Systematic Compiler Construction

Michael Sperber Peter Thiemann

January 26, 2004

Dieses Werk ist urheberrechtlich geschützt; alle Rechte mit Ausnahme der folgenden sind vorbehalten:

Studenten der Informatik dürfen diese Dokument für ihre persönlichen Lehrzwecke verwenden.

Ausdrücklich verboten wird jede andere Verwendung von Ausdrucken, Dateikopien oder Paperkopien. Insbesondere darf dieses Skriptum nicht in irgendeiner Weise vetrieben werden und es dürfen keine Kopien der Dateien auf öffentlich zugänglichen Servern angelegt werden.

Copyright © Michael Sperber, 1999; Peter Thiemann, 2000, 2002

Chapter 5

The IBM POWER Architecture

The time has come to turn to actual machines in order to see what comes next in compilation. Most traditional compiler texts use imaginary processor architectures for that purpose to avoid the quirks and conventions associated with real-world microprocessors. However, compiling to a virtual processor gives few of the satisfactions associated with writing a compiler. Therefore, this chapter uses the IBM POWER/PowerPC architecture as a typical representative for modern RISC processor architectures. For now, the only concern is to understand the design principles underlying the architecture without concrete compilation issues in mind. As will become clear in the next chapter, however, the CPS representation of the last chapter converts neatly into real machine code.

5.1 Instruction Set Overview

IBM evolved the RS/6000 POWER architecture into the current PowerPC. The PowerPC mostly generalizes the older POWER chips. Therefore, the overlap is substantial, and the material presented here is common to both chips. Therefore, only the term "POWER" will be used.

The POWER architecture is a 32-bit RISC machine which has four separate functional units:

- the branch processor,
- the fixed-point processor,
- the floating-point processor,
- and the storage-control processor.

Each of these functional units executes a certain set of instructions and manages its own set of registers. Some are shared between several functional units, and special instructions move data between the register sets. Figure 5.1 shows the complete register set. Note that IBM in its documentation numbers bits in the exact opposite way as most other literature: Bit 0 is the most-significant bit, and the significance decreases with increasing bit indices. This chapter sticks with IBM's convention.

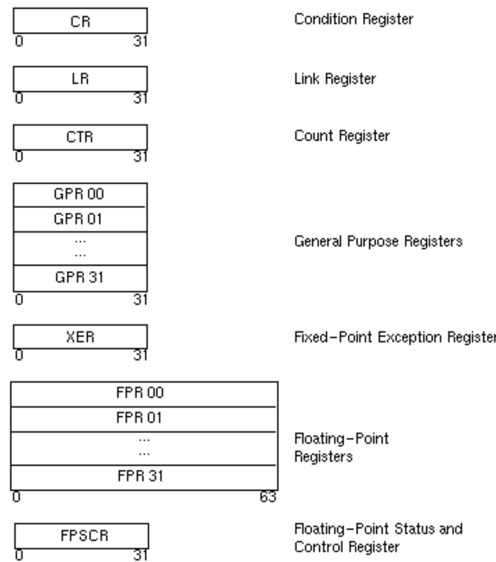

Figure 5.1: The POWER register set

5.1.1 The branch processor

is responsible for executing branches (conditional and unconditional), procedure calls, system calls, as well as certain logical operations. In addition to the condition register, the branch processor also manages the Link and Count registers.

The destination of a branch instruction can be one of the following:

- the sum of a constant and the address of the branch instruction itself,
- the absolute address given as an operand to the instruction,
- the content of the Link register, or
- the content of the Count register.

Certain branches store a return address into the link register, thereby supporting a form of procedure call. The Count register can serve as a loop count, and certain branch instructions implicitly decrement it and test if its value is zero.

The condition register is somewhat unusual compared to its counterpart in more traditional architectures: It consists of eight independent 4-bit condition fields with the following meanings for fixed-point instructions:

Bit 0 less than

Bit 1 greater than

Bit 2 equal

Bit 3 summary overflow

The summary overflow is irrelevant for the purposes of this chapter.

Each field of the condition register can be the target of selected fixed-point and floating-point instructions and can control branching. However, some instructions by default access fixed fields of the condition registers. It is possible to copy condition fields onto each other, thereby allowing the code to use secondary fields for backups.

```
bc    4,1,L38 # jump relative to L38 if not greater than in field 0
```

(In the instruction, 4 is the value of the "branch option field" which stands for "Branch if condition is false." The 1 is the number of the CR bit referenced. The `bc` instruction always refers to CR field 0.)

5.1.2 The fixed-point processor

The fixed-point processor manages the 32 32-bit *general-purpose registers* (GPRs) as well as a 32-bit fixed-point exception register. It implements loads (moves from memory *into* registers) and stores (moves *from* registers into memory), arithmetic, logical, compare, shift, rotate, trap, and system-control instructions. Some of its operations perform quite complex tasks and are reminiscent of older CISC architectures. The instruction format varies widely from instruction to instruction. Some instructions come in two variants, one of which has a mnemonic with a trailing dot . which indicates that the instruction sets the condition register field 0 according to the result of the operation.

Load and store The load instructions move information from a memory location into one of the GPRs. The store instructions do the reverse. Load and store instructions exist for bytes, halfwords, and words, but can only access word-aligned locations in memory. A load or store instruction denotes the memory location by an *effective address* which is either the contents of a GPR, the sum of the contents of a GPR and a signed 16-bit offset or the sum of the contents of two GPRs. Occasionally, GPR 0, when part of an effective address computation, actually denotes the number 0 rather than the contents of GPR 0.

```
lwz 6,16(5)  # [GPR 6] := [16 + [GPR 5]]
lhzx 6,5,4   # [GPR 6] := [[GPR 5] + [GPR 4]],
             # [GPR 6]_{0-15} := 0
stw 6,16(5)  # [16 + [GPR 5]] := [GPR 6]
```

Load and store with update Load and store instructions have an "update" form in which the base GPR is updated with the effective address in addition to the regular move of information from or to memory.

```
lwzu 6,16(5) # [GPR 6] := [16 + [GPR 5]], [GPR 5] := [GPR 5] + 16
stwu 6,16(5) # [16 + [GPR 5]] := [GPR 6], [GPR 5] := [GPR 5] + 16
```

String moves The string instructions allow the movement of data from storage to registers or from registers to storage regardless of alignment.

```
lswi 6,5,4  # [GPR 6] := <4 consecutive bytes at [GPR 5]>
stwi 6,5,4  # [[GPR 5]] := <4 consecutive bytes in [GPR 6]>
```

Arithmetic The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

```
add 4,5,6   # [GPR 4] := [GPR 5] + [GPR 6]
add. 4,5,6  # [GPR 4] := [GPR 5] + [GPR 6], set CR0
```

Comparisons The compare instructions algebraically or logically compare the contents of a register with either a 16-bit immediate constant (signed or unsigned) or the contents of another register.

Algebraic comparison compares two signed integers. Logical comparison compares two unsigned integers.

```
cmp 1,0,4,5 # set CR1 according to [GPR 4] <cmp-algebraic> [GPR 5]
cmpl 1,0,4,5 # set CR1 according to [GPR 4] <cmp-logical> [GPR 5]
```

(The 0 field is a mandatory value without any special meaning.)

Logical operations Logical instructions perform bitwise logical operations.

```
andc 6,4,5 # [GPR 6] := [GPR 4] and ~[GPR 5]
```

Rotate and shift The rotate and shift operations work on the contents of a GPR in one of the following ways:

- The result of the rotation is inserted into the target register under the control of a mask. If the mask bit is 1, the associated bit of the rotated data is placed in the target register. If the mask bit is 0, the associated data bit in the target register is unchanged.
- The result of the rotation is ANDed with the mask before being placed into the target register.

The shift instructions logically perform left and right shifts. The result of a shift instruction is placed in the target register under the control of a generated mask.

```
rlmi 6,4,5,0,23 # [GPR 6] := [GPR 4] <rotate-left> [GPR 5]_{27-31}
                # ... but only bits 0-23
                # (rotate left then mask insert)
slw 6,4,5 # [GPR 6] := ([GPR 4] << [GPR 5]_{27-31}) & <mask>
```

(The `<mask>` in the `slw` instructions is controlled by bit 26 of GPR 5 through quite complicated rules.)

Move to/from special-purpose registers Several instructions move the contents of one Special-Purpose Register (SPR) into another SPR or into a GPR.

```
mfspr 6,1 # [GPR 6] := [EXC]
```

5.1.3 Extended Mnemonics

Many POWER instructions are excessively general, often hiding simple and common tasks behind complex operations. Because of this, IBM's assembler provides a large set of so-called *extended mnemonics* that are synonymous for special cases of complex statements. Some examples:

```
nop      # no-op, same as ori 0,0,0
mr 5,6   # (GPR 6) := (GPR 5), same as or 5,6,6
li 5,17  # (GPR 5) := 17, same as addi 5,0,17
```

5.2 Assembler Basics

An assembler source file consists of the following components:

Instructions consist of a mnemonic and, possibly, operands. An instruction corresponds to a machine instruction, and directs the assembler to generate the corresponding executable code.

Labels are names that mark locations in the generated code.

Pseudo Operations are directives to the assembler independent of code generation per se.

An assembly source file consists of lines, each of which contains an instruction or a pseudo operation, optionally marked by a label:

⟨line⟩ ::= ⟨label⟩$^?$ ⟨statement⟩ ⟨comment⟩$^?$
⟨statement⟩ ::= ⟨empty⟩
  | ⟨mnemonic⟩ ⟨operands⟩$^?$
  | `.` ⟨pseudo-op⟩ ⟨pseudo-operands⟩$^?$
⟨label⟩ ::= ⟨symbol⟩ `:`
⟨operands⟩ ::= ⟨operand⟩ ⟨another-operand⟩*
⟨another-operand⟩ ::= `,` ⟨operand⟩
⟨comment⟩ ::= `#` ⟨char⟩*
⟨pseudo-operands⟩ ::= ⟨pseudo-operand⟩ ⟨another-pseudo-operand⟩*
⟨another-pseudo-operand⟩ ::= `,` ⟨pseudo-operand⟩

As the assembler advances through a source file, it generates code and keeps a *location counter* marking the address of the code generated. The symbol `$` refers to the value of the location counter. Also, certain pseudo operations change the location counter directly without generating any code.

5.2.1 Pseudo Operations

Here are some common pseudo operations:

⟨statement⟩ ::= `.set` ⟨symbol⟩ `,` ⟨expression⟩

makes ⟨symbol⟩ a synonym for the value of ⟨expression⟩.

```
.set ap,14
lil ap,2    # assembler substitutes value 14 for the symbol
            # Note that ap is a register number as lil's operand.
```

⟨statement⟩ ::= `.byte` ⟨expressions⟩$^?$
⟨expressions⟩ ::= ⟨expression⟩ ⟨another-expression⟩*
⟨another-expression⟩ ::= `,` ⟨expression⟩

generates consecutive bytes corresponding to the values of the parameters of `.byte`.

```
.byte 0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x20,0x77,0x6F,0x72,0x6C,0x64
.byte "Hello, world"
.byte 'H,'e,'l,'l,'o,',,' ,'w,'o,'r,'l,'d
```

There are corresponding pseudo operations `.double`, `.float`, `.long`, `.short`, `.string`, and `.vbyte`.

⟨statement⟩ ::= `.align` ⟨expression⟩

advances the current location counter until a boundary specified by the parameter is reached. The parameter values correspond to the following alignments:

0 byte

1 halfword (16 bits)

2 word

3 doubleword

```
.byte   1 # Location counter now at odd number
.align  1 # Location counter is now at the next halfword boundary.
.byte   3,4
```

5.2.2 AIX conventions

Knowing the machine language and assembler syntax of an architecture is not enough to be able to write running programs: The operating system and runtime environment impose a number of constraints and conventions on assembler code. In the case of AIX, these conventions have to do with the subdivision of a program into sections and procedure calling conventions.

Programming with the TOC An AIX executable consists of *sections* that contain different parts of the running program. The important sections are called `.text`, `.data`, and `.bss`:

.text contains code or read-only data.

.data contains read-write data.

.bss contains uninitialized mapped data.

Each section consists of subsections called *csects*. A csect is assigned to a *storage mapping class* further describing the role of a csect. The set of storage mapping classes is fixed, and each class has a two-letter name. The storage mapping class also determines a certain section. For example, `PR` is for executable code residing in the `.text` section, and `RO` is for read-only data also residing in `.text`

The code for each code-generating statement of an assembler source file is assigned to a specific csect, and assembler and linker together take care to assemble the various csects into consecutive chunks of code.

Code can be assigned to a csect with the `.csect` directive. For example,

```
.csect bla[pr]
```

announces that subsequent code will be put into a csect named `bla[pr]` with (obviously) storage mapping class `PR`.

Now, since instruction opcodes are all exactly 32 bits in size, they cannot contain a full absolute memory reference. Instead, memory access needs to be indirect. Therefore, AIX provides access to absolute memory locations via a *table of contents*, or *TOC* accessible through a GPR. The TOC is simply a table of indirections

accessible globally in a program. It can contain data directly or pointers to csects. Only the latter case is of interest here.

TOC entries have special storage mapping classes. A TOC entry with storage mapping class `TC` contains the address of a csect or a global symbol. The `.toc` directive announces that subsequent code goes into the TOC, and the `.tc` directive creates TOC entries. As a convention, AIX programs keep a pointer to the TOC in general-purpose register 2. Here is a usage example:

```
        .set    RTOC,2
        .csect prog1[pr]
        ...
        l  5,TCA(RTOC)          # [GPR5] := a[rw]
        ...
        .toc
TCA:    .tc  a[tc],a[rw]        # csect a[rw] goes into TOC name a[tc]

        .csect a[rw]
        .long 25
```

Calling Conventions AIX also specifies a set of *calling conventions* meant to ease interoperability between different programming languages. It is not the main concern of the compilers here, but since all programs run in the same environment, they must at least outwardly observe these conventions.

Register	Status	Use
GPR0	volatile	In function prologs.
GPR1	dedicated	Stack pointer.
GPR2	dedicated	Table of Contents (TOC) pointer.
GPR3	volatile	First word of a function's argument list; first word of a scalar function return.
GPR4	volatile	Second word of a function's argument list; second word of a scalar function return.
GPR5	volatile	Third word of a function's argument list.
GPR6	volatile	Fourth word of a function's argument list.
GPR7	volatile	Fifth word of a function's argument list.
GPR8	volatile	Sixth word of a function's argument list.
GPR9	volatile	Seventh word of a function's argument list.
GPR10	volatile	Eighth word of a function's argument list.
GPR11	volatile	In calls by pointer and as an environment pointer for languages that require it (for example, PASCAL).
GPR12	volatile	For special exception handling required by certain languages and in glink code.
GPR13:GPR31	nonvolatile	These registers must be preserved across a function call.

Figure 5.2: General Purpose Register Conventions

The preferred method of using GPRs is to use the volatile registers first. Next, use the nonvolatile registers in descending order, starting with GPR31 and proceeding down to GPR13. GPR1 and GPR2 must be dedicated as stack and Table of Contents (TOC) area pointers, respectively. GPR1 and GPR2 must appear to be

saved across a call, and must have the same values at return as when the call was made.

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Boc76] G. V. Bochmann. Semantic evaluation from left to right. *Communications of the ACM*, 19:55–62, 1976.

[Cha87] Nigel P. Chapman. *LR parsing: theory and practice*. Cambridge University Press, Cambridge, UK, 1987.

[DeR69] Franklin L. DeRemer. *Practical Translators for LR(k) Parsers*. PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Ma., 1969.

[DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, 1971.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.

[DP82] Franklin DeRemer and Thomas Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, October 1982.

[DS95] Charles Donnelly and Richard Stallman. *Bison—The YACC-compatible Parser Generator*. Free Software Foundation, Boston, MA, November 1995. Part of the Bison distribution.

[FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.

[FHP92] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code generator generator. *Letters on Programming Languages and Systems*, 1(3):213–226, 1992.

[Fis93] Michael J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3/4):259–288, 1993.

[Ive86] Fred Ives. Unifying view of recent LALR(1) lookahead set algorithms. *SIGPLAN Notices*, 21(7):131–135, July 1986. Proceedings of the SIGPLAN'86 Symposium on Compiler Construction.

[Ive87a] Fred Ives. An LALR(1) lookahead set algorithm. Unpublished manuscript, 1987.

[Ive87b] Fred Ives. Response to remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(8):99–104, August 1987.

[Joh75] S. C. Johnson. Yacc—yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[Lee93] Rene Leermakers. *The Functional Treatment of Parsing*. Kluwer Academic Publishers, Boston, 1993.

[PC87] Joseph C.H. Park and Kwang-Moo Choe. Remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(4):30–32, April 1987.

[PCC85] Joseph C. H. Park, K. M. Choe, and C. H. Chang. A new analysis of LALR formalisms. *ACM Transactions on Programming Languages and Systems*, 7(1):159–175, January 1985.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Spe94] Michael Sperber. Attribute-directed functional LR parsing. Unpublished manuscript, October 1994.

[SSS90] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory*, volume II (LR(k) and LL(k) Parsing) of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1990.

[ST95] Michael Sperber and Peter Thiemann. The essence of LR parsing. In William Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 146–155, La Jolla, CA, June 1995. ACM Press.

[ST00] Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(2):224–264, March 2000.

[WM92] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau — Theorie, Konstruktion, Generierung*. Lehrbuch. Springer-Verlag, Berlin, Heidelberg, 1992.