コンピュータアーキテクチャ チートシート

1. 基本性能評価

重要公式

- **ICのコスト (IC Cost)**
  - ダイ1個あたりのコスト: $Cost\ per\ die = \frac{Cost\ per\ wafer}{Dies\ per\ wafer \times Yield}$
  - ウェーハあたりのダイ数: $Dies\ per\ wafer \approx \frac{Wafer\ area}{Die\ area}$
  - 歩留まり: $Yield = \frac{1}{(1+(Defects\ per\ area \times Die\ area/2))^2}$
- **性能と実行時間 (Performance and Execution Time)**
  - 相対性能: $性能 = \frac{1}{実行時間}$
  - **CPU実行時間 (CPU Time):**
    $CPU時間 = CPUクロックサイクル数 \times クロックサイクル時間 = \frac{CPUクロックサイクル数}{クロック周波数}$
    $CPU時間 = \frac{実行命令数 \times CPI}{クロック周波数}$
    $CPU時間 = 実行命令数 \times CPI \times クロックサイクル時間$
- **消費電力 (Power)**
  - CMOS ICの消費電力: $Power = Capacitive\ load \times Voltage^2 \times Frequency$

基本用語

- **CPI (Cycles Per Instruction):** 命令あたりの平均クロックサイクル数。
- **IPC (Instructions Per Cycle):** 1サイクルあたりの実行命令数。CPIの逆数。
- **応答時間 (Response Time):** タスクを完了させるまでに必要な合計時間。
- **スループット (Throughput):** 単位時間あたりに終了した作業量。
- **Amdahlの法則 (Amdahl's Law):** システムのある部分を改善しても、その改善が全体の性能向上に寄与する割合には限界があるという法則。
- **電力の壁 (Power Wall):** 消費電力と冷却の問題により、クロック周波数の向上が限界に達している状況。

---

2. MIPS命令セットアーキテクチャ (ISA)

命令フォーマット

- **R形式 (R-type):** 主にレジスタ間の算術・論理演算に使用。
  フィールド: `op | rs | rt | rd | shamt | funct`
- **I形式 (I-type):** 即値を含む演算、ロード/ストア、条件分岐に使用。
  フィールド: `op | rs | rt | address/immediate`
- **J形式 (J-type):** 無条件ジャンプに使用。
  フィールド: `op | target address`

アドレッシングモード

- **即値アドレッシング:** オペランドが命令内に含まれる定数。
- **レジスタアドレッシング:** オペランドがレジスタ。
- **ベース相対アドレッシング:** レジスタの値と命令内の定数（オフセット）の和でメモリアドレスを指定。
- **PC相対アドレッシング:** プログラムカウンタ(PC)からの相対アドレスで分岐先を指定。
- **疑似直接アドレッシング:** ジャンプ命令で、PCの上位4ビットと命令内の26ビットアドレスを結合してジャンプ先を指定。

C言語とアセンブリの対応

C言語の配列アクセスとポインタ演算は、MIPSでは以下のようにアドレス計算を経て実現される。

- **配列アクセス `array[i]`:**
  1. インデックス i を4倍する（int型は4バイトのため）。(sll $t1$,t0, 2 ※$t0=i)
  2. ベースアドレス（配列の先頭アドレス）に加算する。(add $t2$,s0, $t1$※s0=arrayのアドレス)
  3. 計算したアドレスにアクセスする。(lw $t3, 0$(t2) または sw $t3, 0$(t2))
- **ポインタ演算 `*p = 0; p++;`:**
  1. ポインタ p が指すアドレスにアクセスする。(sw $zero, 0$(t0) ※$t0=p)
  2. ポインタ p の値を4（int型ポインタの場合）だけインクリメントする。(addi $t0$,t0, 4)

主要命令とレジスタ規約

- **主要命令:**
  - 算術演算: `add`, `sub`, `addi`
  - データ転送: `lw` (ロード), `sw` (ストア)
  - 条件分岐: `beq` (等しいなら分岐), `bne` (等しくないなら分岐)
  - 比較: `slt` (より小さいならセット), `slti` (即値で比較)
  - 無条件分岐: `j` (ジャンプ), `jal` (ジャンプしてリンク), `jr` (レジスタへジャンプ)
  - **move命令:** `move $t0, $s0` は、実際には `add $t0, $s0, $zero` や `addi $t0, $s0, 0` のような命令で実現される疑似命令。
- **レジスタ規約:**
  - `$a0`-`$a3`: 引数用
  - `$v0`-`$v1`: 戻り値用
  - `$t0`-`$t9`: 一時変数用（呼び出し側で待避責任）
  - `$s0`-`$s7`: 保存用変数（被呼び出し側で待避責任）
  - `$sp`: スタックポインタ
  - `$ra`: 戻りアドレス
  - `$fp`: フレームポインタ
  - `$gp`: グローバルポインタ

---

3. 算術演算

重要公式

- **IEEE 754 浮動小数点数表現:**
  - 単精度: $(-1)^S \times (1 + Fraction) \times 2^{(Exponent-127)}$

- 倍精度: $(-1)^S \times (1 + Fraction) \times 2^{(Exponent - 1023)}$

主要アルゴリズム

- **乗算アルゴリズム:**
  1. 64ビットの積レジスタの下位32ビットに乗数をセット。被乗数レジスタを用意。
  2. 積レジスタの最下位ビット(LSB)をチェックする。
  3. もしLSBが1なら、被乗数を積レジスタの上位半分に加算する。
  4. 積レジスタ全体を1ビット右にシフトする。
  5. この処理を32回繰り返す。最終的な積が64ビットレジスタに得られる。
- **除算アルゴリズム:**
  1. 64ビットの剰余レジスタの右半分に被除数をセット。除数レジスタを用意。
  2. 剰余レジスタを1ビット左にシフトする。
  3. 剰余レジスタの上位半分から除数を引く。
  4. 結果が0以上なら、剰余レジスタを1ビット左にシフトし、LSBに1をセットする。
  5. 結果が負なら、除数を足して値を戻し、剰余レジスタを1ビット左にシフトしてLSBに0をセットする。
  6. この処理を32回繰り返す。商は剰余レジスタの下位半分に、剰余は上位半分に残る。

基本用語

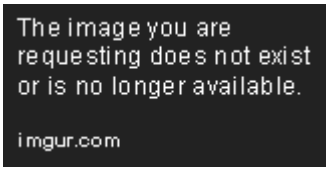
- **2の補数 (2's Complement):** 負数を表現する方法。ビットを反転して1を加えることで得られる。
- **符号拡張 (Sign Extension):** 短いビット長の符号付き数を長いビット長に変換する際、符号ビットを上位ビットにコピーすること (`lb` と `lbu` の違い)。
- **IEEE 754:** 浮動小数点数の標準規格。符号部、指数部、仮数部から構成される。
- **仮数部 (Fraction/Mantissa):** 浮動小数点数の有効数字部分。
- **指数部 (Exponent):** 浮動小数点数の桁を表す部分。大小比較を容易にするため、実際の値にバイアス（ゲタ）を加えたゲタばき表現が用いられる。
- **非数 (NaN - Not a Number):** 0÷0など、不正な演算結果を示す特別な値。

---

4. プロセッサのデータパスと制御

データパスと制御信号

単一サイクルプロセッサは、1クロックで1命令を実行する。データパスは命令メモリ、レジスタファイル、ALU、データメモリ等から構成され、制御ユニットが命令に応じて制御信号を生成する。



- **データパスの構成要素:**
  - **命令メモリ:** PCが示すアドレスから命令を読み出す。
  - **PC (プログラムカウンタ):** 次に実行する命令のアドレスを保持する。

    - **レジスタファイル:** レジスタの読み書きを行う。
    - **ALU (算術論理ユニット):** 算術演算や論理演算を実行する。
    - **データメモリ:** `lw` や `sw` 命令でデータの読み書きを行う。

- **主要な制御信号の機能:**

    - **RegDst:** 書き込み先レジスタを`rt` (0) と `rd` (1) のどちらにするか選択。
    - **ALUSrc:** ALUの第2入力をレジスタ(`rt`) (0) と即値 (1) のどちらにするか選択。
    - **MemtoReg:** レジスタへの書き込みデータをALUの実行結果 (0) とメモリからの読み出しデータ (1) のどちらにするか選択。
    - **RegWrite:** レジスタファイルへの書き込みを有効化 (1) するか。
    - **MemRead:** データメモリの読み出しを有効化 (1) するか。
    - **MemWrite:** データメモリの書き込みを有効化 (1) するか。
    - **Branch:** 分岐命令 (`beq`) であり、かつALUのゼロ判定が真の場合にPCを分岐先アドレスに更新する。
    - **Jump:** PCをジャンプ先アドレスに更新する。
    - **ALUOp:** 命令のopコードに基づき、ALU制御ユニットに送られる信号。ALU制御ユニットはこれとfunctフィールドから最終的なALU操作を決定する。

ALU制御

命令のopコード	ALUOp	命令操作	functフィールド	ALUの演算	ALU制御コード
lw/sw	00	load/store	xxxxxx	加算	0010
beq	01	branch equal	xxxxxx	減算	0110
R形式	10	add	100000	加算	0010
R形式	10	sub	100010	減算	0110
R形式	10	and	100100	AND	0000
R形式	10	or	100101	OR	0001
R形式	10	slt	101010	set on less than	0111