マイクロプロセッサ演習

2004 年度

第 6 回

1　今日の演習の内容 (手続きと関数)

「手続き (procedure)」や「関数 (function)」とは、いくつかの命令をひとまとめにし、プログラムの他の部分から呼び出せるようにしたものである。特に、戻り値を持たないものを「手続き」、戻り値をもつものを「関数」と呼ぶ。

```
int main(void){
     ...
     hello();
     ...
}

void hello(void){
     printf("Hello World.\n");
}
```

図 1: 手続き hello

例えば、図 1 は C 言語において「Hello World と画面に表示する」という機能を手続き hello として実現したものである。main (プログラムのメイン部分) の外部に手続き hello の宣言を置き、main の内部で hello() として呼び出している。

```
int main(void){
     ...
     a=5;
     b=square(a);
     ...
}

int square(int i){
     return(i*i);
}
```

図 2: 関数 square

一方、「整数 i の 2 乗を与える」機能を関数 square として C 言語で実現したのが図 2 である。このプログラムを実行すると b には a の 2 乗、すなわち 25 が代入される。

以上の 2 つを MIPS のアセンブリ言語で実現するのが今回の演習である。なお、「手続き」と「関数」の用語の区別は C 言語では存在せず、どちらも「関数」と呼ばれることに注意しておく。

2　[SPIM] 手続き hello の MIPS のアセンブリ言語による実現

2.1　jal (jump and link) と レジスタ $ra

本章では、前章でみた手続き hello を MIPS のアセンブリ言語を用いて実現する。演習の Web ページより、data06.zip をダウンロードせよ。

HelloFunc.asm がこの問題で取り扱うプログラムファイルである。これはこのままで完成しているので、シミュレータで動かすことができる。(ただし問題の最後に、少しプログラムに手を加えてもらうことになる)

このプログラムをテキストエディタで開いて中身を確認し、内容を理解しよう。まず、HelloFunc.asm をロードしたときのメモリのテキスト領域の様子の模式図を示したのが図 3 である。前回の Switch 文のところでも触れたように、プログラムの本体はメモリのテキスト領域に格納される。(テキスト領域は、Windows 版 SPIM (pcspim.exe) では上から 2 番目の窓で見ることができる。)

プログラムのメイン部 (main:) はメモリアドレス 0x00400020 から始まる [1]。それより前の領域はシミュレータが main を呼び出すために使われる。まず、次の 2 つの事実を了解して欲しい。

- **main も手続き (関数) である。**

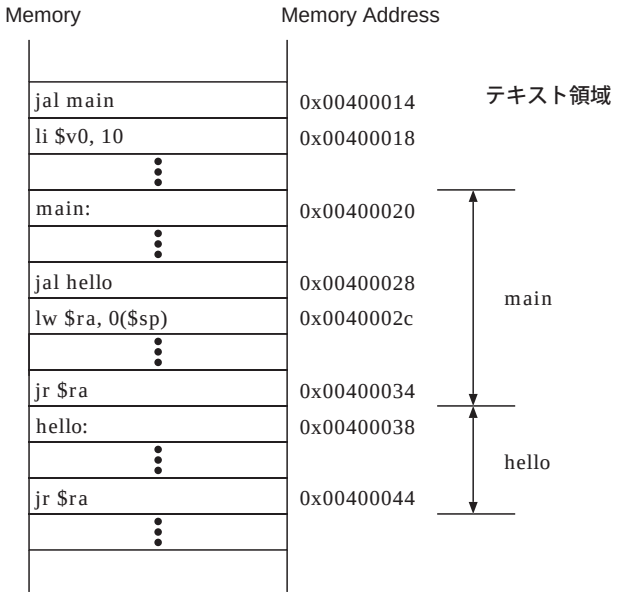

図 3: HelloFunc.asm をロードした時の、メモリのテキスト領域の様子。

- **手続きは「jal (手続き名)」で呼び出される。**

すなわち、プログラムの本体 main は「`jal main`」という命令で呼びだされる。これはメモリアドレス 0x00400014 で実行されている。

手続き main が `jal` 命令で呼び出されてから終了した後、処理は元の位置の次の命令、すなわちメモリアドレス 0x00400018 に戻らねばならない。それはジャンプ命令 「`jr $ra`」 で実現される。

- **手続きが終了したら、`jr $ra` で元の位置に戻らねばならない。**

$ra (return address) は特別なレジスタで、「関数を呼び出したアドレスの次のアドレス」の値が `jal` を呼び出した時に (自動的に) 代入されている。つまり、`jal main` が実行されると $ra = 0x00400018 にセットされ、`jr $ra` を実行することで main を呼び出した所の次の命令に処理が戻る。

今まで扱ったプログラムは全て最後に `jr $ra` が実行されていたが、これは以上のような理由による。

2.2 スタック、スタックポインタ $sp

さて、ここまでで (手続き) main が呼び出されて処理が終了するまでの流れを概観した。

もともとの意図を思い出すと、「main という手続きの中から、hello という別の手続きを呼び出す」必要がある。

想像がつくように、hello という手続きを定義しておき、main の内部で `jal hello` を実行すれば目的は達成される。

しかし、`jal hello` を実行すると$ra が新たなアドレス (図 3 では 0x0040002c) で上書きされ、main 終了後の戻りアドレス 0x00400018 が消されてしまい、プログラムが正しく終了できない、という問題が起こる。

そこで「手続き hello を呼び出す前に $ra の値を保存しておく」必要が生じる。このように、レジスタの値等を一時的に保存するための記憶領域として**スタック**というメモリ領域が用意されている。

- **手続きを呼び出す前に、保存しておきたいデータをスタックに退避させておく。**

スタックは、メモリアドレス 0x80000000 より low address のメモリが high から low に向かって (図では下から上へ) 使われる。なお、配列 (データ領域) の場合は low から high へだったが、スタックは逆に用いられるので注意。

スタックはメモリ上にあるので、今まで同様 `lw` や `sw` を用いて値を読み書きできる。ただし、読み出しや書き出しを行なう位置はスタックポインタ$sp という特別なレジスタに保持しておく規約になっている。

ここで、スタックポインタ $sp を用いて戻りアドレス $ra を退避する方法を図 4 に示す。(なお、メモリのロード/ストアに対応するものを、スタックではポップ/プッシュと呼ぶ)

図で網かけの領域はシミュレータで使われる領域であるのでそこにはアクセスせず、それより low address の領域を用いるようにする。(なお、教科書などではスタックの説明をするのに上下逆の絵を描く慣習だが、ここでは混乱を避けるため今まで通りの描き方で統一する。)

図にあるように、スタックに退避 (プッシュ) するときは$sp の値を 4 引いてから従来通り `sw` で値を書き込む。

逆に、スタックから値を戻す時は `lw` で値を読み込んでから $sp の値に 4 を足し、初期状態に戻しておく。

以上で HelloFunc.asm のプログラムの内容を理

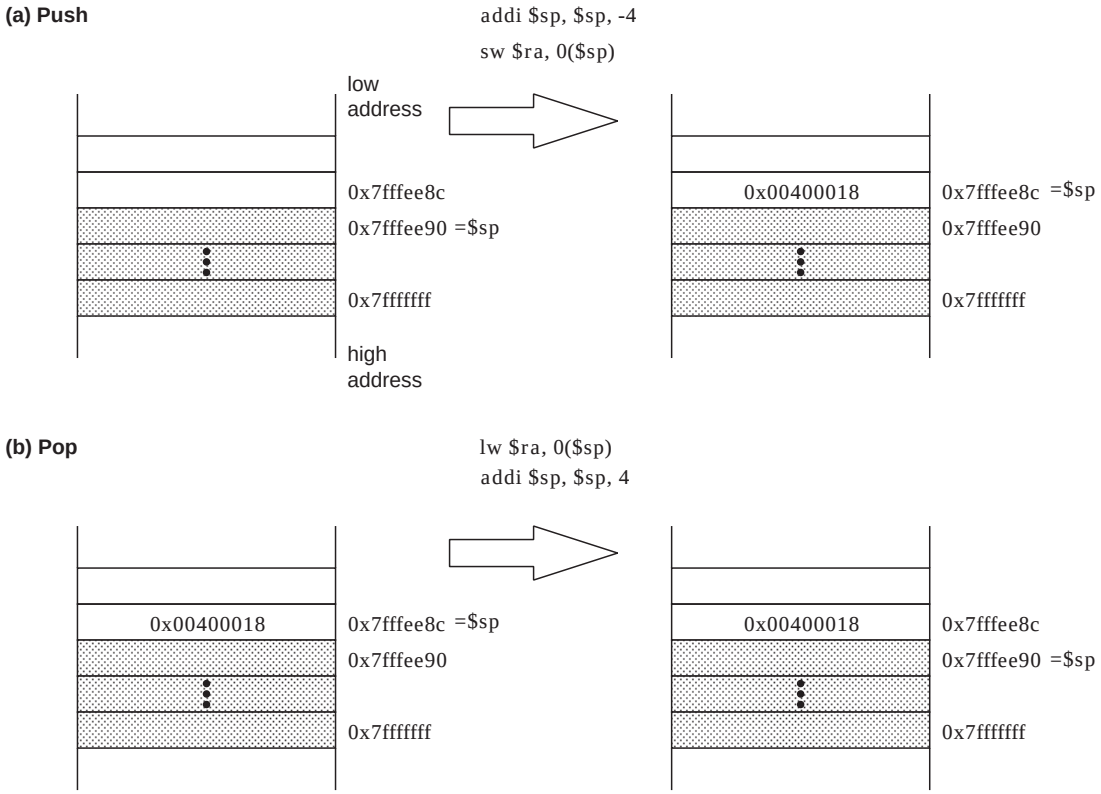

図 4: スタックに対するプッシュとポップ

解する準備が整った。プログラムの中身を見て、何が行なわれているのか理解してみよう。

また、プログラムを少し修正して以下の設問に答えよ。

1. プログラム中に「この位置における $ra の値は？」とかかれた部分が 2 ヵ所ある。この位置における $ra の値を調べてみよ。$ra の値を適当なレジスタに代入して、Windows 版 SPIM のウィンドウ上で確認すれば良いだろう。(ヒント: $t0 = $ra + 0 など)

2. 上で調べた $ra の値は何を意味しているか考察せよ。

3 [SPIM] 関数 square の MIPS のアセンブリ言語による実現

本問題では、第 1 章に出て来た関数 square を MIPS のアセンブリ言語で記述することを考える。ファイルは Square.asm である。

さきほどの HelloFunc.asm との違いは「引数」と「戻り値」があることである。引数とは、関数 square の `i` に相当し、戻り値は `return(i*i)` で返される値のことを指す。

MIPS のアセンブリ言語では、関数の引数として $a0 ··· $a3、戻り値として $v0、$v1 という特別なレジスタを用いる約束になっている。

さて、以上に注意して Square.asm を完成させよ。

A [補足] 特別なレジスタ

ここまでに出て来たレジスタをまとめると以下のようになる。

- $zero 常に 0 を保持
- $v0, $v1 関数の戻り値を保持
- $a0 ··· $a3 関数への引数を保持
- $t0 ··· $t7 演算の途中での一時的データ
- $s0 ··· $s7 プログラムの変数の値を保持
- $sp スタックポインタ

- $ra 関数からの戻りアドレス

B [補足] C 言語でのメモリの利用のされ方

B.1 メモリの 4 領域

本演習ではこれまでメモリの領域として「**テキスト領域**」、「**データ領域**」、「**スタック領域**」の 3 つを取り扱った。本章ではこれらのメモリ領域が C 言語や C++ 言語でどのように使われるかを簡単に解説する。本章は C 言語の知識が必要であるため、自信が無い人は読み飛ばして構わない。なお、各領域の名称は処理系によって異なることがある (たとえば 80x86 系では「テキスト領域」は「プログラム領域」と呼ばれる、など) が、その本質は多くの処理系で共通している。

まず、C 言語で扱うメモリの論理的な模式図は図 5 のように 4 つに分けられる。このうち「静的領域」

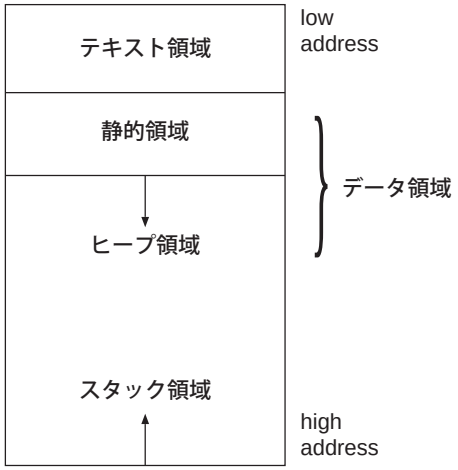

図 5: 論理的なメモリ上の模式図。

と「ヒープ領域」を合わせて「データ領域」と呼ぶこともあり、本演習でもその呼び方をしてきた。なお、「論理的」という言葉は図 5 がハードウェア上での配置をそのまま表しているのではなく、OS によって提供された仮想的なメモリの配置を表していることを意味している。

各領域の役割は以下の通りである。

- **テキスト領域**: 機械語に翻訳されたプログラムが格納される. この機械語の命令が 1 行づつ実行されることでプログラムが動く。
- **静的領域**: グローバル変数などの静的変数が置かれる。
- **ヒープ領域**: メモリの動的管理 (C 言語の malloc 関数や C++ の new 演算子でメモリを確保すること) で用いられる。
- **スタック領域**: 今回の演習で扱ったように CPU のレジスタを一時的に退避させたり、また C 言語の自動変数 (多くのローカル変数) が置かれる。

本演習でいままで用いたのは「テキスト領域」、「データ領域 のうちの静的領域」、そして今回取り扱った「スタック領域」である。

今回はリターンアドレス $ra を保存するためにスタック領域を用いたが、C 言語ではそれ以外にも**ローカル変数**に代表される**自動変数**をスタックに確保するところに大きな特徴がある。次節でもう詳しくそれを見てみよう。

B.2 スタックの使われ方の例

図 2 のプログラムを完成させた図 6 の C 言語プログラムを考えよう。既に解説したように、変数 b には a の 2 乗、すなわち 25 が格納されてプログラムが終了する。この時、main 関数や square 関数の

```
int square(int i); // 関数のプロトタイプ
int S; // グローバル変数

int main(void){ // main 関数
     int a, b;
     a=5;
     b=square(a);
     return(0); // プログラムの終了
}

int square(int i){ // square 関数の本体
     return(i*i);
}
```

図 6: 関数 square の利用。なお、S は定義しただけで使用していない。

外部で定義した変数 S は**グローバル変数**となり、プログラム中の全ての位置で利用することができる。一方、main 関数内の変数 a, b や square 関数の引

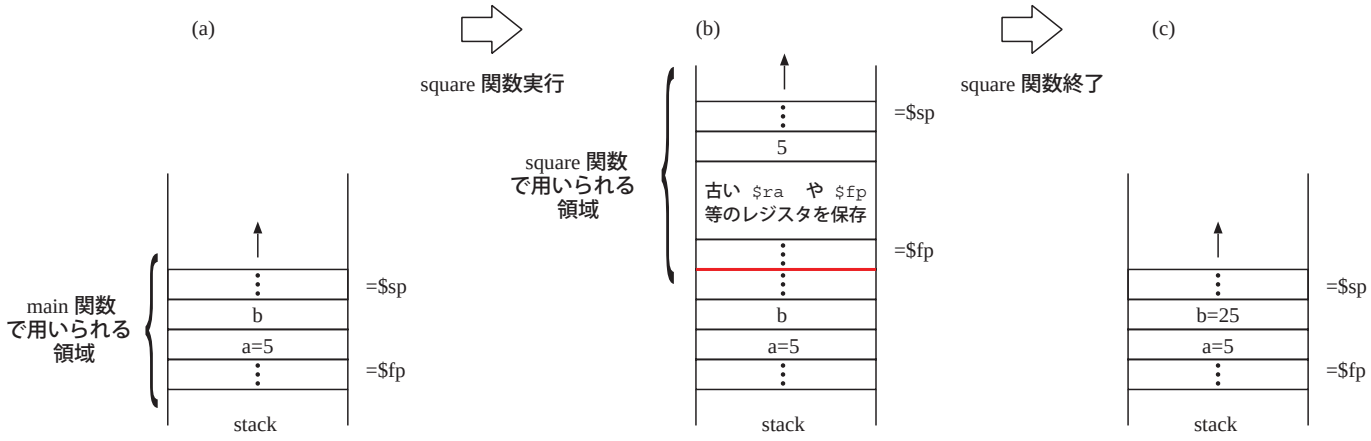

図 7: square 関数を用いた時のメモリのスタック領域の模式図。(a) square 実行前、(b) 実行中、(c) 実行後。

数 i はその関数内部でしか参照できない**ローカル変数**である。前節でふれた様に、グローバル変数 S はメモリの静的領域に確保され、変数 a, b, i はメモリのスタック領域に確保される。グローバル変数 S の取り扱いは本演習で今まで何度も取り扱って来たので解説は割愛し、ここでは変数 a, b, i のメモリ上での配置について概説しよう。図 7 に square 関数を用いた時のメモリのスタック領域の模式図を示した。

まず square 関数が実行される前には、main 関数で定義された変数 a, b はスタック領域に図 7(a) のように確保される。スタックポインタ $sp は演習で取扱ったように使用されているスタック領域の low address 側の境界を指し示している。このとき、教科書図 3.12 のようにフレームポインタ $fp はスタック領域の high address 側の境界を表す [2]。なお、図は教科書と本資料で上下が逆転していることに注意しよう。

square 関数が実行されている最中のスタック領域の模式図は図 7(b) である。本演習で扱ったように、リターンアドレス $ra が保存されるとともに、$fp などの他のレジスタも保存される。さらに square の引数である a の値、すなわち 5 も確保されていることがわかる。この 5 の 2 乗が CPU で計算されて戻り値を表すレジスタ $v0 に格納される。

square 関数が終了するとスタック領域は図 7(c) のようになり、戻り値 $v0 の値が変数 b に代入されてプログラムが終了する。

なお、第 3 章で記述してもらった関数 square は図 7 をそのままアセンブリ言語化しているわけではなく、その簡略版になっていることに注意しておく。

以上のように C 言語のプログラムを書く際、使用されるメモリの状態を想像できるようになれば、C 言語の初心者のレベルを卒業したと言えるのではないだろうか [3]。

参考文献

[1] spim のバージョンによってこのアドレスが異なる場合があるが、以下の議論は本質的に共通である。

[2] ただし、教科書 3.6 章の最後でふれられているように、$fp は必ずしも必要ではない。

[3] 最近は Java や C# のように「プログラマにメモリの状態を意識させない」プログラミング言語が主流になりつつあるが、工学部の学生であれば、ソフトウェアが動作するときのプロセッサやメモリの状態 (すなわちハードウェアの振る舞い) を知ることは必須と言えるのではないだろうか。