイルの r1 に格納される。EX では、前のクロック同様に LDI r1,#2 が実行され、2 が creg に書き込まれている。ID では、LD r3,(r0) が実行される。この様子に注目されたい。今までと異なり、rd, rs によって示されたレジスタの値である r3,r0 が両方共 areg, breg に格納されている。(しかし今度は breg の方は使われない。)

次の図 9.6 は 6 クロック目である。ここでは、LDI r2,#2 が WB, LD r3, (r0) が EX、ADD r0,r1 が ID で実行され、ADD r2,r2 が IF で命令メモリから読み出されている。

WB の動作は今までと同じだが、EX では LD 命令が実行されている。このために、前のクロックで格納されたアドレスがメモリに送られている (データも送られている

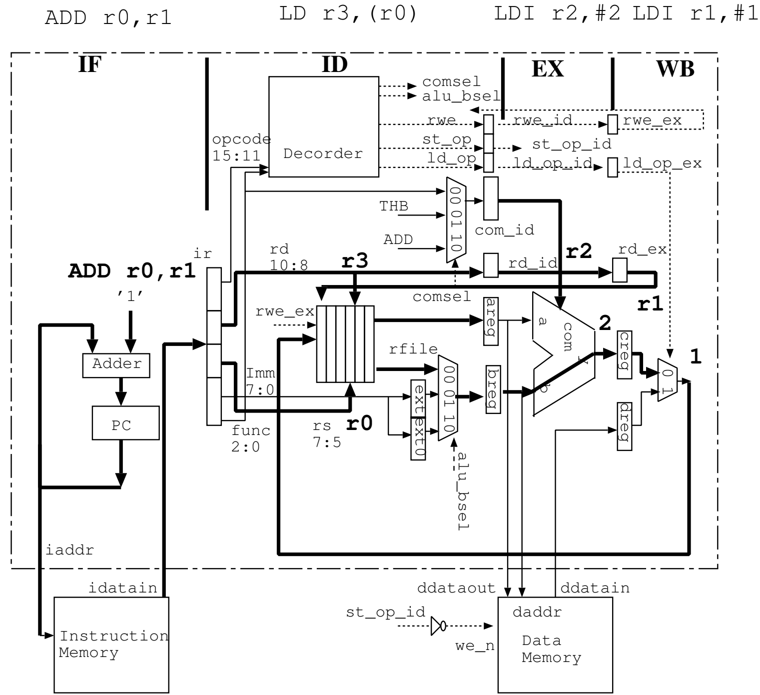


図 9.5: 5 クロック目の様子

が、LD 命令なので使われない)。読み出されたデータは dreg に格納される。また、ADD r0,r1 が実行される ID にも注目されたい。両方のレジスタが読み出されて areg, breg に設定されると共に、命令コードの funct の下のビットが ALU のコマンドレジスタに格納されている。

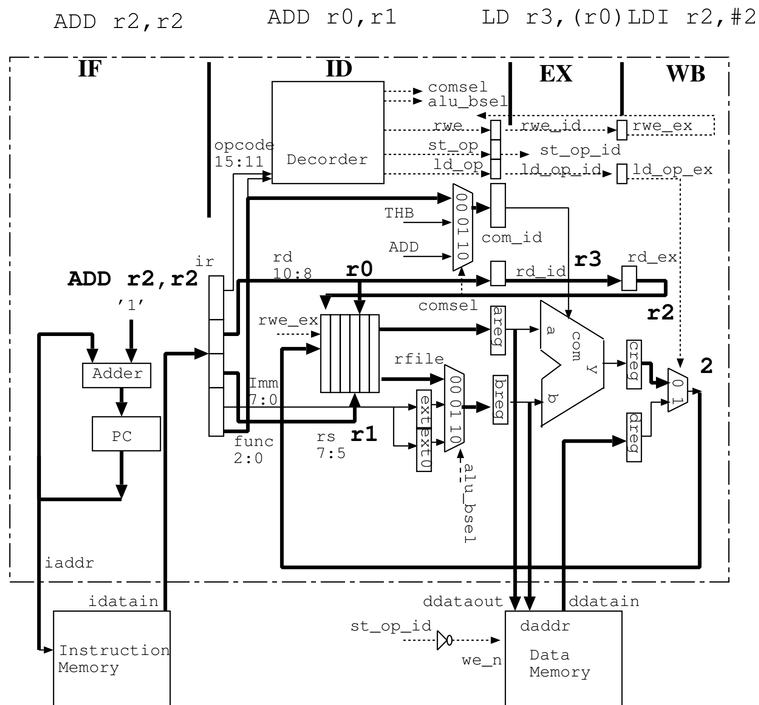


図 9.6: 6 クロック目の様子

次の図 9.7 は 7 クロック目である。ここでは、LD r3,(r0) が WB, ADD r0,r1 が EX、ADD r2,r2 が ID で実行され、ADD r2,r2 が IF で命令メモリから読み出されている。WB では dreg の値が選ばれて、レジスタファイルに格納される。はるばるパイプラインレジスタを経由して運ばれてきた ld_op_ex 信号がここで利用される。EX では ADD 命令が実行されて答えが creg に格納される。ID ではもう一つの ADD 命令が実行され、同様にレジスタとコマンドの値がレジスタが設定される。

これ以降は省略するが、毎クロック各ステージで命令が実行されている様子が理解していただけるだろう。

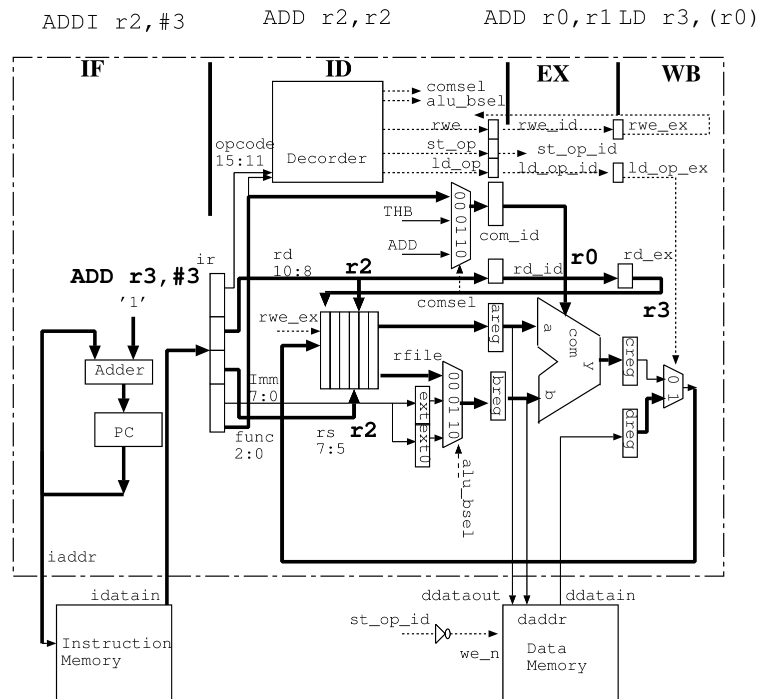


図 9.7: 7 クロック目の様子

演習 9.1

演習用の imem_enshu.dat をシミュレーションして、r0 がどのように変化するか 1 クロックずつ調べよ。

9.2.4 パイプライン処理とタスクレベルの並列処理

パイプライン処理と並んで性能を上げるために使われる方法としてタスクレベルの並列処理がある。これは、たくさんの人間や処理装置がそれぞれ独立に同じ作業を行うことで高速化する方法である。要するに手伝ってくれる人が居て、皆で同じことを同時にやれば、仕事が早く終わるわけで、これも日常的に使われている。

パイプライン処理は並列処理に比べて以下の利点を持っている。

- 並列処理は、それぞれの人間が、処理全体を通して行うことが必要とされるが、パイプライン処理は、どれか一つの作業のみを行えば良い。(人間が一つの工程のみ繰り返し行うと、単調すぎるためにかえって効率が低下する。しかし、論理回路の場合一つの作業のみで済むということは、単純な回路で済むことになって有利である。)
- 並列処理は、全ての工程を行うために必要な工具、施設、作業場所が必要であるが、パイプライン処理は、ある工程のみを行う工具、施設、作業場所があれば良い。
- 並列処理は、各人受け持つ作業が完全に独立でなければ同時に実行するのが難しいが、パイプライン処理は、一連の流れの作業を効率化することができる。

これらの利点により、工場などでは並列処理よりもパイプライン処理あるいは流れ作業を行う場合がはるかに多かった[2]。一方で並列処理の利点は以下の通りである。パイプラインは一連の作業を均等にステージに分割すれば、性能をステージ数倍上げることができるが、ステージ間の受け渡しのロスがあるため、普通はさほど多くのステージに切ることはできない。計算機の場合、多くてもパイプラインの段数は 15 程度である。これに対して並列処理は、同時に処理が可能であれば、100 人同時に仕事をすれば 100 倍に生産量を向上させることができる。このため、パイプライン処理での性能が限界に達した後に、プロセッサはまず命令レベルの並列処理を用いて性能を向上させ、最近はスレッドレベルというプログラムの大きな塊間での並列処理を利用するようになった。マルチコアプロセッサ、メニーコアプロセッサなど、最近のプロセッサは並列処理を用いている。

[2]最近は、多品種少量生産あるいは新製品の出現サイクルが短くなったこと、また、人間の達成感が得られることの効果が従来考えられていたより大きいことがわかったため、工場の流れ作業も見直されている。

9.3 パイプラインハザード

今までは、パイプラインがスムーズに流れる場合のみを考えて来た。しかし、パイプラインは常にスムーズに流れるわけではなく、様々な原因によってその動作を停止する危険が生じる。この危険のことをパイプラインハザードと呼ぶ。一般的に、パイプラインハザードには以下の三種類がある。

- 構造ハザード:パイプラインの違ったステージ間で、同じ資源を使う場合に生じるハザード。POCO ではこれは起きないが、例えば命令メモリとデータメモリに同じものを使うと起きる。つまり、LD 命令を実行している時に、次の次の命令がフェッチできなくなってしまう。構造ハザードは資源を増やせば解決できる。この場合、poco でやったようにメモリを分ければ解決できるが、もちろんその分コストが増える。構造ハザードを認めるのは、パイプラインを停止 (ストールと呼ぶ) させて若干性能が落ちても、コストが小さい方が良い場合である。
- データハザード:データの依存性によって生じるハザード
- 制御ハザード:分岐命令に関連するハザード

命令メモリとデータメモリを単一のメモリで共有した場合生じる、構造ハザードを例にハザードによる性能低下を考えよう。図 9.8 に示すように、LD 命令などメモリを利用する命令は、3 クロック目の EX ステージでメモリを利用する。このため、2 番目の命令 (Inst2) がフェッチする時にメモリを使えなくなり、一クロック待たなければならない。これは、パイプライン中の泡 (バブル) となり、このため、命令の終了は 1 クロック遅れる。バブルによりパイプラインの性能が低下することをパイプラインストールと呼ぶ。

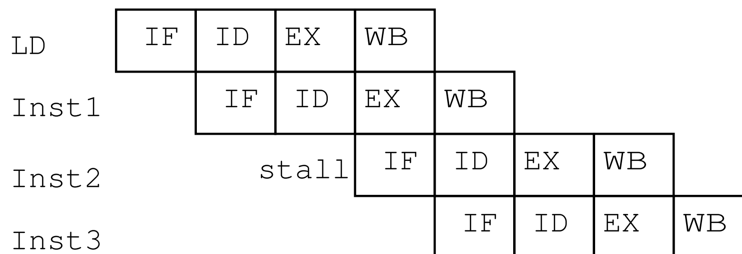


図 9.8: 構造ハザード

パイプラインストールにより、CPI がどれだけ増えるか調べよう。泡が入った後の CPI は以下の式で表される。

$$\text{理想の場合の } CPI + \text{泡の入る確率} \times \text{泡の入る個数}$$

例　題9.1: メモリをアクセスする LD 命令、ST 命令の確率が合わせて 25%である場合、パイプラインの CPI はどのようになるか。

答　: 理想の CPI は 1 である。泡は 1 個分なので、1+0.25*1=1.25 となる。

　構造ハザードは、リソースの複製により対処可能である。メモリについては命令メモリとデータメモリを分離すれば構造ハザードは生じない。実際にはメモリを分離するには相当のコストが掛かるため、キャッシュを分離して対処する。これは次の章で触れる。

　さて、POCO では構造ハザードは生じないが、データハザードと制御ハザードは深刻な問題である。順にこれを検討していく。

9.3.1　データハザード

　実は先に例を示したプログラムは、不自然であったことに気づくかもしれない。以下のプログラムは 0 番地の内容と 1 番地の内容を加算してその結果を 1 番地に格納するプログラムである。

```
01000_000_00000000   // 0 :  LDI  r0,#0
00000_001_000_01001  // 1 :  LD  r1,(r0)
01000_000_00000001   // 2 :  LDI  r0,#1
00000_010_000_01001  // 3 :  LD  r2,(r0)
00000_001_010_00110  // 4 :  ADD r1,r2
00000_001_000_01000  // 5 :  ST r1,(r0)
00000_000_000_00000  // 6 :  NOP
```

　ところがこれを上記のパイプラインで動かすと、そもそも LD 命令がうまく動かず、全く答えが出ないことがわかる。なぜだろう？

　これは、最初の命令でセットする r0 の値（ここでは 0）をすぐ次のステージで利用するためである。r0 に 0 が格納されるのは、WB ステージである。しかし、LD 命令は、LDI 命令がまだ EX ステージに居る時に、レジスタファイルから値を読み出してしまう。すなわち、r0 が 0 になる前の古い値を使ってしまうことになる。この様子を図 9.9 に示す。

　このようにパイプラインの動作を妨げる障害のことをパイプラインハザードと呼ぶ。ここで問題のハザードは、先行した命令の計算結果をすぐ次の命令が使うことから起きる。すなわちデータの依存性が問題になることからデータハザードと呼ばれる。データハザードを解決するためには、命令間に距離を置けば良い。図 9.10 に示すように、依存関係がある二つの命令間に NOP(あるいは関係のない命令) を置いてやれば、正しく動く。前回のプログラムがうまく動いたのは実はこの関係が満足されていたためで、何だかプログラムが不自然だったのはこのためである。この場合の NOP は、パ

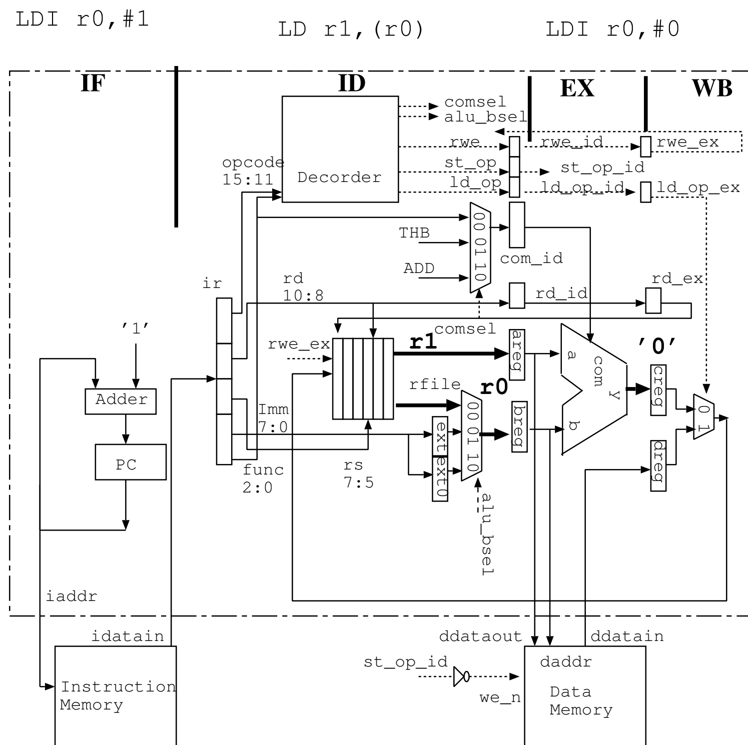


図 9.9: データハザード

イプラインの泡（バブル）と呼ばれる。バブルにより、パイプラインは停止（ストール）し、性能が低下する。

例　題 9.2: 前の命令の計算結果を次の命令が利用する可能性が 80%とする。この依存関係を NOP 命令を二つ入れて対処する場合、性能低下はどの程度になるか。

答　: 理想の CPI を 1 とすると、NOP による泡は 2 クロックになるため、1+0.8*2=2.6 となる。

これでは CPI は 2 を上回ってしまい、いくら周波数が高くてもパイプライン処理の利点が台無しである。

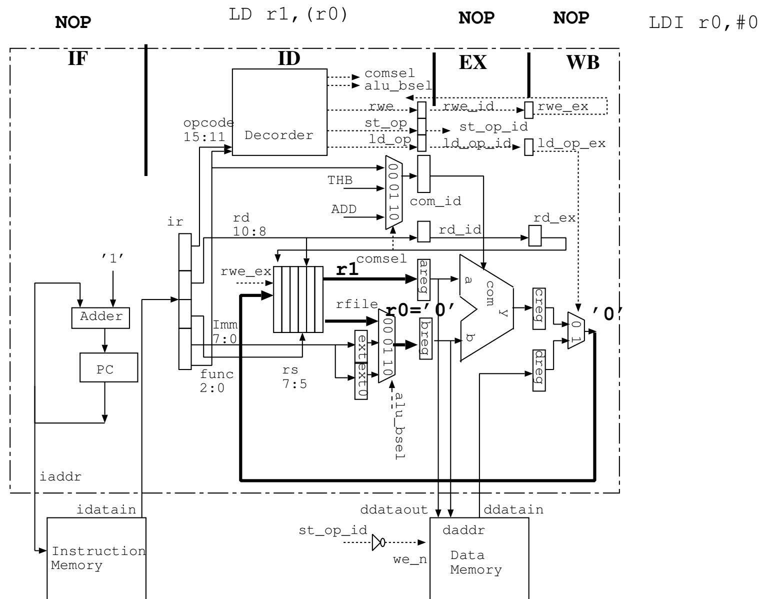


図 9.10: パイプラインのバブル

データハザードを解決すための手段はデータのフォワーディング (横流し) である。LD 命令で利用したい値 (0) は、一つ先の EX ステージで先行命令によって計算済みである。ただ、これがレジスタファイルに書き込まれるのが 2 クロック先になってしまうのが問題である。そうかといってレジスタファイルはすでに 3 ポート構成であり、これ以上様々なステージからデータを書き込むために入力を付けることはできない。そこで、計算したての値をそのまま使えるように横流ししてしまおう、というのがフォワーディングのアイディアである。まずは、レジスタファイルにフォワーディングを設ける。すなわち、書き込みイネーブル信号が出されていて、書き込みレジス

タ番号と読み出しレジスタ番号が一致したら、レジスタ書き込む値をそのまま出力してしまうようにする。

これで、図9.11に示すように書き込みを行う先行命令がWBステージに存在すれば、その書き込みデータはIDステージで読めるようになり、NOPは一つに減る。

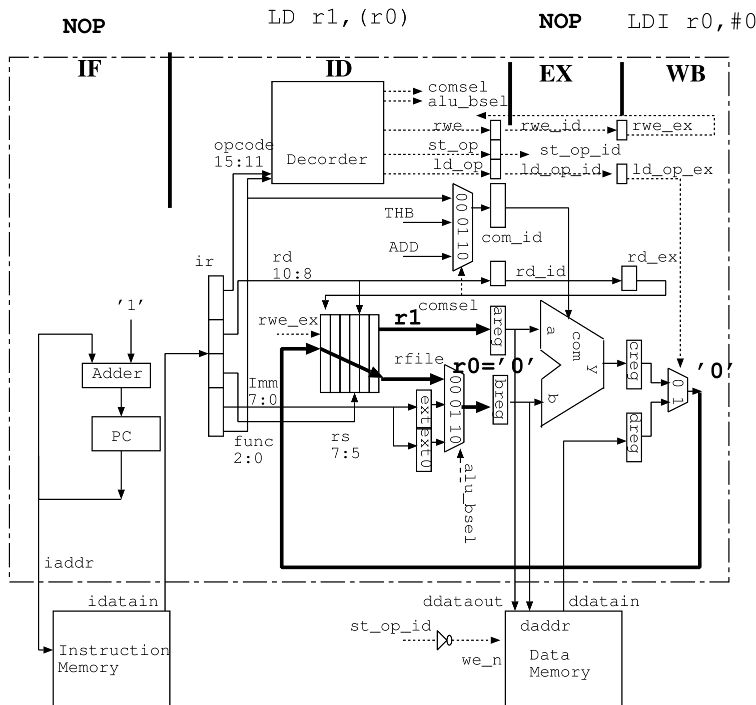


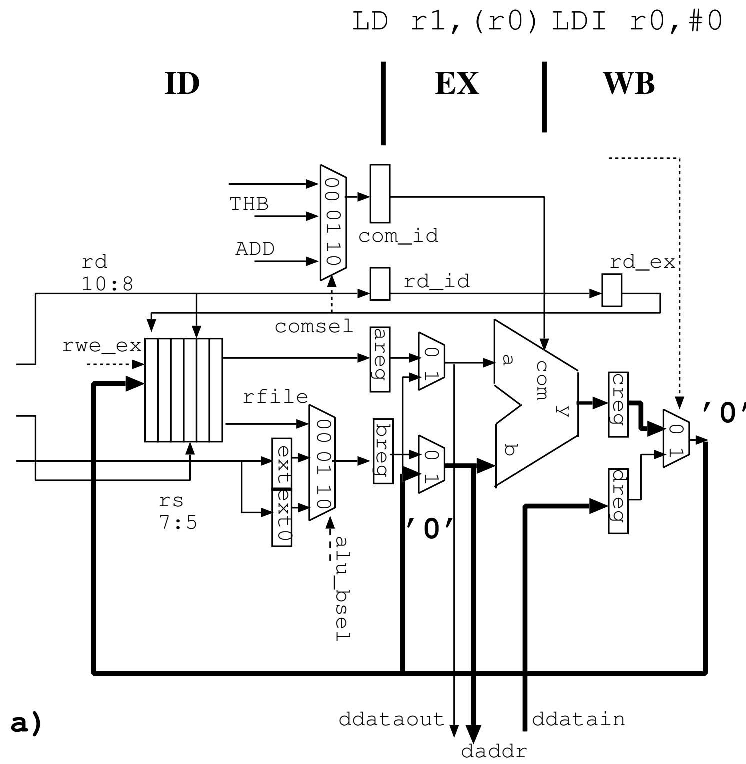
図9.11: レジスタファイルのフォワーディング

次に同じような方法をEXステージの命令についても適用しよう。図9.12に示す2つの方法が考えられる。最初の方法(図a))では、計算済み、あるいはデータを読んできた後にレジスタに入れた値をWBステージからEXステージに対してそのままALUの入力やLD命令のアドレスとして使えるように横流しする。

これに対して、二番目の方法(図b))では、ALUの出力、データメモリの出力をEXステージからIDステージに対して横流しし、レジスタファイルの値と入れ替えてareg, bregに格納する。

二番目の方法は、遅延パスが長くなるが、後で分岐命令を付け加える際に楽なので、今回は、こちらの方法を採用する。この方法では、IDステージを以下のように改造する。すなわち、

- EXステージの命令のrdフィールドとIDステージのrdフィールドが同じで、



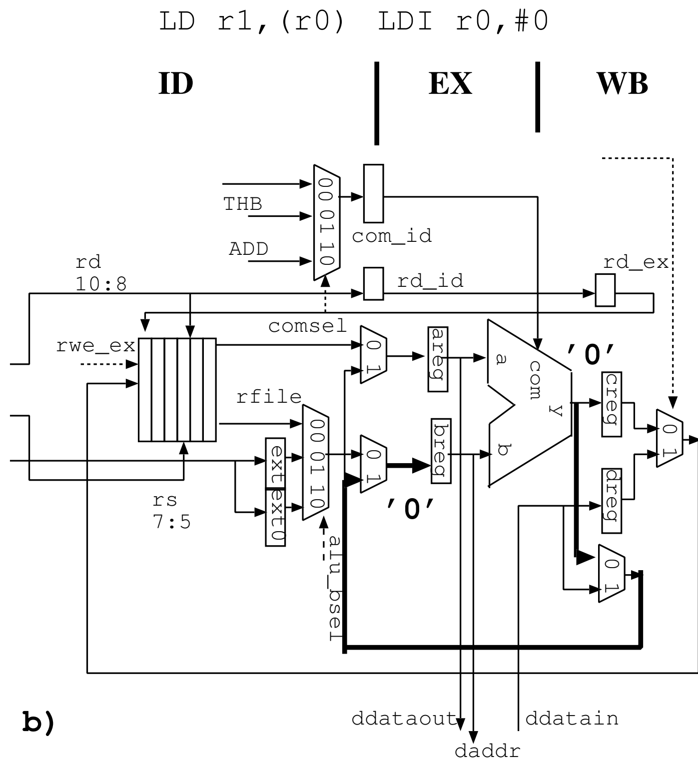


図 9.12: フォワーディング

EX ステージの命令がレジスタに結果を書き込む命令（つまり ST 命令や分岐命令でない）である場合、EX ステージの結果を横流しして、areg に格納する。

- EX ステージの命令の rd フィールドと ID ステージの rs フィールドが同じで、EX ステージの命令がレジスタに結果を書き込む命令（つまり ST 命令や分岐命令でない）である場合、EX ステージの結果を横流しして、breg に格納する。

EX ステージの結果は LD 命令かどうかによって持ってくる場所が違うのでこれを判別して切り替える必要がある。パイプラインの全体構成を図 9.13 に示す。areg と breq の前のマルチプレクサは、図中には書けなかったが、それぞれ以下の条件で切り替える。

- areg: rd_id=rd, rwe_id=1
- breg: rd_id=rs, rwe_id=1

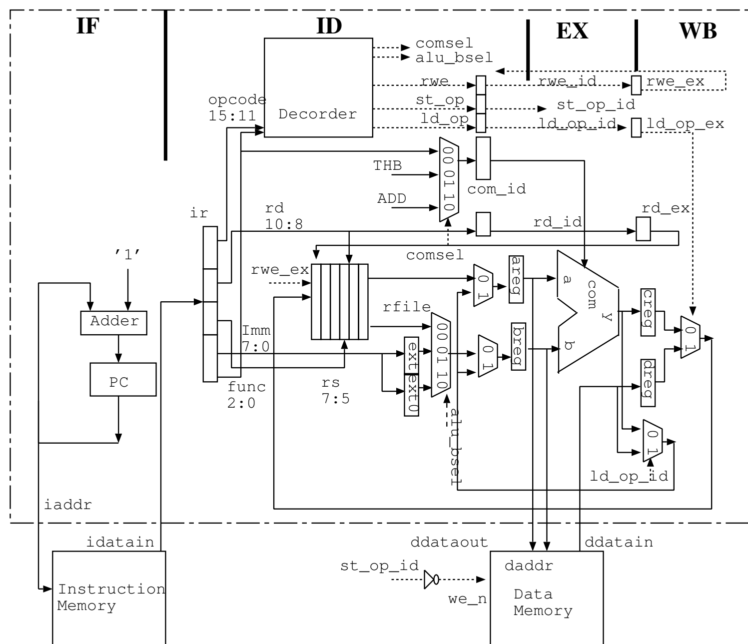


図 9.13: パイプラインの全体構成

各種データハザード

実は、データハザードにはここで扱ったものを含めて以下の三種類がある。

- RAW(Read After Write) ハザード: データを書く前に、値が読み出されてしまう問題。ここで扱った問題。
- WAR(Write After Read) ハザード: 値を読み出す前にそれが書き潰されてしまう問題
- WAW(Write After Write) ハザード：値を書き込む前に別の値を書き込んでしまうタイプの書き潰し。一般的には、メモリやレジスタに対しては生じない（書いた値を読む前に次の値を書くことはないはずなので、先に WAR 問題が起きるはずである。このような問題が起きるとすればプログラムのバグだろう）が、I/O に対するデータ転送などで問題になる。

現在の POCO では、レジスタファイルへの書き込みは最終ステージである wb ステージで行われるため、WAR ハザードと WAW ハザードは生じない。問題となるのは、RAW ハザードのみであるもちろん、RAR(Read After Read) はハザードを生じない。

9.3.2 コントロールハザード

今までの POCO のパイプラインには分岐命令がなかった。これを付け加えてみよう。以前同様、分岐先のアドレスを計算させるには PC に 1 を足す時に使った加算器を使う。また、レジスタの値が 0 かどうかを比較する比較器を ID ステージに取り付ける。ここで、レジスタの値は、EX ステージからフォワーディングされて来たものを使う点に注意しよう。こうしないと直前の命令でレジスタの値が変化した場合、比較に使えないことになる。分岐命令を取り付けたパイプラインを図 9.14 に示す。

ここでは、左に書いた命令メモリのアドレスに以下の命令が格納されているとする。

```
3  ADD r4,r2
4  ADDI r3,#-1
5  BNZ r3, -3
6  ADDI r0,#1
```

ここで、ADDI 命令は EX ステージに居り、BNZ 命令が ID ステージに居るケースを考える。ここでは、図中に示すように直前の命令で計算された結果 (r3-1) も、フォワーディング機構を使って分岐命令での判定に用いることができるようになっている。ここで、判定の結果 r3 が 0 ではなく、分岐が成立する場合を考えよう。pcsel=1 となり、6-3 が計算されて飛び先アドレスの 3 が PC から現れる。しかし、図 9.15 に示すように、これが PC に格納されるのは、次のクロックである。すなわち、6 番地の命令は分岐が成立した場合でも、パイプラインに入って来てしまうのだ。この命令は本来分岐が成立した場合は実行してはならないので、これは問題である。このような問

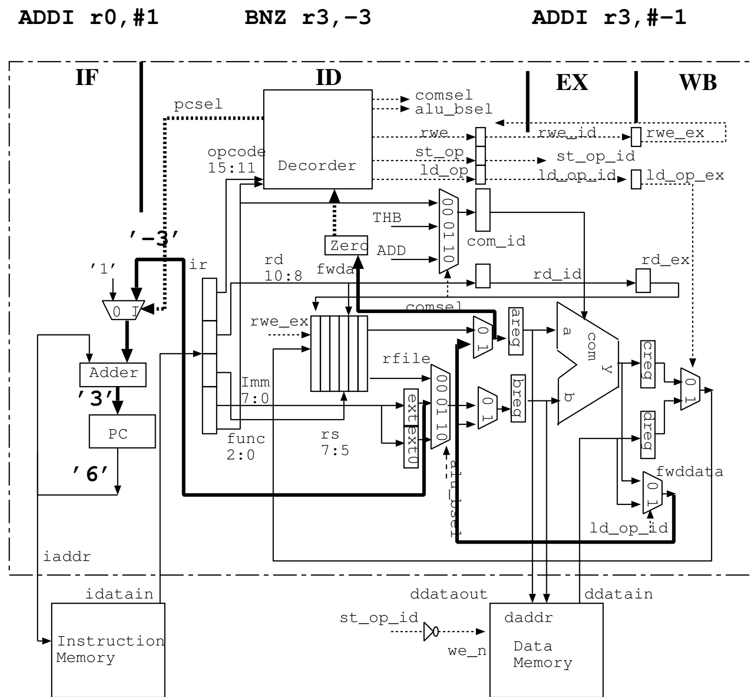


図 9.14: 分岐命令付きパイプライン

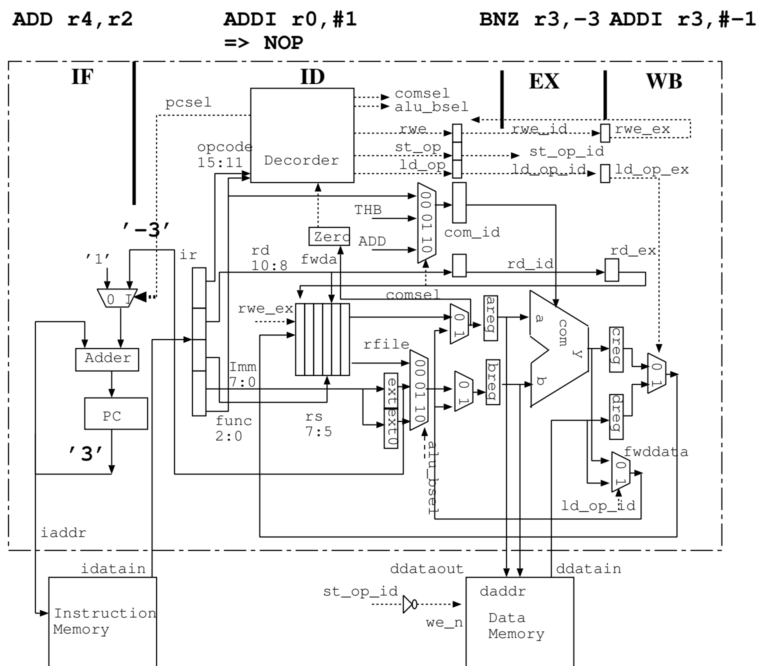


図 9.15: 制御ハザード

題は前回紹介したハザードの一種であり、分岐すなわちプログラムの制御によって生じることから制御ハザードと呼ぶ。

制御ハザードを解決するもっとも簡単な方法は、ID ステージで分岐命令と分かったら、強制的に IF ステージで命令をフェッチするのを止めてしまうことである。ハザードを防ぐためにパイプラインを止めることをパイプラインストールと呼ぶ。IF ステージの場合、止めるのが面倒な場合は、NOP 命令と入れ替えてしまう方法も考えられ、POCO では NOP は機械語の全 bit を 0 にすれば良いので楽である。しかし、いずれの場合でも、パイプラインには 1 クロック分泡が入って、パイプラインの性能が低下する。

例　題 9.3 分岐命令の後の命令を全て NOP に置き換えるとすると、これにより CPI はどの程度増加するか？ただし、分岐命令の存在確率は 25%となる。

1+0.25*1=1.25 となる。

ここで、少し頭を使おう。分岐命令が成立しない場合は、フェッチした命令はそのまま使えるので、NOP 命令に変更する必要はない。したがって、分岐命令が成立する場合だけフェッチした命令を NOP に変更する。このためのハードウェアは成立するかどうか示す信号は作ってあるので大変簡単である。

この方法は、分岐が成立しないと予測して命令をフェッチしてしまうとも考えられるので、もっとも簡単な分岐予測とされており predict-not-taken(飛ばない方に賭ける) と呼ばれる。残念なことに、多くの分岐命令では、飛ぶ確率は飛ばない確率よりも高い。これはループを形成するには前方で飛ぶ必要があるからだ。ここで、飛ぶ確率を 70%と仮定すると、predict-not-taken を使った場合の CPI は

$$1 + 0.25 \times 0.7) \times 1 = \quad 1.175$$

と若干改善できる。

9.3.3　遅延分岐

もう一つ、 もっと簡単な方法がある。この方法はハードウェアに全く手を加えず、考え方のみ変える。「この CPU の分岐命令は効き目が遅く、飛ぶのが一命令分遅れる」と考えるのである。そして、遅れた間の一命令分はパイプラインに入って普通に実行されると考える。この方法を遅延分岐 (delayed branch) と呼び、効き目がでるまでの間のことを遅延スロットと呼ぶ。ここで遅延スロットには分岐が成立しようがしまいが実行される命令を埋めておけば性能は低下しない。例えば、以前紹介した乗算のプログラムを考える。

```
        LDI  r0,#0
        LD  r3,(r0)
        ADDI r0,#1
        LD  r2,(r0)
        LDI  r4,#0
loop:   ADD r4,r2
        ADDI r3,#-1
        BNZ r3,loop
        遅延スロット
```

ここで使われている BNZ が遅延分岐である場合、遅延スロット内の命令はループ内に入るので、ループを一回分回るのに 4 命令を要することになる。ここで ADD 命令を遅延スロットに移動してみよう。

```
        LDI  r0,#0
        LD  r3,(r0)
        ADDI r0,#1
        LD  r2,(r0)
        LDI  r4,#0
loop:   ADDI r3,#-1
        BNZ r3,loop
        ADD r4,r2
```

このプログラムは見るからに変だが、ADD 命令はちゃんとループ内で実行し、正しい結果が得られることがわかる。しかもループを 1 回まわるのは 3 命令で済む。実際に Verilog シミュレーションでこのことを確かめよう。

遅延スロットは、上記のように、分岐が成立しても成立しなくても実行することが必要な命令をもってくるのが良い。しかし、分岐が成立した時だけ有効で、成立しない場合に実行しても害がないもの、逆に分岐が成立しない時だけ有効で、成立した場合に実行しても害がないもので埋めても良い。このように性能を上げるために、命令を移動することを命令スケジューリングあるいはこの場合はパイプラインスケジューリングと呼ぶ。どうしても埋まらない場合は NOP 命令を入れておく。

命令スケジューリングは多くの場合、コンパイラで行う (C のコンパイラのオプションを変えるとスケジュールを実行してくれる)。ここで、遅延スロットがどうしても埋まらない可能性を 20%とすると、CPI は以下のようになる。

$$1 + 0.25 \times 0.2) \times 1 = \quad 1.05$$

この場合、理想の CPI である 1 にかなり近づいたといえる。

演習 9.2

1 から n まで加算するプログラムに関して遅延分岐を想定してパイプラインスケジューリングを行え。

9.4　パイプライン処理の Verilog 記述

前回シミュレーションしたパイプライン化 POCO の Verilog 記述を検討しよう。本来、パイプライン構造は、それぞれのステージをモジュールとして宣言して設計するのがモジュラリティが高くてお勧めである。しかし、今回は、全体として記述量が少ないので単一モジュール内にベタで記述した。Verilog はどこで信号名を宣言しても良いので、各モジュールでローカルに用いる信号はそのモジュール内で極力定義している。ただし、前のステージに戻る信号線は、最初に出てきたモジュールで宣言しており、必ずしもそのモジュールで主として利用する信号でない場合もある。前回の図と信号線名を揃えておいたので、図と記述を見比べて信号を把握して欲しい。

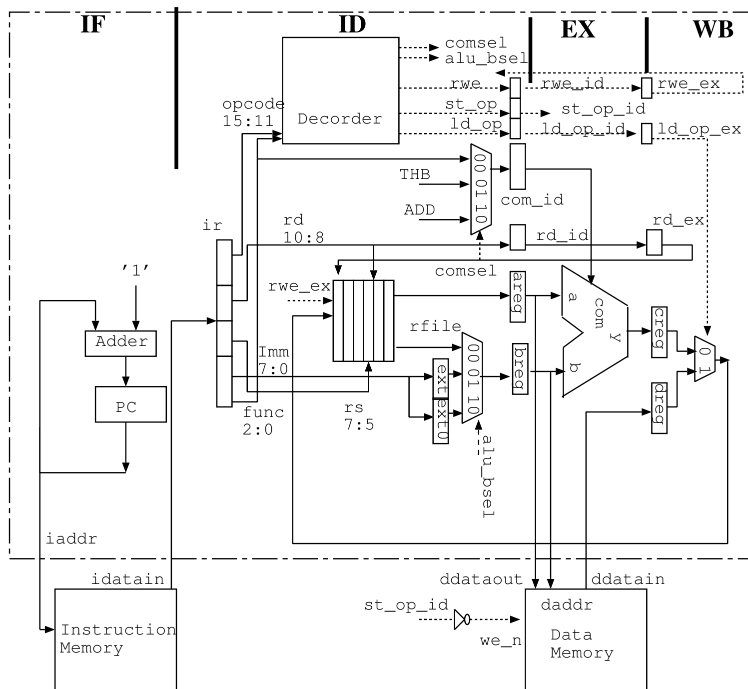


図 9.16: POCO のパイプライン構造

パイプライン化した poco のモジュール名を pocop とした。入出力は今までと同じなので省略した。

9.4.1 IF ステージ

まずは命令フェッチ (IF) のステージである。

```
// Instruction Fetch //
reg [‘DATA_W-1:0] pc;
reg [‘DATA_W-1:0] ir;

assign iaddr = pc;
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else
     pc <= pc+1;
end

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) ir <= 0;
   else ir <= idatain;
end
```

これは、毎クロック、pc の指し示す命令メモリから命令をフェッチして ir に格納する。今の所、分岐命令は付けてないので、大変簡単である。

9.4.2 ID ステージ

次は、命令デコード (ID) のステージである。このステージでは制御信号を生成すると共に、ir 中のレジスタ番号に従ってレジスタファイルからレジスタを読み出す。

```
// Instruction Decode & Register Fetch //
```

```
wire st_op, addi_op, ld_op, alu_op;
wire ldi_op, ldiu_op, ldhi_op, addiu_op;
wire [`SEL_W-1:0] com;
wire [`OPCODE_W-1:0] opcode;
wire [`OPCODE_W-1:0] func;
wire [`REG_W-1:0] rs, rd;
wire [`IMM_W-1:0] imm;
wire [`DATA_W-1:0] rf_a, rf_b, rf_c, alu_b;
reg [`SEL_W-1:0] com_id;
reg st_op_id, ld_op_id;
wire rwe;
reg rwe_id;
reg [`REG_W-1:0] rd_id;
reg [`DATA_W-1:0] areg, breg;
assign {opcode, rd, rs, func} = ir;
assign imm = ir[`IMM_W-1:0];
assign st_op = (opcode == `OP_REG) & (func == `F_ST);
assign ld_op = (opcode == `OP_REG) & (func == `F_LD);
assign alu_op = (opcode == `OP_REG) & (func[4:3] == 2'b00);
assign ldi_op = (opcode == `OP_LDI);
assign ldiu_op = (opcode == `OP_LDIU);
assign addi_op = (opcode == `OP_ADDI);
assign addiu_op = (opcode == `OP_ADDIU);
assign ldhi_op = (opcode == `OP_LDHI);

assign alu_b = (addi_op | ldi_op) ? {{8{imm[7]}},imm} :
                (addiu_op | ldiu_op) ? {8'b0,imm} :
                (ldhi_op) ? {imm, 8'b0} : rf_b;

assign com = (addi_op | addiu_op ) ? `ALU_ADD:
                (ldi_op | ldiu_op | ldhi_op) ? `ALU_THB: func[`SEL_W-1:0];

assign rwe = ld_op  | alu_op | ldi_op | ldiu_op | addi_op |
   addiu_op | ldhi_op ;

rfile rfile_1(.clk(clk), .a(rf_a), .aadr(rd), .b(rf_b), .badr(rs),
        .c(rf_c), .cadr(rd_ex), .we(rwe_ex ));
```

```
always @(posedge clk ) begin
  st_op_id <=  st_op; ld_op_id <=  ld_op; rwe_id <= rwe;
  areg <= rf_a; breg <= alu_b;
  com_id <= com; rd_id <= rd; end
```

レジスタファイルから読み出した値は A ポートの方はそのまま rega に格納して EX ステージに送る。B ポートの方は、イミーディエイト命令では利用しないので、命令に応じて ALU の B 入力 (alu_b) を生成してやる。さらに ALU の com も生成する。この生成の仕方、命令デコードの仕方は、パイプライン化されていない判と同じである。前回解説したように、生成した信号のうち、実行 (EX) ステージで利用するものについてはレジスタに格納して送ってやる。末尾に_id を付けた信号線がこれに当たる。レジスタファイルの書き込み部である rf_c, ed_ed, rwe_ex は前回述べたように WB ステージからの信号線である。

9.4.3　EX ステージ

お膳立ては ID ステージで終わっているので、EX ステージは、ALU で演算を行うと共に、データメモリをアクセスする。計算結果は creg に、メモリの読み出したデータは dreg に入れて次の WB ステージに送る。LD 命令かどうか、レジスタファイルに書き込むかどうか (rwe)、書き込む際の番号 (rd) は再びレジスタに格納して WB ステージに送る。

```
// Execution

wire [`DATA_W-1:0] alu_y;
reg [`DATA_W-1:0] creg, dreg;
reg ld_op_ex ;
reg [`REG_W-1:0] rd_ex;
reg rwe_ex;

alu alu_1(.a(areg), .b(breg), .s(com_id), .y(alu_y));

assign ddataout = areg;
assign daddr = breg;
assign we = st_op_id;
```

```
always @(posedge clk ) begin
  creg <= alu_y;
  dreg <= ddatain;
  ld_op_ex <= ld_op_id;
  rd_ex <= rd_id;
  rwe_ex <= rwe_id; end
```

9.4.4 WB ステージ

たった一文しかないが、これは、レジスタファイルに書き込むデータとして LD 命令ならば、dreg をそうでなければ creg を選ぶための記述である。

```
// Write back
assign rf_c = ld_op_ex  ? dreg : creg;
```

後の記述は、実は ID ステージ内にあり、EX ステージでレジスタに格納された信号線に従って、この rf_c がレジスタファイルに書き込まれる（ST 命令などでは書き込まない)。

```
// Instruction Decode and Register Fetch
...
rfile rfile_1(.clk(clk), .a(rf_a), .aadr(rd), .b(rf_b), .badr(rs),
        .c(rf_c), .cadr(rd_ex), .we(rwe_ex ));
```

図に書くより Verilog 記述を示した方が早いだろう。rfile の記述を以下のように変更すれば良い。

```
        assign a =
                (aadr == cadr)&we ? c:
                aadr == 0 ? r0:
                aadr == 1 ? r1:
                aadr == 2 ? r2:
                aadr == 3 ? r3:
                aadr == 4 ? r4:
                aadr == 5 ? r5:
```

```
                aadr == 6 ? r6: r7;
        assign b =
                (badr == cadr)&we ? c:
                badr == 0 ? r0:
                badr == 1 ? r1:
                badr == 2 ? r2:
                badr == 3 ? r3:
                badr == 4 ? r4:
                badr == 5 ? r5:
                badr == 6 ? r6: r7;
```

これに対応する Verilog 記述は、ID ステージの記述を下記のように書き換える。fwdata が図中には EX ステージにあるマルチプレクサであり、（EX の所に書いてもよかったが分かりやすさを重視してこちらに書いた）フォワーディングすべきデータが ALU からか、メモリからかを選択する。

fwda は、areg の前に設けた rd に対するフォワーディングを行うためのマルチプレクサであり、先行命令の rd_id と現在 ID ステージに居る命令の rd を比較して等しく、かつ先行命令が書き込みを行えば (rwe_id) フォワーディングを行う。b 入力 (alu_b) の方は、図とは異なり、マルチプレクサを一体化して記述している。b 入力に対するフォワーディングは、R 型の命令だけに有効にする必要がある。マルチプレクサを一体化することで、イミーディエイト命令等ファワーディングをしてはならない命令を最初に選び出してやることで、設計を簡単化している。

```
assign fwddata = (ld_op_id) ? ddatain: alu_y;

assign alu_b = ((rd_id == rs)& rwe_id) ? fwddata :
                (addi_op | ldi_op) ? {{8{imm[7]}},imm} :
                (addiu_op | ldiu_op) ? {8'b0,imm} :
                (ldhi_op) ? {imm, 8'b0} :  rf_b;

assign fwda = ((rd_id == rd)& rwe_id) ? fwddata: rf_a;
...

always @(posedge clk or negedge rst_n ) begin
  if(!rst_n)
        rwe_id <= `DISABLE;
  else begin
       st_op_id <=  st_op; ld_op_id <=  ld_op; rwe_id <= rwe;
       com_id <= com; rd_id <= rd;
```

```
        areg <= fwda; breg <= alu_b;
    end
end
```

他に初期化と NOP 命令の識別を行ったコードを web 上に示す。これで先に示したプログラムが NOP 無しに実行できることを確かめよう。

9.4.5　遅延分岐の Verilog 記述

遅延分岐を使う場合、パイプラインに細工をする必要がないため、分岐命令を付け加えるのは楽である。まず IF ステージ内の pc の設定を以下のように書き直す。

```
always @(posedge clk or negedge rst_n)
begin
    if(!rst_n) pc <= 0;
    else if(pcsel)
      pc <= pc +{{8{imm[7]}},imm} ;
    else
      pc <= pc+1;
end
```

次に ID ステージに分岐命令のデコードと、pcsel の制御を付け加える。

```
assign bez_op =  (opcode == `OP_BEZ);
assign bnz_op =  (opcode == `OP_BNZ);
...
assign pcsel = (bez_op & fwda == 16'b0 ) | (bnz_op & fwda != 16'b0) ;
```

ここで、fwda はフォワーディング後の areg に蓄えるデータである。前回に付けたマルチプレクサからのフォワーディングデータをそのまま利用している。

演習 9.3

JALR 命令をパイプライン化された POCO に実装せよ。

第10

記憶の階層とキャッシュ

10.1　記憶の階層

POCOは単純な構造を持っているので、最近の半導体プロセス(65nm)を用いて論理合成すると300MHz程度で動作可能となる。実際には配置配線の段階で配線遅延の影響で動作周波数が低下するが、商用のマイクロプロセッサでは組み込み用途であっても200MHz以上の周波数で動作するものが多い。これがデスクトップ用、ラップトップ用ともなると動作周波数は3GHzに達する。

一方、主記憶の容量は1MIPS当り1Mbyte必要であると言われており、現在の高性能プロセッサの主記憶は数百Mbyteの容量を必要とする。しかし、これに追従するようなアクセス速度と容量を持つメモリは存在しないか、あるいはひどく高価なものとなる。そこで、高速／小容量のメモリと低速／大容量のメモリを組み合わせ、記憶の階層を構成する。図10.1に記憶の階層の構成例を示す。

コンピュータのメモリシステムは、CPUの近くから、順に高速なメモリを階層的に接続し、高速なメモリ上にデータ（命令）が存在しなければ、階層が上の、低速だがより大容量なメモリにデータを取りにいく。よく使うデータ（命令）がうまく高速小容量なメモリ内に入れば、低速なメモリに取りにいく確率を小さくすることができるので、仮想的に、みかけは大容量で高速なメモリシステムを実現することができる。これが記憶の階層である。

現在の多くのコンピュータは、以下の階層構造を持っている。

- L1キャッシュ　オンチップSRAM　4K-32K 1クロックでアクセス可能
- L2キャッシュ　オンチップSRAMまたはオンボードSRAM　64K-1M 10クロック程度
- (L3キャッシュ　オンボードSRAM　256K-4M 50-100クロック程度)
- メインメモリ　DRAM 64M-4G 100クロック程度

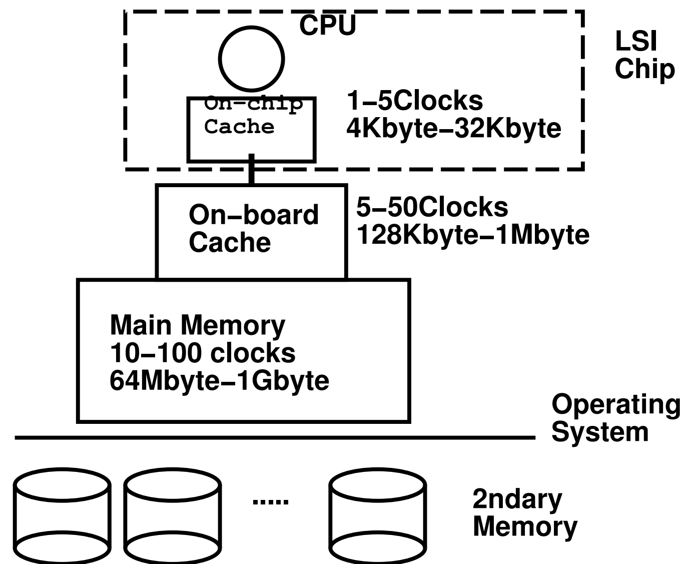


図 10.1: 記憶の階層

- 補助記憶　ディスク　 4G-1T 10msec 程度

キャッシュは、記憶階層の最も CPU 側に位置し、特に頻繁にアクセスする命令やデータを格納しておく高速小容量のメモリである。最近のプロセッサでは、図 10.1 に示すように、チップ内に L1 キャッシュ(オンチップキャッシュ)、チップ外に L2 キャッシュ(オンボードキャッシュ) を持ちキャッシュ自体が、階層構成になっている場合が多い。

1 次キャッシュはチップ内に置くため、容量は制限され、4K-32KByte であるが、命令／データを分離した構成を取ることができる。小容量のオンチップキャッシュは 1 クロックでアクセスできるが、容量が大きいものだとチップ内でも数クロックから 10 クロック程度時間がかかる場合もある。一方、2 次キャッシュは、チップ外に置くため、64K-1MByte の容量の大きなメモリを用いることができるが、アクセス時間は大きく、場合によっては数 10 クロックになる。キャッシュは、高速で使い易い SRAM(Static RAM) 素子を用いる。SRAM 素子は簡単な構造を持つ Flip Flop の 2 次元配列状の記憶領域を持ち、10nsec から 80nsec 程度の時間で読み書き可能である。

これに対して主記憶（メインメモリ）は、チップ当たりの記憶容量の大きい DRAM(Dynamic RAM) 素子を用いる。DRAM 素子は、チップ内にコンデンサを持ち、ここに電荷が蓄えているかどうかで情報を記憶するため、1 チップ当たり大きな情報量を記憶することができる。最近の DRAM 素子は 1 チップ当たり 64Mbit の記憶が可能である。しかし、電荷は情報をアクセスしたり、時間がたつと放電してしまうため、読み出し用のプリチャージや、再充電のためのリフレッシュ操作が必要が必要で、使い難い。最

近の DRAM 素子は同期型 DRAM や Rambus 型 DRAM が用いられるが、これらの素子は、最初のデータをアクセスするまでには時間がかかるが、連続読み書きを高速に行うことができる。

通常、キャッシュは CPU から見て透過 (Transparent) である。すなわち、CPU はキャッシュ中にアクセスしたデータが存在しない場合に、プログラムで何かしなければならないわけではなく、キャッシュ制御用ハードウェアが自動的にアクセスしたデータが主記憶からキャッシュ中に転送されるまで CPU の命令実行は待ち状態になる。したがって、CPU はキャッシュの存在を意識しないで動くことができる。このうち、メインメモリまでは、ハードウェアで管理され、ソフトウェアからは透過（トランスペアレント）である。つまり、プログラムにとっては、メモリのどの階層から取ってくるかは気にしなくてよい。

一方、補助記憶は、OS(Operating System) によって制御される。ディスクに代表される補助記憶の利用法は二つある。

- ファイルシステム
- 仮想記憶のスワップ領域

ファイルは、ユーザに対しても見える形で、管理するメモリ領域であり、純粋に OS の範囲である。一方、仮想記憶 (Virtual Memory) は、メインメモリに載らないような大きなプログラムや、多数のジョブを動作させるための機構であり、ハードウェアは論理アドレスと物理アドレスの変換およびメインメモリに載っているかどうかの判定のみを行う。

このような記憶の階層がうまく働くためには以下の２つの性質がなければならない。

- 時間的局所性: 一度アクセスした場所は、近いうちに再びアクセスされる可能性が高い。
- 空間的局所性: 一度アクセスされた番地に近い番地が再びアクセスされる可能性が高い。

命令フェッチにおけるアクセスは、この両者の局所性を強烈に持っている。すなわち、多くのプログラムでは、多くの時間帯で、比較的小さなループを回っているのだ。一般に、プログラムの実行時間の 90%は、コードの 10%が占めているという 10%-90%の法則があるが、これは局所性を表したものといえる。データに対するアクセスは命令程強烈ではないが、巨大な行列を扱う科学技術演算以外は、かなり存在する。

10.2　キャッシュの基本

10.2.1　キャッシュのマッピング

主記憶のアドレスが CPU から出力された時に、それがキャッシュのどこに対応するかを決めてやる必要がある。この番地の対応付けのことをマッピングと呼ぶ。プロセッサからのアクセスが発生した時に、その番地からアクセスされたデータがキャッシュ上に存在するかどうかを判別し、存在する（ヒットする）場合はキャッシュからアクセスし、存在しない（ミスヒットまたはミスする）場合は主記憶からアクセスし、キャッシュの内容を更新する等を行なう必要がある。この判断は高速に行なわなければならないので、テーブルを順に検索したりするわけにはいかない。

さて、キャッシュは、管理上のオーバヘッドを少なくするため、一定サイズの、連続した番地のデータを一括管理する。これをキャッシュブロックまたはラインと呼ぶ。ちなみに 8 章で導入したバイトアドレスで説明すべきだが、簡単のため、16bit アドレスで説明する。ここで、POCO は 16 ビットマシンなので、ここでは 16 ビットのことをワードと呼ぶ。(世の中は 32 ビットマシンが多いので 32 ビットをワードと呼ぶことも多いので注意されたい。)

キャッシュのブロックサイズは 8 から 64 ワード程度が多い。ここでは 1 ブロックが 8 ワードと考えよう。キャッシュも主記憶もブロック単位で区切られている。つまり、主記憶のアドレスの中で、下位 3bit は、ブロック内に 8 つあるワードのどの位置を表すかを示すのに使われる。この部分をブロック内アドレスと呼ぶ。

ここで、非常に小規模の主記憶 (1k $\times$ 16 ビット)、キャッシュ(64 $\times$ 16 ビット) 間の割りつけを考えてみる。主記憶は 1k(2^{10}) ワードすなわちアドレス 10bit 分、キャッシュは全体として 64(2^6) $\times$ 16 ビットとする。ブロックサイズは 8 なので、キャッシュは 64/8=8 ブロック分を保持することになり、主記憶中には 1k/8=128 ブロックを保持することになる。この例はあくまで説明のためのもので、前節に述べた現在の主記憶、キャッシュの容量を考えると、まったく小規模なものである点に注意されたい。しかし、原理は全く同じで実際の CPU ではビット数が長くなるだけである。

ダイレクトマップ方式

主記憶は順にブロック単位に分割されていると考える。これに図 10.2 に示すように 0(0000000) から 127(1111111) まで番号を降り、10bit のアドレスの上位 7bit で表すことにする。下位 3bit はブロック内アドレスである。一方、キャッシュ内のブロックは 8 個であり、同様に 0 から 7 まで番号を降る。そこで、まず主記憶の 0(0000000) から 7(0000111) までをキャッシュの 0 から 7 まで順に割りつける。次に 8(00001000) から 15(00001111) までを同様に 0 から 7 までに割りつけ、さらに 16(00010000) から 23(00010111) までを 0 から 7 までに割りつけ、以下最後に割りつけて、最後の 120(11111000) から 127(11111111) を 0 から 7 までに割りつける。この方式だと、ア

ドレスの上位 7bit のなかで、下の 3 ビットがキャッシュブロックのどの場所に格納されるかを示すことになる。この下位 3 ビットをインデックスと呼ぶ。インデックスで表される 8 つのキャッシュに対して、それぞれ主記憶上のどのブロックが入っているかを示すためには、残った 4 ビットを記憶すれば良い。これをタグあるいはキーと呼ぶことにする。

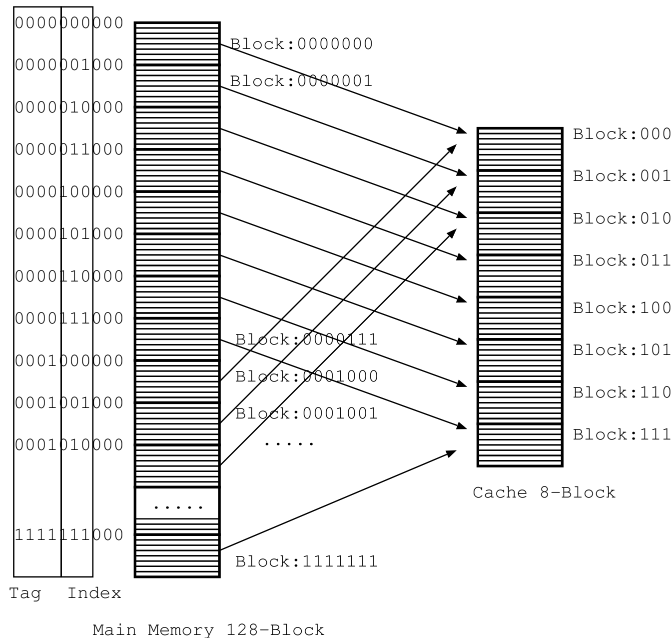


図 10.2: ダイレクトマップ方式

ダイレクトマップでは、図 10.3 に示すように、インデックスで表されるキャッシュのそれぞれのブロックに対して 4 ビットのタグメモリを設けて、そこにどの主記憶上のブロックが入っているかを記憶させておく。

CPU から読み出しアクセスが行われた時、キャッシュは以下のように働く。(1) タグメモリをインデックス (010) をアドレスとして参照し、(2) テーブルから得られたタグ (0000) と CPU からアクセスされたアドレスのタグ部を比較する。等しければキャッシュ中にアクセスされたブロックが存在 (ヒット) し、等しくなければ存在しない（ミスヒット）ことがわかる。(3) ヒットすれば、キャッシュからデータを読み出し、(3’) ミスヒットすれば、主記憶からキャッシュを転送して、インデックスで示す位置 (この場合 010) に格納して、タグメモリのインデックスで示される番地 (この場合 010) にアドレスのタグ部 (0000) を格納してから、アクセスを行う。

この方法では、キャッシュ中の格納されるブロックの総数に等しいエントリのタグメモリを持てば良い。インデックスを使ってタグメモリと同時にキャッシュをアクセ

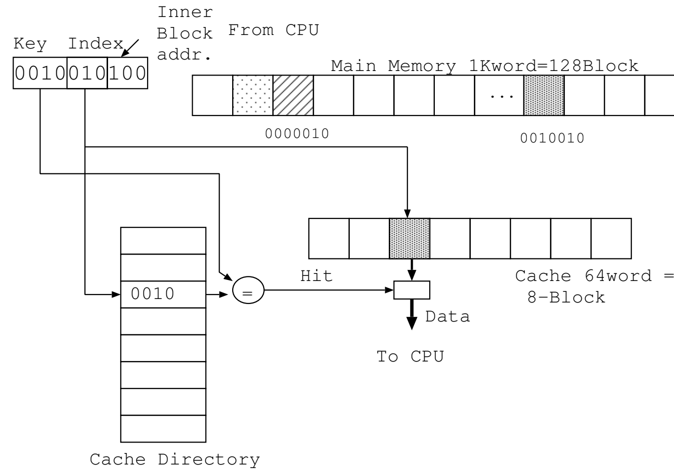


図 10.3: ダイレクトマップ方式のキャッシュ

スできる。タグメモリから読み出した値と比較してミスヒットした場合は、読み出した値を捨ててしまえば良い。

しかし、この方法では、キャッシュ中に格納される位置が一意に決まってしまうため、インデックスが等しい 2 つのブロックは、たとえキャッシュの他の位置が空いていたとしても、同時にキャッシュ中に格納することができなくなる。この例では、ブロック 0000010 が格納された状態では、ブロック 0010010 は、インデックス 010 が一致してしまうために、格納することができない。したがって、ブロック 0010010 をアクセスした場合は、ミスヒットとなり、主記憶からブロックを取ってきて入れ替える必要がある。このようにインデックスが重なったことによって起きるミスをインデックス衝突ミス (conflict miss) と呼ぶ。インデックス衝突ミスは、特にキャッシュサイズが小さく、キャッシュに格納可能なブロック数が少ない場合に発生し、性能を阻害する原因となる。

セットアソシアティブ方式

同じ容量のキャッシュでも、図 10.4 に示すようにインデックスの bit 数を 1bit 減らせば、同一のインデックスに対して格納可能なブロック位置を 2 箇所用意することができる。2 箇所のペアで同じ番号を割り当てているものをセットと呼ぶ。

この場合、キャッシュは図 10.5 に示すようにタグメモリを 2 組用意し（インデックスが 1bit 減るのでエントリ数は半分になる）、それぞれ対応するキャッシュに格納されたブロックのタグを格納する。このことにより、キャッシュ上半分のインデックス 10 にはブロック 00100 を、下半分のインデックス 10 にはブロック 00000 を格納することが可能となる。このように分割したキャッシュの 1 まとまりを way と呼ぶ。この

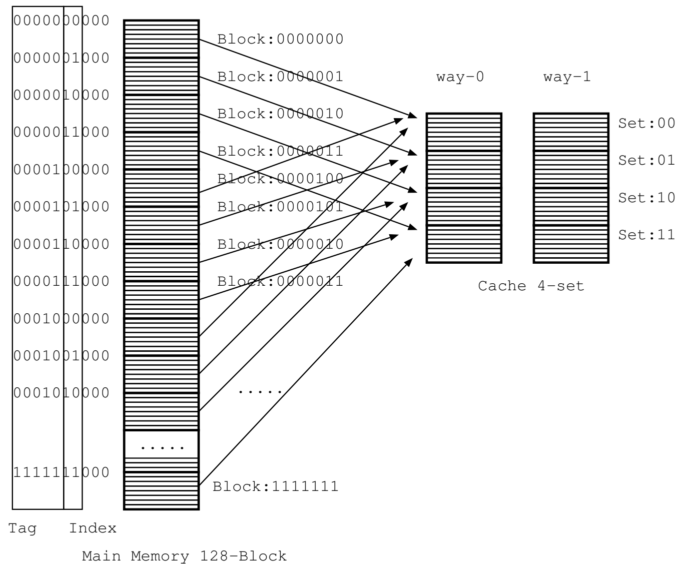


図 10.4: 2-way セットアソシアティブ方式

例では 2-way に分割したことになる。CPU がアクセスした場合、今度は同一のインデックスで同時に２つのタグメモリを参照し、得られたアドレスと、アドレスのタグ部を同時に比較する。そして、一致した way のキャッシュに対しデータの読み書きを行う。両方共一致しなかった場合は、ミスとなり、主記憶からどちらかの way にブロックを転送する必要が生じる。

この方法は、キャッシュを 2-way に分割することから 2-way セットアソシアティブ方式と呼ぶ。ダイレクトマップ方式に比べ、ブロックの格納可能な位置が倍になるためインデックス衝突ミスを減らすことができる。一方で、タグメモリと比較器が２組必要で、アクセスされたキャッシュからデータを CPU に送り出すパスにもマルチプレクサが必要になる。２つのタグメモリは同時に参照可能であるが、構造が複雑なため、全体としてダイレクトマップ方式に比べてアクセスのための遅延が大きくなる傾向にある。

ちなみに、2-way にしたからといってタグメモリが 2 倍必要なわけではない。エントリ数が半分なものが 2 組必要になるわけで、メモリ要求量の増加はタグの増加分 1bit 分で済む。この考え方を拡張し、キャッシュを n 個に分割して、タグメモリと比較器を n 組持たせれば、n-way セットアソシィアティブ方式を実現することができる。図 10.6 は 4-way セットアソシアティブキャッシュを示す。この例では、インデックスがさらに減り 1 ビットになっている。ちなみに、ダイレクトマップは、1-way セットアソシアティブキャッシュである。

ダイレクトマップは 16 人が 1 つだけの椅子を争う椅子取りゲームである。2-way

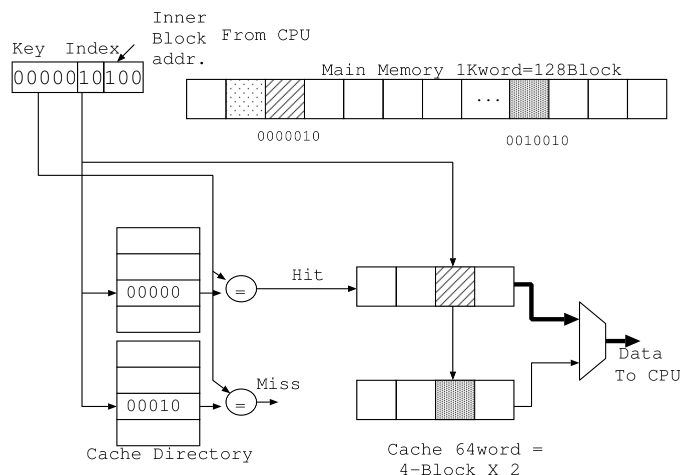


図 10.5: 2-way セットアソシアティブ方式のキャッシュ

は椅子が二つに増えるが、参加者の数も倍になって 32 人になってしまう。これでは確率が同じないのではないか、と思うかもしれないが、実はそうでもない。あなたが椅子が欲しいタイミングにちょうど椅子を要求するしつこいライバルが一人居るとしよう。椅子が一つしかなければ毎回熾烈な争いをしなければならないが、椅子が二つあれば多くの場合は平和共存できるのである。これがインデックス衝突ミスを減らす効果である。一般にダイレクトマップを 2-way にするとヒット率が、キャッシュの容量を倍にしたのと同じ位の上がると言われている。もちろん動作させるプログラム、キャッシュの容量にも依存する。一般にキャッシュの容量が小さいほど、way 数を増やす効果がある。n の数を増やすことによるインデックス衝突ミスを減らす効果は n が大きくなる程頭打ちになり、十分な大きさのキャッシュサイズに対しては n は 4 より大きくしてもあまり効果がないことが知られている。

フルアソシアティブ方式

2-way set associative 方式を同様にもう一回、つまり 4 つのブロックでセットを組めば 4-way set associative 方式にすることができる。この場合、インデックスは 1bit になり、キーが 6bit となる。それでは、さらに折り畳むとどうなるか？

ここでは、キャッシュ容量が非常に小さいため、図 10.7 に示すように、すべてのキャッシュ領域に対してブロックの格納が可能になる。このように、どこにでも格納が可能な方式をフルアソシアティブ方式と呼ぶ。この方式を実現するには、大量の比較器を用いるか、メモリ中のデータを並列に検索する連想メモリ (associative memory, content addressable memory) が必要になる。したがって、このフルアソシアティブ方式は、キャッシュ容量が非常に小さい場合にのみ現実的な方法で、キャッシュよりも、

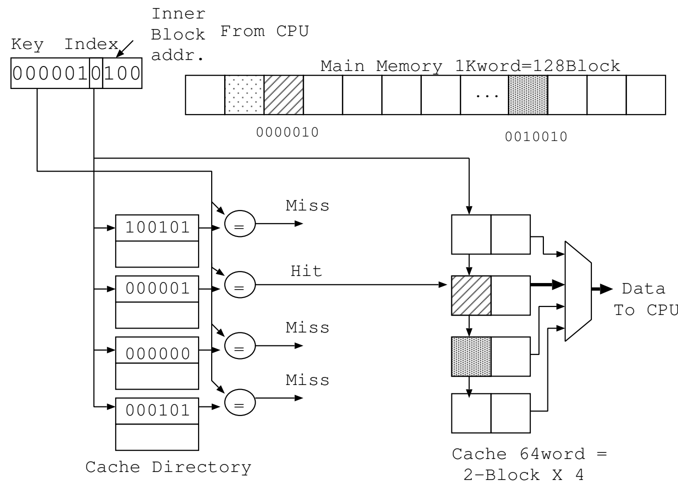


図 10.6: 4-way セットアソシアティブ方式

後に紹介する TLB(Translation Look aside Buffer) に用いられる。

演習 10.1 poco の全アドレス空間は 64Kwords であり、この空間すべてに対して 1Kwords のキャッシュを設ける。キャッシュブロックを 8words とする時、以下の方式でインデックス、タグのビット数はそれぞれどうなるか？

1. ダイレクトマップ方式
2. 2-way セットアソシアティブ方式
3. 4-way セットアソシアティブ方式

10.2.2 書き込みの制御

読み出しがヒットした場合は、単にキャッシュからデータを読み出して CPU に送れば良い。読み出しがミスすると、通常、主記憶からキャッシュブロックをキャッシュへ転送して、今までのブロック上に上書きする。この操作を置き換え（リプレイス）と呼ぶ。

問題は、キャッシュがミスした場合である。この時の制御がキャッシュの方式を分けることになる。

キャッシュの制御方式は次の二つの選択肢がある。

- ライトスルー（Write Through）あるいはストアスルー (Store Through): 書き込みデータをその都度主記憶に書き込む方法

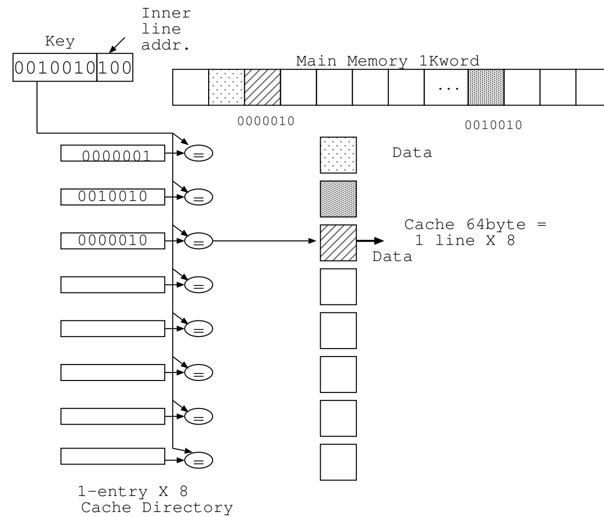


図 10.7: フルアソシアティブ方式

- ライトバック（Write Back）あるいはストアイン (Store In): 書き込みデータはキャッシュのみに格納し、その場では主記憶には書き込まない方法

ライトスルー（Write Through）方式:　ライトスルー方式は、図 10.9 に示すように、キャッシュに書いたデータが常に主記憶に反映されるので、キャッシュブロックが追い出される時に主記憶に対して書き戻してやる操作は不要だが、常に書き込みを行なうため、キャッシュ／主記憶間のバスが混雑する。

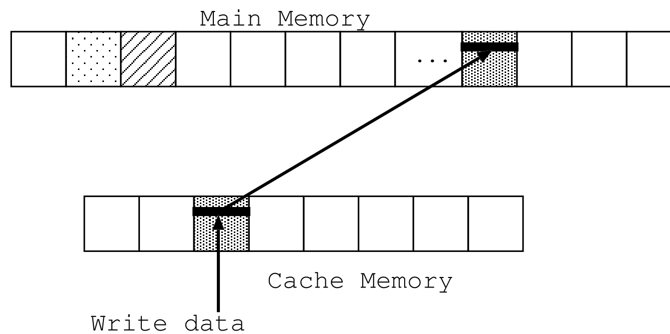


図 10.8: ライトスルー方式

CPU はデータの書き込みに関しては、終了を待たずに次の命令に進むことができるので、書き込みデータを主記憶に書き込む際に仮に格納してやる簡単なメモリ（ラ

イトバッファ）を設け、連続したアドレスをまとめて書き込むマージ機構をきちんと作ってやれば、CPU の実行時間に悪影響を与えることは防ぐことができる。ただし、連続したアドレスに多数のデータを書き込んで、ライトバッファが溢れる場合には、書き込みが行えなくなって、CPU が待ち状態になる場合がある。

ライトスルー方式は、書き込み時にミスした場合については、直接主記憶のみを更新する方式 (ダイレクトライト：Direct Write) と、そのブロックを一度キャッシュ上に取ってきてから書き込む方法（フェッチオンライト：Fetch on Write) がある。前者は実装が簡単であるが、次に同じ番地を Read する場合にミスヒットしてしまう。これに対し、後者はまずブロックを読み込んでリプレイスしてから書き込むので制御が複雑になるが、次の Read がヒットするためヒット率を高めることができる。

ライトバック（Write Back）方式:　ライトバック方式は、キャッシュに書き込んだデータをその都度主記憶に反映せず、キャッシュのみを更新し、そのブロックが追い出される際に主記憶に書き戻して一致を取る。このため、一定期間、主記憶上のデータとキャッシュの一致が取れなくなる。そこで、タグメモリ上にダーティ(Dirty) bit を設け、主記憶上のデータと一致しているかどうかを管理する。一致している状態を Clean、そうでない状態を Dirty と呼ぶ。

読み出しミスが起き、ブロックを主記憶から取り込んでリプレイスすると、最初はブロックの状態を Clean とする。次にそのブロックに書き込みを行うと、その書き込みは主記憶に反映せず、状態を Dirty に変更する。この先何度書いてもキャッシュのみにデータが書き込まれる。これが図 10.9 a) に当たる。次に、Dirty のブロックが、リプレイスの対象となって、キャッシュから追い出される場合、単純に上書きすると、今まで書き込んだデータが消失してしまう。そこで、図 10.9 b) に示すように、主記憶に書き戻し (Write Back) を行って一致を取ってから、ブロックを上書きする必要がある。Clean なブロックを追い出す場合には書き戻しの必要はなく、直接新しいブロックを上書きする。書き込みミスに際しては、常にフェッチオンライト方式を取る。すなわち、一度キャッシュ内にブロックを読み込んでから、書き込みを行う。

この方法は、ライトバッファを設けなくてもキャッシュが高速ならば性能が落ちることがなく、データの連続書き込みでライトバッファが溢れる心配も不要である。しかし、一方で、追い出しの際の書き戻しのロスに加え、主記憶とキャッシュの一致が取れないことから、I/O が主記憶から DMA(Direct Memory Access) 転送 (以前やったが覚えているか？) を行う場合などには、問題が生じることがある。この場合、I/O から書き戻し要求を出して強制的に書き戻しを行う必要がある。普通のプロセッサの場合は、きちんと作ればどちらの性能がいいとも言えないが、複数のプロセッサを用いるマルチプロセッサ方式の場合は、バスの混雑が性能に致命的な影響を与えるためライトバック方式が優れている。

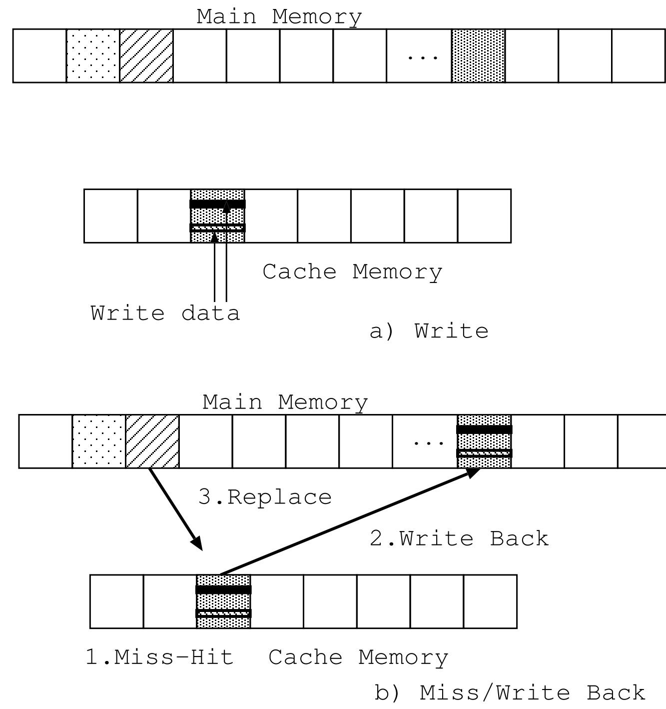


図 10.9: ライトバック方式

例題 10.2:　前回のダイレクトマップキャッシュを想定して、以下のアクセスが順に行われたとする。ダイレクトライトのライトスルーキャッシュ、フェッチオンライトのライトスルーキャッシュ、ライトバックキャッシュ、はそれぞれヒットするか？
(1) 0100001000 番地から読み出し　(2) 1100001001 番地に書き込み　(3) 1100001001 番地から読み出し

答:　(1) は最初なのですべてのキャッシュでミスする。これを初期化 (Conpulsory:必須) ミスと呼ぶ。(2) はインデックス部分の 001 が重なるために、やはりすべてのキャッシュがミスをする。これをインデックス衝突 (Conflict) ミスと呼ぶ。この時に、ダイレクトライトのライトスルーキャッシュは、主記憶にデータを直接送りつけて更新するためキャッシュの置き換え (リプレイス) を行わない。しかし、フェッチオンライトのダイレクトマップキャッシュとライトバックキャッシュはここで、リプレイスを行う。したがって、(3) で同じ番地を読み込む場合、フェッチオンライトとライトバックはヒットし、ダイレクトライトのみミスする。
この例は、フェッチオンライトとライトバックに有利な場合であるが、逆の場合も考えられる。これが以下の演習問題である。

演習 10.2　例題と同様に場合に以下のアクセスが順に行われたとする。それぞれのキャッシュはヒットするか？
(1) 0100001000 番地から読み出し　(2) 1100001001 番地に書き込み　(3) 0100001001 番地から読み出し

追い出しの制御

ダイレクトライトを採用したライトスルー方式における Write ミスを除き、ミスヒットの際は、どこかのキャッシュのブロックをリプレイス（置き換え）するために、領域を確保する必要がある。ダイレクトマップ方式は 1 つしか可能性がないので問題ないが、セットアソシアティブ方式ではどの way のブロックを追い出すかを決める必要がある。この時用いられるのが、LRU(Least Recently Used) 方式である。この方法は、最近アクセスされなかった方の way を選ぶ方式で、主記憶-ディスク間の管理にも用いられる。2-way を越えると、厳密にもっとも最近用いられなかった way を選ぶのは難しく、最近用いられなかった way のうちどれかを選ぶ擬似的な方法を採る場合も多い。ちなみに、先にキャッシュに格納した方の way を選ぶ FIFO(First In First Out) 方式は、時としてランダムに追い出す方式より性能が悪いことが知られている。

10.3　キャッシュの性能評価

メモリアクセスのロスは、今回の設計の場合、以下の式で示すように CPI を増加させる。ここで、CPI(ideal) は、全メモリがクロック内でアクセスできると考えた際の CPI(Clock cycles Per Instruction) のことである。

平均 $CPI = CPI(ideal)$+命令キャッシュのミス率×リードミスペナルティ+LD 命令の確率×データキャッシュのミス率×ミスペナル

読み出しミスと書き込みミスは、本来ミスペナルティが違う。書き込みミス時は、ライトスルー方式、ライトバック方式共に簡単なライトバッファを備えることで CPU は処理完了を待たずに先に進むことができる。したがって、簡単なモデルでは書き込みミスを無視する場合もある。しかし、書き込みミスが引き起こすリプレイス処理をキャッシュが行っている際に、先に進んだ CPU が次のデータを読み書きすると、ここで CPU を待たせなければならない事態が出てきて、この辺は複雑である。本当に正確に性能を評価する場合は命令レベルのシミュレータを用いなければならない。

演習 10.3　ミスペナルティを命令、データ共に 20 クロック、命令キャッシュのミス率を 2%、データキャッシュのミス率を 4%、LD 命令の生起確率を 0.2 として、平均 CPI を計算せよ。

さて、キャッシュを装備したメモリシステムの性能は、ミス率とミス時のペナルティで決まることがわかる。これらを小さくする方法を簡単に紹介する。

10.3.1　ミス率を減らす

ミスの原因

キャッシュミスは、以下の 3 つの原因で生じる[1]。

- 容量ミス (Capacity miss): キャッシュ容量の絶対的な不足によって起きる
- インデックス衝突ミス (Conflict miss):インデックス番号の重なるブロックは、ダイレクトマップキャッシュや way 数が不足すると、キャッシュ全体の容量が不足していなくても、格納することができない。これに起因するミス。
- 初期化ミス (Cold start miss/Compulsory miss):ジョブのスタート時、あるいは、切り替え時には、キャッシュ上に有効データが存在しないことにより起きるミス。ちなみに前節の実行例で生じたミスは全てこのタイプのミスである。

[1] 3 つの C と呼ぶ

キャッシュの全体容量を増やせば、ミス率が減るのは当然だが、その分のコスト増は避けられない。したがって、設計時には全体容量は決まってしまっており、その中で少ないコストでミス率を抑える構成を取ることが要求される。設計の目安は以下の通りである。

ブロックサイズ

アクセス局所性により、ブロックサイズを大きくするとミス率は下がる傾向にあるが、大きなブロックを主記憶から持ってくるための転送時間がかかるようになる。また、キャッシュの容量が小さい時に大きなブロックサイズを選択すると、キャッシュ中に格納可能なブロック数自体が少なくなってしまい、インデックス衝突ミスが増加してミス率が増えてしまう。このため、16byte-64byte の範囲内で、キャッシュサイズが大きい場合やや大きめに設定するのが有利である。

way 数

増やすことにより、インデックス衝突ミスを減らす効果があるが、数が大きくなる程、その効果は頭打ちになり、4 より大きくてもあまり効果がない。ブロックサイズが大きい場合は、インデックス衝突ミス自体が少ないため、way 数の効果がさらに頭打ちになる。一般的に 1 つまりダイレクトマップから 4-way までの構成を取る。オンチップキャッシュ等、キャッシュ自体の動作を高速化したい場合はダイレクトマップ方式が有利な場合もある。

その他のテクニック

ミス率を抑えるためのテクニックとしては以下の方法が用いられる。

- プリフェッチ: アクセスしそうなブロックを先取りしておくことでミス率を減らす方法。ハードウェアによる方法とソフトウェアによる方法がある。ハードウェアによる方法は、命令のプリフェッチが代表的で、最近のほとんどのプロセッサで用いられている。この方法では、命令フェッチの際に、次のいくつかのブロックまで先行してフェッチしてしまう。フェッチされたブロックはキャッシュに格納せず、ストリームバッファとよばれる小規模高速の専用メモリに格納しておく。命令フェッチ時にキャッシュがミスすると、ストリームバッファをアクセスし、存在すればこちらから命令を持って来ると共に、キャッシュ上にも転送する。一方、ソフトウェアによるプリフェッチは、コンパイラまたはプログラマがあらかじめコード中にプリフェッチ命令を埋め込んでおく。

- データ構造の変更: データについてのキャッシュは、データの配置を変えることによってミス率を低減することができる。同一配列を扱うループ構造をまとめ

てしまったり、配列の添字を入れ替えてなるべく番地順にアクセスしたり、配列自体をキャッシュに入るようにブロックしたりする方法が用いられる。

10.3.2　ミスペナルティの削減

Early restart と Critical word first

前節に示した実装では、キャッシュミスの際、メインメモリからキャッシュブロックを転送し、それが終わるまでCPUを待たせておいた。この方法を取ると、ミス時のペナルティはブロックのサイズに対応して大きくなってしまう。そこで、全部キャッシュに転送するまで待たず、CPUのアクセスしたデータが到着したらそれをCPUに渡し、CPUは次の命令を開始する方法が用いられる。これをEarly restartと呼ぶ。さらに、この方法を取る際に、CPUが要求したワードを先に転送し、あとは、サイクリックに1ブロック分を転送すれば、さらにCPUの待ち時間を短縮することができる。この方法をCritical word firstと呼ぶ。ただし、転送を開始するアドレスに制限があるDRAMもあり、このような場合はCritical word firstは使えない。これらの方法によりキャッシュコントローラは複雑になるが、メモリ構成を変更せずに、ミスペナルティを削減することができる。

階層キャッシュ

キャッシュの階層を増やすのも、ミスペナルティの削減を目的としていると考えることができる。前節で紹介した通り、最近のプロセッサは前回プリントの図に示すようにオンチップの1次キャッシュ(Level-1: L1キャッシュと呼ぶ)と、オンボードの2次キャッシュ(Level-2: L2キャッシュと呼ぶ)の2階層構造が一般的である。しかし、プロセッサの動作周波数が上がるにつれて、チップ内でも階層構造を持たせ、全体として3階層以上になる場合も増えている。基本的に1次キャッシュに入るデータは必ず2次キャッシュにも入っているように追い出し方法等を工夫する。1次キャッシュと2次キャッシュではブロックサイズやway数を変える場合もある。

キャッシュについては、性能に与える影響が大きいことから、ここでは紹介されていない様々な高速化技法が提案され、用いられている。

10.4　仮想記憶方式

キャッシュで行なった記憶階層に関する手法は、主記憶と2次記憶間でも行なわれる。例えば32bitマイクロプロセッサでは論理アドレスは4GB取ることができるが、実際に搭載するメモリは例えば16M-64MBなどこれより小さい場合が多い。この場合、プログラムは論理的なアドレス空間を全て利用できると考えて、論理アドレス空

間 (仮想アドレス空間) で動作させ、一方、実際の記憶の実体は物理アドレス空間上のメモリにマップする。これを仮想記憶方式と呼ぶ。この際、データ管理の単位はページと呼び、キャッシュブロックよりも大きく 4KB 程度に設定する。

通常、論理アドレスと物理アドレスの変換は CPU からアドレスが出た直後に行われる。これは、複数のプログラムを同時並行的に処理する場合、論理アドレスは重複する可能性があり、物理アドレスに変換しないでこれを解決するのが難しいためである (シノニム問題と呼ばれる)。プログラムが出力した論理アドレスは、アドレス変換用のテーブルによって物理アドレスに変換されるのだが、対象のアドレス幅が大きいため、このためのテーブルは巨大になってしまう。この変換を一次キャッシュの手前で高速に行うことが仮想記憶方式の問題である。

このための機構が TLB(Translation look-aside buffer) と呼ばれる小容量のバッファで、変換テーブルのキャッシュとして働き、高速にアドレス変換を行なう。アドレス変換テーブルの本体は、主記憶上に置かれるが、まともに作ると巨大なものとなるため、ハッシュ等の技術を用いて、OS の記憶管理プログラムがソフトウェアで管理する。TLB によるアドレス変換の様子を図 10.10 に示す。この例では論理アドレス空間を 4GB(32bit) とし、物理アドレス空間は 64MB(24bit)、ページは 4KB とした。この場合、アドレスの下位 12bit はオフセットとなり、論理／物理アドレス共に共通となり、変換の対象は、31bit 目から 12bit 目までの 20bit の論理ページ番号となる。

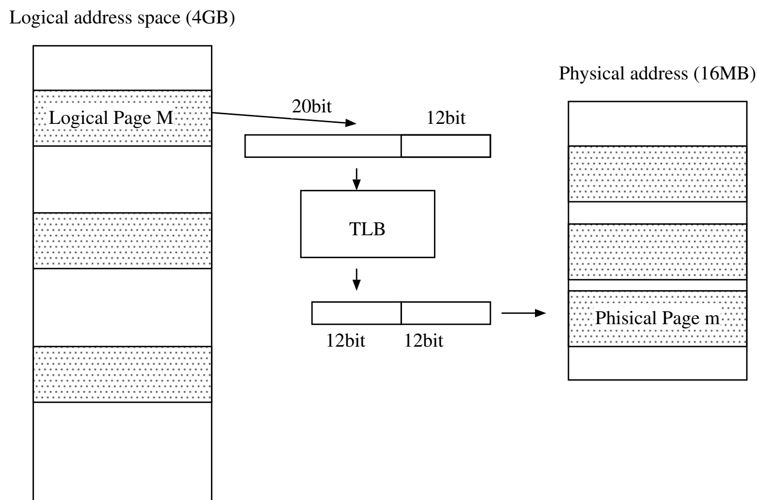


図 10.10: 仮想記憶のアドレス変換

TLB は、この 20bit の論理ページ番号から、12bit の物理ページ番号を生成するテーブルである。キャッシュ同様ページ番号を二つに分割して下位アドレスで参照して上位アドレスで照合するダイレクトマップ方式も考えられるが、高速性と conflict miss

をなくすために、小容量の連想メモリ (Content Addressable Memory) あるいは多入力のマルチプレクサを用いてフルアソシアティブ方式とする場合もある。

図 10.11 に示すように、表の各ビットは有効かどうかを示すビット (V)、プロテクション情報、Dirty bit より構成される。

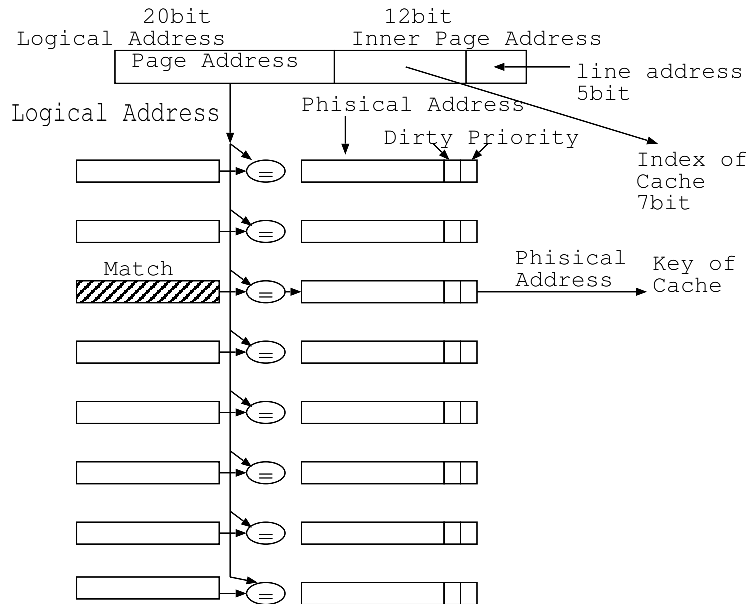


図 10.11: TLB

論理アドレスが TLB 上に存在すれば、物理アドレスとプロテクション情報が読み出され、存在しなければページフォルト (TLB ミス) として例外処理が生じ、主記憶上のテーブルが OS のメモリ管理プログラムによって参照される。そして、アクセスされたページが主記憶上に存在しなければ、2 次記憶との間にページの入れ替えが行なわれる。2 次記憶にページを書き戻すことをスワップアウト、2 次記憶からページを読み込むことをスワップインと呼ぶ。スワップアウトするページはキャッシュ同様 LRU で選ばれる。

TLB がヒットした場合は、プロテクション情報をチェックする。ページには、OS などの特権モードでのみアクセスできるページと、ユーザがアクセス可能なページがあり、それぞれに Read/Write 等の保護を行なっている場合が多い。特権モードのページに対してアクセスしようとすると、Protection Violation の error が生じ、これも例外処理となる。

さらに、始めて当該ページに対して書き込みを行なう場合、主記憶の方のテーブルを更新する必要があるので、これも例外処理となる。更新後はそのことを示すため、Dirty bit がセットされる。

TLB は、キャッシュの前段に置かれなければならないため、高速性が必要とされる。最も容易な実装法は、キャッシュのアドレス変換はページ内のアドレスのみを用い、TLB による変換とキャッシュのタグ引きを同時行なう方法である。この方法は小規模なシステムには有効だが、キャッシュの容量がページサイズ X way 数に制限される。

第 11

CPU のさらなる高速化技術

本書の目的である「コンピュータアーキテクチャの入門から一通りパイプライン化されたプロセッサの設計ができる」は、今までの章で概ね記述が終わった。以下、最近のプロセッサに使われている技術をざっと紹介し、今後の学習への繋ぎとしたい。

11.1　基本的な高速化技術

11.1.1　命令の動的スケジューリング

POCO のパイプラインは通常ストールを生じないが、4 ステージしかないので動作周波数をあまり高くすることができない。CPU の性能向上のために、最も直接的な方法は動作周波数を上げることであり、このため 90 年代以降、パイプラインの段数はどんどん多くなり、10-20 ステージに及んだ。長いパイプラインは種々の要因でストールするために、これを防ぐ技術が発達した。

アウトオブオーダ実行　データハザードにより、ある命令が結果を待ってストールしている時に、その結果を用いない後続の命令が依存関係がなければ、待っている命令を追い越して先に進んでしまう方法。命令の非順序実行 (out-of order execution) と呼ぶ。コンパイラでは、解析仕切れない場合でも動的に条件が満足されれば、命令の追い越しができるため、効率的であるが、追い越す命令の条件、フォワーディングを管理するハードウェアの構造が複雑になる。効率的な実行をシステマチックに管理するために、トーマスローの方法などが提案され、利用されている。

分岐予測　パイプラインが長くなると分岐命令による制御ハザードが深刻な問題となった。このため、分岐が成立するかどうかを予測する分岐予測の様々な方式が提案されている。まず、分岐命令自体を識別するため、実行した分岐命令をキャッシュの

形で CPU 内に持つ分岐予測バッファを持たせる。このバッファの上でそれぞれの分岐命令に対して分岐が行われるかどうか予測を行う。

最も簡単な分岐予測は「以前実行した命令が分岐したから次も分岐するだろう」という 1bit 分岐法だが、これはループの構造が少しでも複雑になると役に立たなくなる。このため、図 11.1 に示すように、2bit 設けて 4 状態とし、予測が２度はずれたら、次の予測を変更する 2bit 分岐法を導入すると、予測精度はさらに改善される。

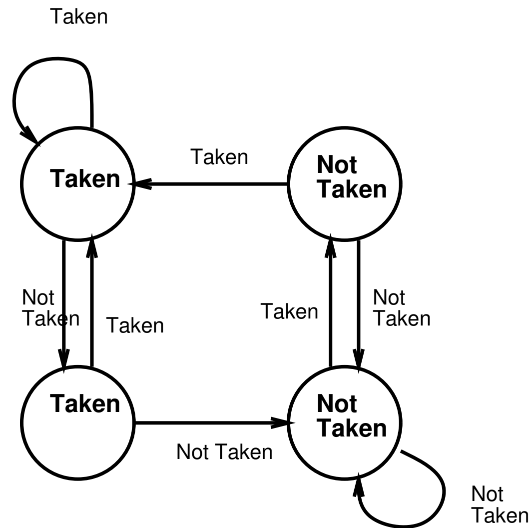


図 11.1: 2bit 分岐予測方式

さらに精度を上げるために、その前に実行された他の分岐命令が分岐したかどうかで予想を変える方法なども提案されている。また、分岐した場合の分岐先アドレスを高速に得るために、分岐予測バッファに分岐先アドレスを記憶しておく方法も用いられる。最近の高速 CPU では、いくつかの予測法を選択的に用いるなどさらに高度な分岐予測が用いているものもある。

投機的実行:　高度な分岐予測を用いれば多くの場合に予測は的中するため、当ると見込んで既に処理を実行してしまう場合もある。このような方法を投機的実行 (Specurative Exection) と呼ぶ。投機的実行は、予測がはずれた場合に、実行をキャンセルしなければならないため、とりあえず実行した結果を取っておき、その後、予測が当たった場合にはこれを確定させる機構が必要となる。

11.1.2 命令レベル並列処理

スーパスカラ

長いパイプラインのストールを分岐予測、アウトオブオーダ実行、投機的実行を組み合わせて抑える方法にも限界がある。これらはすべて、CPI を 1 に近づけるための方法であり、これ以上、性能を向上させるために、CPI を 1 以下にすることを考える。

このためには、複数の命令を同時にフェッチし、実行する機構を備える必要がある。このような機構をスーパスカラ方式と呼ぶ。図 11.2 にスーパスカラ方式の基本概念を示す。メモリ中の命令は、種類毎に分かれてキャッシュから同時に CPU に送られ、同時にパイプラインに投入されて実行される。当然、キャッシュから CPU には二セットのバスが必要だが、同一チップ上のオンチップキャッシュならばこのことは十分可能である。この方式は、複数の命令を同時に発行することから命令レベル並列性 (Instruction Level Parallellism: ILP) を利用した方式と呼ばれる。

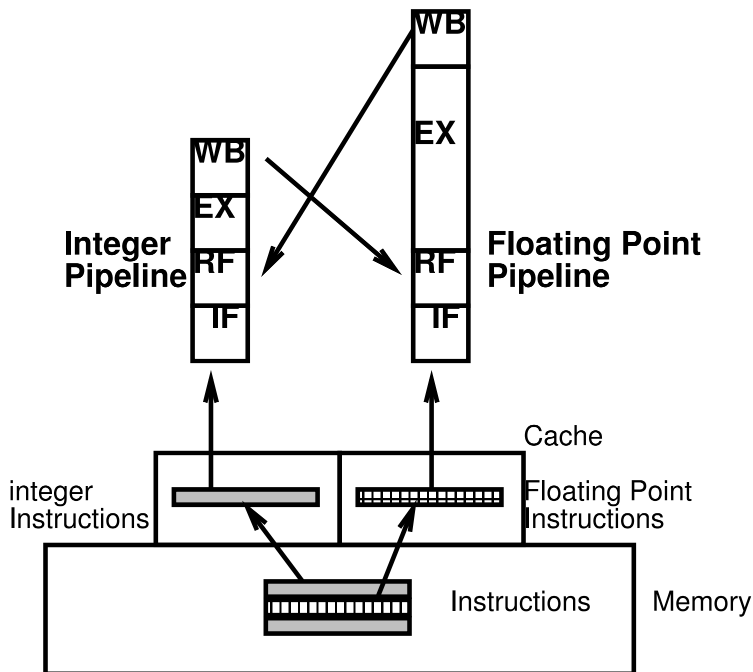


図 11.2: スーパスカラ方式（2 命令同時発行）

図 11.2 では、二命令同時発行のスーパスカラを示す。同時に発行される命令は、なるべく独立性が高いことが望ましいので、二命令同時発行の場合は、整数パイプラインを用いる整数演算命令と、浮動章演算パイプラインを用いる浮動小数演算命令に分けるなどする。発行された命令間にはもちろん依存性がある場合が考えられるので、

実行パイプライン間のフォワーディング、待ち合わせ等は、パイプラインの動的実行と同様にハードウェアで管理する必要がある。三命令以上を同時に発行する場合、(1) 浮動小数演算命令を複数発行する。(2) メモリアクセス命令を独立に発行する。等の方法がある。最近の高性能マイクロプロセッサの多くは、三命令あるいは四命令同時発行のスーパスカラ構成を取っている。このうちどの程度の命令が同時に発行可能かは、実行するプログラムによって異なるが、平均して二を越えるようにするのは困難である。

スーパスカラ方式には、アウトオブオーダ実行および投機実行を行うものとこれらを行わないものがある。後者はハードウェアが簡単ですむため、組み込み用の CPU で用いられる。

VLIW(Very Long Instruction Word) 方式

スーパスカラ方式は、複数命令を同時に発行するためのハードウェア制御が複雑で、全体としてプロセッサの制御部が非常に複雑になる。このため、実行周波数を落とさずに多数の命令を同時発行するのは困難である。そこで、図 11.3 に示すように、一つのフィールドが、普通のコンピュータの命令に相当するような非常に長い命令を設け、コンパイラにより同時に実行可能な命令を見つけてフィールドを埋めることによって、複数命令同時発行と同様な性能向上を実現する方法を VLIW(Very Long Instruciton Word) 方式と呼ぶ。

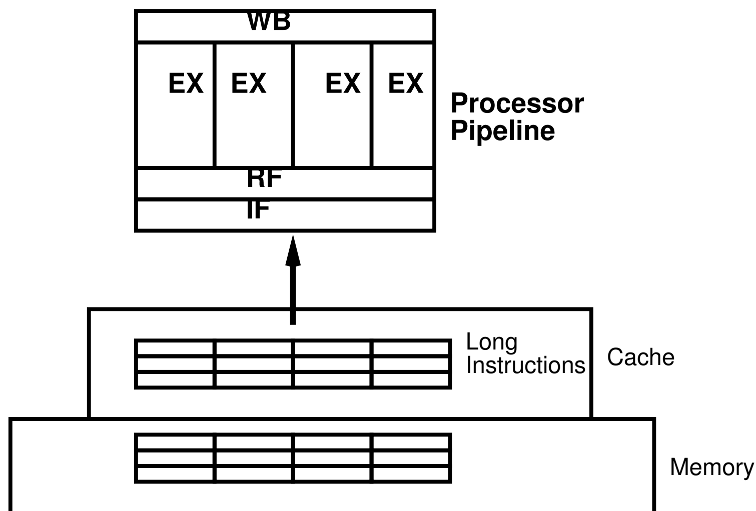


図 11.3: VLIW 方式（4 命令分のフィールドを持つ）

VLIW 方式は、命令実行の依存関係、データの依存関係の調整を全てコンパイラに任せているため、制御部はスーパスカラに比べて単純化することができる。このため、

同時実行可能なフィールドは 5 以上設けることが可能であるが、長い命令のそれぞれのフィールドに対応する演算装置、バス等のデータパスは同時実行を構造化ハザード無しに行えるように、きちんとフィールドに対応して用意する必要がある。また、フィールドを埋めるために、トレーススケジューリング等、高度なコンパイル技術が必要となり、既存の命令コードとの互換性が保たれない問題もある。このため、VLIW 方式は、主に信号処理向けプロセッサ、携帯用プロセッサなどで用いられている。

VLIW 方式とスーパスカラ方式の中間的な方式として EPIC(Explicitly Parallel Instruction Computer) が Intel 社の 64bit マイクロプロセッサ Itanium に用いられた。しかし、この方法は両者の良い点と共に悪い点も合わせることになり、Itanium では期待されたほど効果を発揮しなかった。このため、VLIW、EPIC 方式はやや特殊な方式として位置づけられている。

11.1.3 より大きなレベルの並列処理

長いパイプラインを用いて周波数を上げ、命令レベル並列性 (Instruction Level Parallelism: ILP) を利用して高速化する手法は 2000 年代の始めころまでは有効であった。しかし、2004 年以降、この方式は以下の原因で限界を迎えた。(1) チップの発熱がひどくなり、これ以上周波数を上げることが難しくなった。(2) 利用できる命令レベルの並列性が限界に達した。(3) キャッシュを含めたメモリの遅延により、周波数を上げてもさほど性能が向上しなくなった。

そこで、PC を複数持つことで、並列に複数のスレッド (Thread) を実行することのできるプロセッサの開発が盛んになった。ILP にとらわれず、もっと大きな処理の単位で並列に実行すれば、もっと多くの処理を同時に実行することができ、性能を向上できる可能性がある。これがいわゆるマルチコアプロセッサであり、既に IBM の Power5、Intel の Core2Duo など 1 チップ上に 2-4 程度のプロセッサを実装する方式が普及している。さらに、もっと多く、16-256 のプロセッサを 1 チップに搭載したメニーコアと呼ばれるプロセッサも開発されている。しかし、これらのプロセッサでは、単純に今までのプログラムを動かしただけでは高速化されず、プログラマが明示的に並列に処理を記述するか、コンパイラが並列化を行わなければならず、ソフトウェアの開発が問題点となっている。

第 12

参考文献

[1] D.A.Patterson and H.L.Hennessy. Computer Organization and Design, Fourth Edition. *Morgan Kaufmann (*パターソン& ヘネシー：コンピュータの構成と設計、成田光彰訳 日経 *BP* 社*)*, Nov. 2008.

[2] H.L. Hennessy and D.A.Patterson. Computer Architecture: A Quantitative Approach. *Morgan Kaufmann (*ヘネシー& パターソン：コンピュータアーキテクチャ定量的アプローチ、中條、吉瀬、佐藤、天野訳、翔泳社, Nov. 2006.

[3] D.M.Harris and S.L.Harris. Digital Design and Computer Architecture. In *Morgan Kaufmann (*ディジタル回路設計とコンピュータアーキテクチャ、鈴木、天野、中條、永松訳、翔泳社*)*, Aug. 2009.