PAW Binutils Information

This document describes the binutils for PAW. Binutils are the utilities used to create and manipulate object files. These utilities are:

- paw-as - the PAW assembler
- paw-objdump - the PAW object file displayer
- paw-objcopy - the PAW object file translator
- paw-ld - the PAW linker

These utilities are based upon the GNU binutils, which are the standard binutils installed on the Nobel machines. Because they are standard, full documentation for the command line options and file formats is available through the 'info' command. Simply use commands like 'info as' or 'info ld' to see the information file for the utility. All of these utilities also offer command-line help through the --help option.

The object file format used by the binutils for PAW is a file format known as ELF. You do not need to know any of the details of this format, but you should be aware that this format stores not only the machine code and data for your program, but also the symbol table which the assembler generated. This allows the linker to link together separate object files.

Your simulator will not be required to read ELF files. Instead, it will read binary files. The paw-objcopy command can be used (as described below) to generate binary files from ELF files. To prepare to use the binutils, you must run the following lines on Nobel cluster (for sh/bash), or put them in your bash configuration file:

```
export PATH=/u/ee375/public/bin-linux:${PATH}
```

```
export MANPATH=/u/ee375/public/man:${MANPATH}
```

The commands for using binutils are now described:

`paw-as [option...] [input_file] [-o output_file]`

This is a full-featured assembler. It is able to define macros, handle expressions, and insert data. The directives you need to know the most about are:

- `.text, .data, .bss`

  These directives specify which "section" the following instructions and data are going to go into. Instructions and read-only data should go into the text section. Variables which may change, but have an initial value go into the data section. Variables which do not need to be initialized go into the bss section.

- `.word, .long, .short, .byte <list of constants>`

  These directives put constants of the specified size (4, 4, 2, 1 bytes respectively) into the program. Labels can be specified as values, i.e:

  ```
  .word some_label
  ```

- `.align <number>`

  This directive causes the assembler to skip bytes (filling them with 0 by default) in the program until the lowest <number> of bits of the address are zero. For example:

  ```
  .align 2
  ```

  skips bytes until the address is divisible by 4. You should use such an alignment directive before the first of a series of .word or .data directives so that the addresses are word-aligned.

- `globl <label>`

  This directive marks the label's entry in the symbol table as "globally visible". This means that the linker can use this symbol when resolving external references in other object files being linked with the one made from this file.

There is one pseudo-operation which you may wish to use. This operation is:

```
adr Rd, label
```

This pseudo-op expands into:

```
add pc, #(label - . - 2) & 0xfffc
```

This rather curious expression is used to deal with tables of constants. You may have noticed that it is difficult in PAW to form constants larger than 255 or negative constants. This is true of Thumb as well. The normal way of overcoming this difficulty is to create a table of constants and then load the constant into a register. Thus, if you needed large constants, you might use code similar to the following:

```
        adr r6, constants

        ldr r5, [r6, #0]  @ constant 1

        ldr r4, [r6, #4]  @ constant 2

        ....

        .align 2 @ so the word alignment is good

    constants:

        .word 0x3456 @ constant 1

        .word 0xfe00 @ constant 2
```

Note that the constant table must come **after** the instruction which loads its address and must come within 510 instructions of it. Constant tables must remain in the text section and must be in the same source file. The constants themselves must be evaluable at assembly or link time. (Just FYI, these constant tables are often called "literal pools".)

Important note: if you use adr, you must link your code to generate the correct address offset to the constant table.

`paw-objdump  [options] -d file_name`

This command disassembles the instructions in the text section of a file.

Please note that r10, r11, r12, r13, r14, and r15 will be called sl, fp, ip, sp, lp, and pc in the output unless you use the -Mreg-names-raw option. Also, the -Mreg-names-std option will give you the "ARM standard" register names where only sp, lp, and pc (r13, r14, r15) are given special names.

Also, all of the format 12 (load address) instructions will also output what they would look like as an adr pseudo-operation.

If you're getting out instructions which look like ARM instructions, try using -Mforce-thumb. This option should not be necessary if you have put a label before the first instruction in your source file.

`paw-objdump  [options] -t file_name`

This command lists the symbol table of a file.

`paw-objcopy -O binary <elf_file> <binary_file>`

This command takes an ELF file as an input and produces a binary file as the output.

`paw-ld [options] <input_files> [-o output file]`

This command links multiple ELF files into one ELF executable. You should use the following options for the simulator assignment:

```
     -Ttext 0 --entry 0
```

A final note

When writing your test programs, using a few conventions will make it much less likely that the tools will break down on you. These conventions are:

1. Always have a label before the first instruction of any source file

2. Always make the "entry point" - the first instruction of a program - the first instruction in some source file and make certain you list the corresponding object file first when linking.