Computer Science 237
Computer Organization
Williams College
Fall 2005

The WC34000 Computer

The WC34000 is a hypothetical machine designed to resemble the MC68000 but simplified in ways intended to make it simpler to work with in various course projects. The main simplification is that while the MC68000 can manipulate byte, word or long word operands, the WC34000 only operates on word length operands. This simplifies the descriptions of the behavior of many of the machines instructions. More importantly, by eliminating the need for the operand length specifications included in many MC68000 instructions, this change relieves the squeeze on space in the operation field and makes it possible to greatly simplify the instruction set encoding. This handout describes the architecture of the WC34000.

1 Data Organization

The basic unit of data manipulated by the WC34000 is a 16 bit word. The machine's memory is word-addressable (rather than byte-addressable as in the 68000). Up to 65,536 words of memory may be addressed.

The processor includes eight 16 bit data registers and eight 16 bit address registers. The data registers are referred to as D0, D1, ... D7. The address registers are referred to as A0, A1, ... A7. Several instructions reference A7 implicitly as a stack pointer. Accordingly, it can also be referenced by the name SP.

In addition, the processor includes a program counter register (PC) and a condition code register (CCR). The CCR in the WC34000 contains bits named N (negative), and Z (zero). These bits are changed only by the CMP (compare) instruction. They are tested by the conditional branch instructions.

2 Instruction Encoding

WC34000 instructions are from one to three words in length. The first word of each instruction is called the *operation field*. The contents of this word determine the operation to be performed and the *addressing modes* to be used to locate the operands. Some of the addressing modes require the specification of more information than can be fit into the operation field. When such modes are used, this information is placed in *extension words* that immediately follow the operation field in memory.

2.1 Effective Address Specifications

Nearly all operands for WC34000 instructions are specified using a 6-bit *effective address specification*. Each effective address specification is composed of two 3-bit sub-fields — the mode field and the register field — shown below. The value in the mode field determines which of the ma-

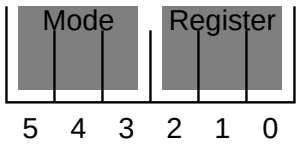


Figure 1: Effective address specification format

chines addressing modes is to be used to locate the operand. The register field contains the number of a register whose value is either the operand itself or is to be used to locate the operand.

2.2 Addressing modes

The addressing modes that can be selected using the mode field of an effective address specification can be grouped into three categories: register direct, memory addressing, and special.

2.2.1 Register Direct modes

These addressing modes specify that the operand is one of the machine's 16 registers.

Data Register Direct The operand is in the data register specified by the register sub-field of the effective address specification. This mode is selected when the mode sub-field contains the value 0.

Address Register Direct The operand is in the address register specified by the register sub-field of the effective address specification. This mode is selected when the mode sub-field contains the value 1.

2.2.2 Memory Addressing Modes

These addressing modes specify that the operand is in memory and is to be referenced relative to one of the machine's address registers.

Address Register Indirect The operand is the word in memory whose address equals the value currently stored in the address register specified in the register sub-field. This mode is selected when the mode sub-field is 2.

Address Register Indirect with Postincrement The operand is the word in memory whose address equals the value currently stored in the address register specified in the register sub-field. After the operand is used, the value in the address register specified is incremented by one. This mode is selected when the mode sub-field is 3.

Address Register Indirect with Predecrement The value stored in the address register specified by the register sub-field is first decremented by one. The operand is the word in memory whose address equals the decremented value stored in the address register specified. This mode is selected when the mode sub-field is 4.

Address Register Indirect with Displacement This addressing mode requires one extension word. The operand is the word in memory whose address is the sum of the value in the extension word and the value in the address register specificed in the register sub-field of the effective address specification. In computing the address, the value in the address register is treated as a 16-bit unsigned integer while the value in the extension word is treated as a signed integer represented using 2's complement notation. This mode is selected when the mode sub-field is 5.

2.2.3 Special Addressing Modes

When one of the special addressing modes described below is used, the mode sub-field of the effective address specification is set to 7 and the register sub-field is used to specify exactly which of the special addressing modes is to be used. Each of the special addressing modes requires one extension word.

Absolute Address The operand is the word in memory whose address equals the value stored in this extension word. The value in the extension word is interpreted as a 16-bit unsigned integer. This mode is selected when the value in the mode sub-field is 7 and the value in the register sub-field is 0.

Program Counter Indirect with Displacement The operand is the word in memory whose address equals the sum of the value in the extension word and the value currently in the program counter register.[1] In computing the address, the value in the program counter is treated as a 16-bit unsigned integer, while the value in the extension word is treated as a 16-bit signed integer represented using 2's complement notation. This mode is selected when the mode sub-field is 7 and the value in the register sub-field is 2.

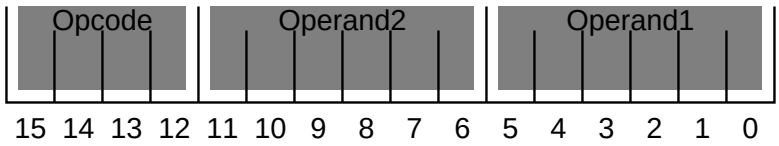
Immediate Data The operand is the extension word. This mode is selected when the mode sub-field is 7 and the value in the register sub-field is 4.

[1] The value in the program counter at this point will be the address of the extension word.

2.3 Instruction Formats

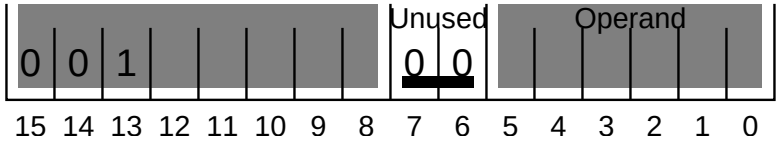
The operation fields of all WC34000 instructions are encoded using one of the four formats described below.

Two Operand Format The operation field of all two operand instructions except LINK are encoded as shown in the following diagram: Only 12 of the 16 possible bit patterns that could appear



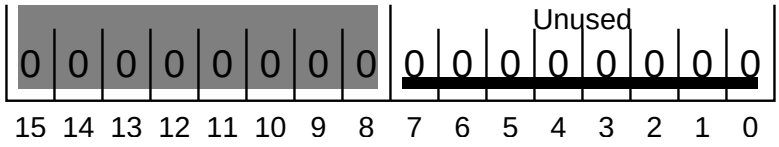
in the 4-bit OP-CODE component of such an instruction are used for two operand instructions. The remaining bit patterns are reserved for use as prefixes for the opcodes of instructions using other encoding formats. The two operand fields are used to hold effective address specifications. Important: Operand 1 is often used as the source operand and operand 2 the destination. Any associated extension words follow the instruction in that order.

Single Operand Format All single operand instructions are encoded using the format shown below: The operand field is interpreted as an effective address specification. Branches are included



as members of this group of instructions. Thus, on the WC34000, the target of a branch is generally specified by explicitly using the “Program Counter Indirect with Displacement” addressing mode.

Zero Operand Format Zero operand instructions are encoded using an 8-bit op-code followed by 8 bits whose values are ignored by the processor.



The LINK Instruction Format The LINK instruction is the exception to the rule that all WC34000 instructions use a uniform encoding scheme. A special format is used for this instruction. It is shown below.

The 'Register' field specifies the address register whose value is to be saved and then modified. The 'Operand' field uses an effective address specification to indicate the value by which the SP register should be incremented when the instruction is executed.

3 Instruction Set

Brief descriptions of all of the instructions the WC34000 implements are given below. The numeric values of the operation codes for these instruction are listed in an appendix at the end of this document.

Instruction		Description
ADD	src-ea, dst-ea	src + dst → dst Add the value of the source operand to the destination operand.
AND	src-ea, dst-ea	bitwise (src & dst) → dst Replace the destination operand with the bit-wise logical AND of the source and destination operand values.
ASL	src-ea, dst-ea	dst shifted left by src bits → dst Shift the value of the destination operand left by a number of bits equal to the value of the source operand.
ASR	src-ea, dst-ea	dst shifted right by src bits → dst Shift the value of the destination operand right by a number of bits equal to the value of the source operand. The shift is arithmetic, i.e. the sign bit is propagated as the shift is performed.
BEQ	dst-ea	if Z then address of dst → PC If the Z bit of the CCR is 1, transfer program execution to the instruction at the address specifi ed by the operand. Otherwise, the instruction has no effect.

Instruction		Description
BGE	dst-ea	if not N then address of dst → PC If the N bit of the CCR is zero, transfer program execution to the instruction at the address specified by the operand. Otherwise, the instruction has no effect.
BGT	dst-ea	if not Z & not N then address of dst → PC If the Z and N bits of the CCR are zero, transfer program execution to the instruction at the address specified by the operand. Otherwise, the instruction has no effect.
BLE	dst-ea	if Z or N then address of dst → PC If the Z bit of the CCR is 0 or the N bit is 1, transfer program execution to the instruction at the address specified by the operand. Otherwise, the instruction has no effect.
BLT	dst-ea	if N then address of dst → PC If the N bit is 1, transfer program execution to the instruction at the address specified by the operand. Otherwise, the instruction has no effect.
BNE	dst-ea	if not Z then address of dst → PC If the Z bit of the CCR is 0, transfer program execution to the instruction at the address specified by the operand. Otherwise, the instruction has no effect.
CLR	dst-ea	0 → dst The operand is set to zero.
CMP	src-ea, dst-ea	dst - src Subtract the source operand's value from the destination operand's value, but do not modify either operand. Instead, simply update the bits of the condition code register to reflect the result of the subtraction. In particular the N bit should be set to 1 only if the result is negative. The Z bit should be set to 1 only if the result is zero.
DIVS	src-ea, dst-ea	dst / src → dst Divide the value of the destination operand by the value of the source operand and store the quotient in the destination. All quantities are 16 bits wide.
EOR	src-ea, dst-ea	bitwise (dst ≠ src) → dst Replace the destination operand with the bit-wise exclusive OR of the source and destination operand values.
GETCH	dst-ea	input → dst Replace the operand's value by the ASCII code representing the next available input character.
GETNUM	dst-ea	input → dst Read input characters until a complete decimal number has been found and store the value of the number in the operand.

Instruction		Description
HALT		Halt the processor.
JMP	dst-ea	address of dst → PC Transfer program execution to the effective address specified by the operand.
JSR	dst-ea	SP - 1 → SP; PC → (SP) address of dst → PC Push the address of the next instruction in memory onto the stack, then transfer program execution to the effective address specified by the operand.
LEA	src-ea, dst-ea	address of src → dst Store the value of the memory address of the source operand in the destination. The source operand may not be specified using either of the register direct addressing modes.
LINK	An, disp-ea	SP - 1 → SP; An → (SP); SP → An; SP + disp → SP Push the current value of the address register specified by the first operand onto the stack. Replace the value of An by the value of the SP register after the push. Finally, increment the SP register by the value of the instruction's second operand.
MOVE	src-ea, dst-ea	scr → dst Move the value of the source operand to the destination operand.
MULS	src-ea, dst-ea	src * dst → dst Replace the value of the destination operand with the product of the source and destination operand. If the size of the result exceeds 16 bits, only the low order 16 bits are stored.
NEG	dst-ea	- dst → dst The operand is replaced by the result of subtracting its original value from zero.
NOT	dst-ea	bitwise not of dst → dst The operand is replaced by the bit-wise logical negation of its value.
OR	src-ea, dst-ea	bitwise (src or dst) → dst Replace the destination operand with the bit-wise logical OR of the source and destination operand values.
OUTCH	dst-ea	dst → output Print the operand, treating its value as the ASCII representation of a single character (the high-order 8 bits of the operand are ignored).
OUTNUM	dst-ea	dst → output Print the operand, treating its value as a signed 16-bit integer.

Instruction		Description
PEA	dst-ea	SP - 1 → SP; address of dst → (SP) Compute the memory address of the operand and then push the address onto the stack.
POPREG	dst-ea	The operand is treated as a 16 bit string with one bit corresponding to each of the data and address registers. Bit 15 is associated with D0, 14 with D1, ... , 7 with A0, ... , and 0 is associated with A7. For each bit that is one, a word is popped off the stack and its value is loaded into the corresponding register.
PSHREG	dst-ea	The operand is treated as a 16 bit string with one bit corresponding to each of the data and address registers (see above). For each bit that is one, the corresponding register is pushed onto the stack.
RTD	src-ea	(SP) → PC; SP + 1 + src → SP Pop a new value for the PC from the top of the stack and then increment the SP register by the value of the instruction's operand.
SUB	src-ea, dst-ea	dst - src → dst Subtract the value of the source operand from the destination operand.
UNLK	dst-ea	dst → SP; (SP) → dst; SP + 1 → SP First, set the SP equal to the original value of the instruction's operand. Then, replace the operand's value by a value popped from the stack.

Appendix A: Operation Codes

Two Operand Instructions

MOVE	0100
ADD	0101
SUB	0110
CMP	0111
MULS	1000
DIVS	1001
AND	1010
OR	1011
EOR	1100
LEA	1101
ASL	1110
ASR	1111

Zero Operand Instructions

HALT	00000000

The LINK Instruction

LINK	0001000

Single Operand Instructions

CLR	00100001
NEG	00100010
NOT	00100011
OUTNUM	00100100
GETNUM	00100101
OUTCH	00100110
GETCH	00100111
RTD	00101000
UNLK	00101001
PEA	00101010
JMP	00101011
JSR	00101100
BEQ	00101101
BNE	00101110
BLT	00101111
BGT	00110000
BGE	00110001
BLE	00110010
PSHREG	00110011
POPREG	00110100

Appendix B: Addressing Mode Summary

Mode Field	Reg Field	Mode Name
000	*any*	Data Register Direct
001	*any*	Address Register Direct
010	*any*	Address Register Indirect
011	*any*	Address Register Indirect with Postincrement
100	*any*	Address Register Indirect with Predecrement
101	*any*	Address Register Indirect with Displacement
111	000	Absolute Addressing
111	010	Program Counter Indirect with Displacement
111	100	Immediate Data

EXAMPLE 34000 MACHINE CODE TRANSLATION FROM C

We disassemble, here, a full 34000 translation of the C code for addToTail, below. Beside each assembly language instruction, we disassemble the particular instruction into its binary form. To help demonstrate decoding of the instructions the separate fields of the opcode are delineated by spaces. Extension words provide further addresses, displacements, and constants demanded by the addressing modes. Careful study of this sheet will answer many questions about the assembly of instructions into 34000 machine code. Comments may be found to the right.

```
   typedef struct {                                 void addToTail(e **lp, short v)
      short data                                    {   e *head = *lp;
      struct e *next;                                   if (head == null) {
   } e;                                                    head = *lp = alloc(2);/* 2==sizeof(e) */
                                                           head->data = v;
   /* lp is a pointer to a pointer to the                  head->next = NULL;
    * head of the list.  The value v is                 } else {
    * added to the end of the list.  A                     addToTail(&(head->next),v);
    * hypothetical routine, alloc, is used               }
    * to generate a pointer to new memory. */      }
```

```
data:   equ     0
next:   equ     1
head:   equ     -1
lp:     equ     2 ;      NOTE: memory addresses of extension words increase left to right
v:      equ     3               ;    addr: opcode (X=usually 0) extension word 1 extension word 2 comments
addToTail:                      ;    ----- -------------------- ---------------- ---------------- --------
        link    a6,#-1          ;     $02: 0001000  110 111 100 1111111111111111                 ( -1)
        pshreg  #64  ; bit 6=a0 ;     $04: 00110011  XX 111 100 0000000001000000                 (a7=1,a6=2,a5=4,...)
        move    lp(a6),a0       ;     $06: 0100 001 000 101 110 0000000000000010                 (+2)
        move    (a0),head(a6)   ;     $08: 0100 101 110 010 000 1111111111111111                 (-1)
        cmp     #0,head(a6)     ;     $0A: 0111 101 110 111 100 0000000000000000 1111111111111111 (0,-1)
        bne     elsepart        ;     $0D: 00101110  XX 111 010 0000000000010001                 ($0e+$11=$1f)
        move    #2,-(sp)        ;     $0F: 0100 100 111 111 100 0000000000000010                 (+2)
        jsr     alloc           ;     $11: 00101100  XX 111 010 0000000000011010                 ($12+$1a=$2c)
        move    lp(a6),a1       ;     $13: 0100 001 001 101 110 0000000000000010                 (+2)
        move    a0,(a1)         ;     $15: 0100 010 001 001 000
        move    a0,head(a6)     ;     $16: 0100 101 110 001 000 1111111111111111                 (-1)
        move    v(a6),data(a0)  ;     $18: 0100 101 000 101 110 0000000000000011 0000000000000000 (+3,0)
        clr     next(a0)        ;     $1b: 00100001  XX 101 000 0000000000000001                 (+1)
        jmp     commonexit      ;     $1d: 00101011  XX 111 010 0000000000001001                 ($1d+$09=$27)
elsepart:
        move    v(a6),-(sp)     ;     $1f: 0100 100 111 101 110 0000000000000011                 (+3)
        move    head(a6),a0     ;     $21: 0100 001 000 101 110 1111111111111111                 (-1)
        pea     next(a0)        ;     $23: 00101010  XX 101 000 0000000000000001                 (+1)
        jsr     addToTail       ;     $25: 00101100  XX 111 010 1111111111011100                 ($26-$24=$02)
commonexit:
        popreg  #64 ; bit 6=a1  ;     $27: 00110100  XX 111 100 0000000001000000                 (see pshreg, above)
        unlk    a6              ;     $29: 00101001  XX 001 110                                  (not like link)
        rtd     2               ;     $2a: 00101000  XX 111 000 0000000000000010                 (+2)
alloc:  ...                     ;     $2c:...
```