A manual for the Plan 9 assembler

Rob Pike

Disclaimer

I wrote this manual to document some of the assembler's strange properties but I am not responsible for any of them.

Machines

There is an assembler for each of the MIPS 3000, SPARC, Intel 386, Intel 960, AT&T Hobbit, and Motorola 68020. The 68020 assembler, `2a`, is the oldest and in many ways the prototype. To keep things concrete, the first part of this manual is specifically about the 68020. At the end is a brief description of the differences among the other assemblers.

Registers

All pre-defined symbols in the assembler are upper-case. Data registers are `R0` through `R7`; address registers are `A0` through `A7`; floating-point registers are `F0` through `F7`.

A pointer in `A6` is used by the C compiler to point to data, enabling short addresses to be used more often. The value of `A6` is constant and must be set during C program initialization to the address of the externally-defined symbol `a6base`.

The following hardware registers are defined in the assembler; their meaning should be obvious given a 68020 manual: `CAAR`, `CACR`, `CCR`, `DFC`, `ISP`, `MSP`, `SFC`, `SR`, `USP` and `VBR`.

The assembler also defines several pseudo-registers that manipulate the stack: `FP`, `SP`, and `TOS`. `FP` is the frame pointer, so `0(FP)` is the first argument, `4(FP)` is the second, and so on. `SP` is the local stack pointer, where automatic variables are held; `0(SP)` is the first automatic, and so on as with `FP`. Finally, `TOS` is the top-of-stack register, used for pushing parameters to procedures, saving temporary values, and so on.

The assembler and loader track these pseudo-registers so the above statements are true regardless of what has been pushed on the hardware stack, pointed to by `A7`. The name `A7` refers to the hardware stack pointer, but beware of mixed use of `A7` and the above stack-related pseudo-registers, which will cause trouble. Note, too, that the `PEA` instruction is observed by the loader to alter SP and thus will insert a corresponding pop before all returns. The assembler accepts a label-like name to be attached to `FP` and `SP` uses, such as `p+0(FP)` to help document that `p` is the first argument to a routine. The name goes in the symbol table but has no significance to the result of the program.

Referring to data

All external references must be made relative to some pseudo-register, either `PC` (the virtual program counter) or `SB` (the ‘‘static base’’ register). `PC` counts instructions, not bytes of data. For example, to branch to the second following instruction, that is, to skip one instruction, one may write

```
	BRA	2(PC)
```

Labels are also allowed, as in

```
        BRA     return
        NOP
return:
        RTS
```

When using labels, there is no `(PC)` annotation.

The pseudo-register `SB` refers to the beginning of the address space of the program. Thus, references to global data and procedures are written as offsets to `SB`, as in

```
        MOVL    $array(SB), TOS
```

to push the address of a global array on the stack, or

```
        MOVL    array+4(SB), TOS
```

to push the second (4-byte) element of the array. Note the use of an offset; the complete list of addressing modes is given below. Similarly, subroutine calls must use `SB`:

```
        BSR     exit(SB)
```

File-static variables have syntax

```
        local<>+4(SB)
```

The <> will be filled in at load time by a unique integer.

When a program starts, it must execute

```
        MOVL    $a6base(SB), A6
```

before accessing any global data. (On machines such as the MIPS and SPARC that cannot load a register in a single instruction, constants are loaded through the static base register. The loader recognizes code that initializes the static base register and treats it specially. You must be careful, however, not to load large constants on such machines when the static base register is not set up, such as early in interrupt routines.)

Expressions

Expressions are mostly what one might expect. Where an offset or a constant is expected, a primary expression with unary operators is allowed. A general C constant expression is allowed in parentheses.

Source files are preprocessed exactly as in the C compiler, so `#define` and `#include` work.

Laying down data

Placing data in the instruction stream, say for interrupt vectors, is easy: the pseudo-instructions `LONG` and `WORD` (but not `BYTE`) lay down the value of their single argument, of the appropriate size, as if it were an instruction:

```
        LONG    $12345
```

places the long 12345 (base 10) in the instruction stream. (On all machines except the 68020, the only such operator is `WORD` and it lays down 32-bit quantities.)

Placing information in the data section is more painful. The pseudo-instruction `DATA` does the work, given two arguments: an address at which to place the item, including its size, and the value to place there. For example, to define a character array `array` containing the characters `abc` and a terminating null:

```
        DATA    array+0(SB)/1, $'a'
        DATA    array+1(SB)/1, $'b'
        DATA    array+2(SB)/1, $'c'
        GLOBL   array(SB), $4
```

Here, the `/1` defines the number of bytes to define, `GLOBL` makes the symbol global, and the `$4` says how many bytes the symbol occupies. Uninitialized data is zeroed automatically.

Defining a procedure

Entry points are defined by the pseudo-operation `TEXT`, which takes as arguments the name of the procedure (including the ubiquitous `(SB)`) and the number of bytes of automatic storage to pre-allocate on the stack, which will usually be zero when writing assembly language programs. Here is a complete procedure that returns the sum of its two arguments:

```
TEXT    sum(SB), $0
        MOVL    arg1+0(FP), R0
        ADDL    arg2+4(FP), R0
        RTS
```

An optional middle argument to the `TEXT` pseudo-op causes the loader *not* to insert profiling code into the subroutine, if it has the value `1`. For example,

```
TEXT    sum(SB), 1, $0
        MOVL    arg1+0(FP), R0
        ADDL    arg2+4(FP), R0
        RTS
```

will not be profiled; the first version above would be. Subroutines with peculiar state, such as system call routines, should not be profiled.

Subroutines to be called from C should place their result in `R0`, even if it is an address. Floating point values are returned in `F0`. Functions that return a structure to a C program receive as their first argument the address of the location to store the result; `R0` is unused in the calling protocol for such procedures. A subroutine is responsible for saving its own registers, and therefore is free to use any registers without saving them (‘‘caller saves’’). `A6` and `A7` are the exceptions as described above.

When in doubt

If you get confused, try using the `-S` option to `2c` and compiling a sample program. The standard output is valid input to the assembler.

Instructions

The instruction set of the assembler is not identical to that of the machine. It is chosen to match what the compiler generates, augmented slightly by specific needs of the operating system. For example, `2a` does not distinguish between the various forms of `MOVE` instruction: move quick, move address, etc. Instead the context does the job. For example,

```
        MOVL    $1, R1
        MOVL    A0, R2
        MOVW    SR, R3
```

generates official `MOVEQ`, `MOVEA`, and `MOVESR` instructions. A number of instructions do not have the syntax necessary to specify their entire capabilities. Notable examples are the bitfield instructions, the multiply and divide instructions, etc. For a complete set of generated instruction names (in `2a` notation, not Motorola’s) see the file `/sys/src/cmd/2c/2.out.h`. Despite its name, this file contains an enumeration of the instructions that appear in the intermediate files generated by the compiler, which correspond exactly to lines of assembly language.

Addressing modes

In this table, `o` is an offset, which if zero may be elided. Many of the modes listed have the same name; scrutiny of the format will show what default is being applied. For instance, indexed mode with no address register supplied operates as though a zero-valued register were used. For "offset" read "displacement." For "`.s`" read one of `.L`, or `.W` followed by `*1`, `*2`, `*4`, or `*8` to indicate the size and scaling of the data.

data register	`R0`
address register	`A0`

floating-point register	`F0`
special names	`CAAR, CACR, etc.`
constant	`$con`
floating point constant	`$fcon`
external symbol	`name+o(SB)`
local symbol	`name<>+o(SB)`
automatic symbol	`name+o(SP)`
argument	`name+o(FP)`
address of external	`$name+o(SB)`
address of local	`$name<>+o(SB)`
indirect post-increment	`(A0)+`
indirect pre-decrement	`-(A0)`
indirect with offset	`o(A0)`
indexed with offset	`o()(R0.s)`
indexed with offset	`o(A0)(R0.s)`
external indexed	`name+o(SB)(R0.s)`
local indexed	`name<>+o(SB)(R0.s)`
automatic indexed	`name+o(SP)(R0.s)`
parameter indexed	`name+o(FP)(R0.s)`
offset indirect post-indexed	`d(o())(R0.s)`
offset indirect post-indexed	`d(o(A0))(R0.s)`
external indirect post-indexed	`d(name+o(SB))(R0.s)`
local indirect post-indexed	`d(name<>+o(SB))(R0.s)`
automatic indirect post-indexed	`d(name+o(SP))(R0.s)`
parameter indirect post-indexed	`d(name+o(FP))(R0.s)`
offset indirect pre-indexed	`d(o()(R0.s))`
offset indirect pre-indexed	`d(o(A0))`
offset indirect pre-indexed	`d(o(A0)(R0.s))`
external indirect pre-indexed	`d(name+o(SB))`
external indirect pre-indexed	`d(name+o(SB)(R0.s))`
local indirect pre-indexed	`d(name<>+o(SB))`
local indirect pre-indexed	`d(name<>+o(SB)(R0.s))`
automatic indirect pre-indexed	`d(name+o(SP))`
automatic indirect pre-indexed	`d(name+o(SP)(R0.s))`
parameter indirect pre-indexed	`d(name+o(FP))`
parameter indirect pre-indexed	`d(name+o(FP)(R0.s))`

MIPS

The registers are only addressed by number: `R0` through `R31`. `R29` is the stack pointer; `R30` is used as the static base pointer, the analogue of `A6` on the 68020. Its value is the address of the global symbol `setR30(SB)`. The register holding returned values from subroutines is `R1`. When a function is called, space for the first argument is reserved at `0(FP)` but the value is actually passed in `R1`.

The loader uses `R28` as a temporary. The system uses `R26` and `R27` as interrupt-time temporaries. None of these registers should therefore be used in user code.

The control registers are not known to the assembler. Instead they are numbered registers `M0`, `M1`, etc. Use this trick to access, say, `TLBVIRT`:

```
#define TLBVIRT 10
        MOVW    M(TLBVIRT), R1
```

Floating point registers are called `F0` through `F31`. By convention, `F24` must be initialized to the value 0.0, `F26` to 0.5, `F28` to 1.0, and `F30` to 2.0.

The instructions are quite different from the book. There are no `lui` and kin; instead there are `MOVW` (move word), `MOVH` (move halfword), and `MOVB` (move byte) pseudo-instructions. If the operand is unsigned, the instructions are MOVHU and MOVBU. The order of operands is from left to right in dataflow order, just as on the 68020 but not as in the book. This means that the Bcond instructions are reversed with respect to the book; for example, a va BGTZ generates a MIPS bltz instruction.

Instruction scheduling is handled by the loader. Place the instructions down as you would if there were no such things as delay slots, except that, because the information to solve the problem is not published, you must insert your own NOOPS after instructions that modify control registers. Use exactly

```
NOR     R0, R0, R0
```

as a no-op.

SPARC

Once you understand the Plan 9 model for the MIPS, the SPARC is familiar. Registers have numerical names only: R0 through R31. Forget about register windows. Plan 9 doesn't use them at all. The machine has 32 global registers, period. R1 (sic) is the stack pointer. R2 is the static base register, with value the address of setSB(SB). R7 is the return register and also the register holding the first argument to a function, again with space reserved at 0(FP). R14 is the loader temporary.

Floating-point registers are exactly as on the MIPS.

The control registers are known by names such as FSR. The instructions to access these registers are MOVW instructions, for example

```
MOVW    Y, R8
```

for the Sparc instruction

```
rdy     %r8
```

Move instructions are similar to those on the MIPS: pseudo-operations that turn into appropriate sequences of sethi instructions, adds, etc. Instructions read from left to right. Because the arguments are flipped to SUBCC, the condition codes are not inverted as on the MIPS.

The syntax for the ASI stuff is, for example to move a word from ASI 2:

```
MOVW    (R7, 2), R8
```

The syntax for double indexing is

```
MOVW    (R7+R8), R9
```

Again, ignore scheduling and delay slots when laying down instructions; the loader will do the scheduling.

i960

Registers are numbered R0 through R31. Stack pointer is R29; return register is R4; static base is R28; it is initialized to the address of setSB(SB). R3 must be zero; this should be done manually early in execution by

```
SUBO    R3, R3
```

R27 is the loader temporary.

There is no support for floating point.

The Intel calling convention is not supported and cannot be used; use BAL instead. Instructions are mostly as in the book. The major change is that LOAD and STORE are both called MOV. The extension character for MOV is as in the manual: O for ordinal, W for signed, etc.

i386

The assembler assumes 32-bit protected mode. The register names are `SP`, `AX`, `BX`, `CX`, `DX`, `BP`, `DI`, and `SI`. The stack pointer is `SP` and the return register is `AX`. There is no physical frame pointer but, as for the 68020, `FP` is a pseudo-register that acts as a floating frame pointer.

Opcode names are mostly the same as those listed in the Intel manual with an `L`, `W`, or `B` appended to identify 32-bit, 16-bit, and 8-bit operations. The exceptions are loads, stores, and conditionals. All load and store opcodes to and from general registers, special registers (such as `CR0, CR3, GDTR, IDTR, SS, CS, DS, ES, FS,` and `GS`) or memory are written as

```
MOVx    src,dst
```

where *x* is `L`, `W`, or `B`. Thus to get `AL` use a `MOVB` instruction. If you need to access `AH`, you must mention it explicitly in a `MOVB`:

```
MOVB    AH, BX
```

There are many examples of illegal moves, for example

```
MOVB    BP, DI
```

that the loader actually implements as psuedo-operations.

The names of conditions in all conditional instructions (`J`, `SET`) follow the conventions of the 68020 instead of those of the Intel assembler: `JOS`, `JOC`, `JCS`, `JCC`, `JEQ`, `JNE`, `JLS`, `JHI`, `JMI`, `JPL`, `JPS`, `JPC`, `JLT`, `JGE`, `JLE`, and `JGT` instead of `JO`, `JNO`, `JB`, `JNB`, `JZ`, `JNZ`, `JBE`, `JNBE`, `JS`, `JNS`, `JP`, `JNP`, `JL`, `JNL`, `JLE`, and `JNLE`.

The addressing modes have syntax like `AX`, `(AX)`, `(AX)(BX*4)`, `10(AX)`, and `10(AX)(BX*4)`. The offsets from `AX` can be replaced by offsets from `FP` or `SB` to access names, for example `extern+5(SB)(AX*2)`.

Other notes: Non-relative `JMP` and `CALL` have a `*` added to the syntax. Only `LOOP`, `LOOPEQ`, and `LOOPNE` are legal loop instructions. Only `REP` and `REPN` are recognized repeaters. These are not prefixes, but rather standalone opcodes that precede the strings, for example

```
CLD; REP; MOVSL
```

Hobbit

The Hobbit book and data sheet are actually based on this assembler, so the correspondence between the hardware and the assembler is much closer than with the other machines. There is no static base register.

The one syntactic difference is that data operand suffices use a colon in the book, but a period in the assembler:

```
MOV     $0, R4.UB
```

Unlike the other assemblers, these suffices are applied to the operands rather than the instructions.