Reference manual is <http://www.bitsavers.org/components/intel/i860/240329-002_i860_64-Bit_Microprocessor_Programmers_Reference_Feb89.pdf>

The m88k simulator is available on Canvas as m88k\_subset.c in the simulations subdirectory.

Features of the i860 instruction set

* 32 integer registers, with r0 always 0
* 32 floating-point registers plus four special operand floating-point registers (kr, ki, t, and merge)
* processor status register and extended PSR
* operand sizes of 8, 16, 32, 64, and 128 bits
* dual-issue mode for fetching, decoding, and executing aligned pairs of integer and floating-point instructions (a simple form of VLIW)
* visible pipeline
* fast loop closing instruction with test and increment

Some unusual characteristics

* low bit in integer load/store address offsets is used to encode operand length (see page B-2: "When src1 is immediate and bit 28 is set, bit zero of the immediate value is forced to zero.")
* auto-increment addressing only for floating-point load and stores
* lock and unlock instructions for protecting trap handlers

We will implement a subset of the processor state

* 32 x 32-bit general registers (note that r0 is always 0)
* 32-bit instruction pointer (i.e., program counter)
* condition code bit
* can represent the registers as an int array in C and the IP and CC as int variables

We will implement a subset of the memory address space

* we will limit the simulation to a 1 MiB memory
* byte addresses aligned accesses on words (thus the memory endian choice doesn't matter)
  + an instruction starts on an address that is a multiple of 4
  + a data word starts on an address that is a multiple of 4
* we can represent memory as an int array in C

Some of the features that we will not implement

* other PSR/EPSR bits, including OF condition code bit and SC saved shift count
* delayed branches, interrupts, or traps
* little-endian byte-addressing with optional big-endian
* virtual memory
* floating-point registers and instructions
* dual issue and visible pipeline

Simulation steps

* you can adapt the structure of the example m88k simulator
  + you will have to pay attention to the differences in instruction details, e.g., sign extension for immediate values in the i860 versus zero-extension in the m88k
  + you are welcome to rewrite to avoid using global variables
* three modes (in the same manner as the example m88k simulator)
  + execution statistics only, which is the default
  + instruction trace, indicated by -t command line flag
  + verbose, indicated by -v command line flag
* load memory from stdin (in the same manner as the example m88k simulator)
  + echo the input in verbose mode
* zero registers, counters, and halt flag
* starting at address 0 the simulation will fetch, decode, and execute the instructions
  + similar to the example m88k simulator but includes the CC bit
    - fetch – load ir using the ip and increment the ip

if( verbose ) printf( "at %02x, ", ip );

ir = mem[ ip >> 2 ]; /\* adjust for word addressing of mem[] \*/

ip = ip + 4;

inst\_fetches++;

* + - decode – extract different fields from ir and then traverse the switch statement
    - execute – call appropriate routine (some may change the ip)
    - force r0 to zero
  + print instructions in trace mode
  + additionally print registers and CC bit after each instruction in verbose mode
  + continue until a halt instruction sets the halt flag
* always print dynamic execution statistics (in the same manner as the example m88k simulator)
  + instruction fetches
  + memory reads
  + memory writes
  + total number of branches
  + taken branches

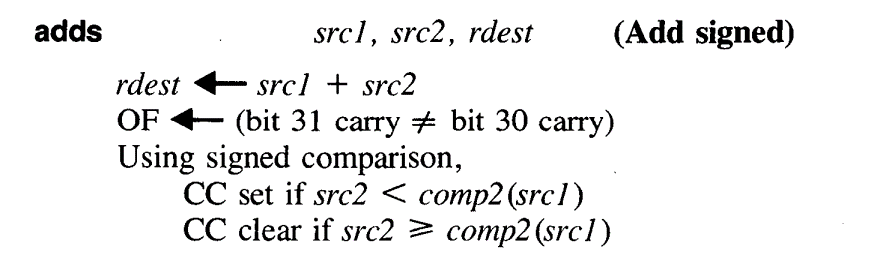
We will implement 21 instructions (from 12 basic i860 instructions; note that halt is not in i860)

|  |  |  |  |
| --- | --- | --- | --- |
| opcode | manual | simulation prints for trace and verbose modes | notes |
| 0x0 | n.a. | printf( "halt\n" ); | set halt flag |
| 0x4 | 5-2 | printf( "ld.l r%x(r%x),r%x\n", src1, src2, dest ); | load integer, general format |
| 0x5 | 5-2 | printf( "ld.l %x(r%x),r%x\n", immed\_16\_bit, src2, dest ); | load integer, immediate format, # |
| 0x7 | 5-3 | printf( "st.l r%x,%x(r%x)\n", src1, offset\_16\_bit, src2 ); | store integer, store format, # |
| 0x14 | 5-12 | printf( "btne r%x,r%x,%x", src1, src2, offset\_16\_bit ); | branch if equal, bte format |
| 0x15 | 5-12 | printf( "btnei %x,r%x,%x", immed\_5\_bit, src2, offset\_16\_bit ); | branch if equal, bte with immediate format |
| 0x16 | 5-12 | printf( "bte r%x,r%x,%x", src1, src2, offset\_16\_bit ); | branch if equal, bte format |
| 0x17 | 5-12 | printf( "btei %x,r%x,%x", immed\_5\_bit, src2, offset\_16\_bit ); | branch if equal, bte with immediate format |
| 0x1a | 5-12 | printf( "br %x", offset\_26\_bit ); | unconditional branch, ctrl format, % |
| 0x1c | 5-12 | printf( "bc %x", offset\_26\_bit ); | branch on CC set, ctrl format |
| 0x1e | 5-12 | printf( "bnc %x", offset\_26\_bit ); | branch on CC clear, ctrl format |
| 0x24 | 5-6 | printf( "adds r%x,r%x,r%x\n", src1, src2, dest ); | add signed, general format |
| 0x25 | 5-6 | printf( "adds %x,r%x,r%x\n", immed\_16\_bit, src2, dest ); | add signed, immediate format |
| 0x26 | 5-6 | printf( "subs r%x,r%x,r%x\n", src1, src2, dest ); | subtract signed, general format |
| 0x27 | 5-6 | printf( "subs %x,r%x,r%x\n", immed\_16\_bit, src2, dest ); | subtract signed, immediate format |
| 0x28 | 5-8 | printf( "shl r%x,r%x,r%x\n", src1, src2, dest ); | shift left, general format |
| 0x29 | 5-8 | printf( "shli %x,r%x,r%x\n", immed\_16\_bit, src2, dest ); | shift left, immediate format |
| 0x2a | 5-8 | printf( "shr r%x,r%x,r%x\n", src1, src2, dest ); | shift right logical, general format |
| 0x2b | 5-8 | printf( "shri %x,r%x,r%x\n", immed\_16\_bit, src2, dest ); | shift right logical, immediate format |
| 0x2e | 5-8 | printf( "shra r%x,r%x,r%x\n", src1, src2, dest ); | shift right arithmetic, general format |
| 0x2f | 5-8 | printf( "shrai %x,r%x,r%x\n", immed\_16\_bit, src2, dest ); | shift right arithmetic, immediate format |

# - must clear operand length bit in immediate field

% - we do not implement the delayed branch; the effect is immediate

* most instructions will be simple



void adds(){ /\* pages 5-6 and 5-7, general format \*/

int src1\_value = reg[ src1 ];

int src2\_value = reg[ src2 ];

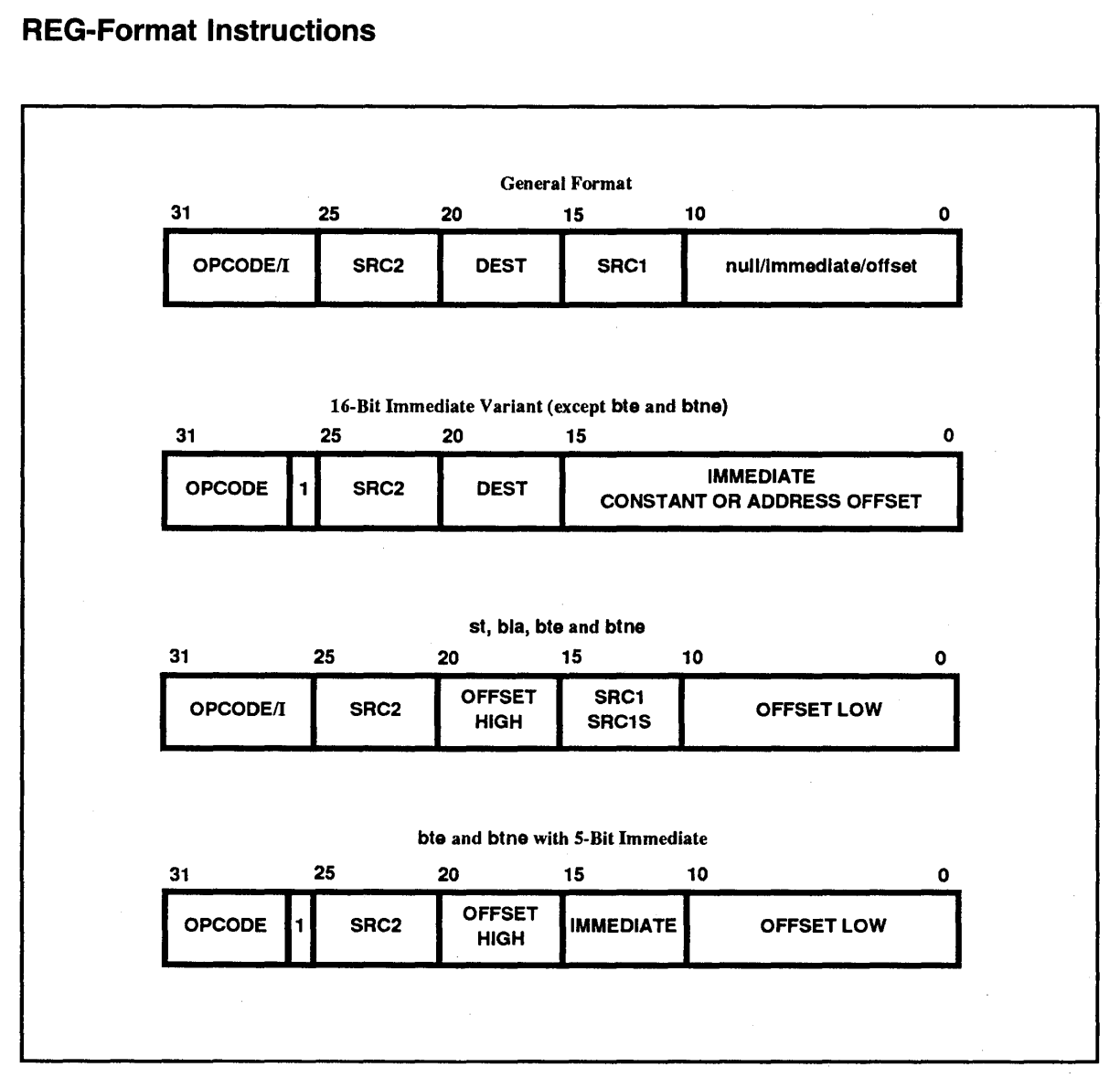
if( verbose ) printf( "adds r%x,r%x,r%x\n", src1, src2, dest );

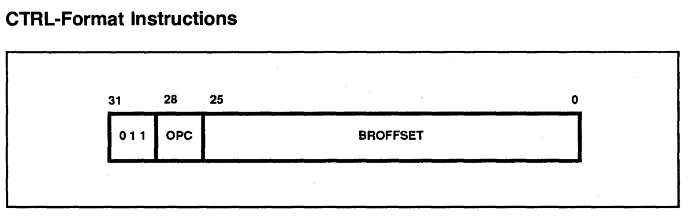
reg[ dest ] = src1\_value + src2\_value;

cc\_bit = ( src2\_value < -src1\_value ) ? 1 : 0;

}

* see the instruction formats on page B-1 of the manual





Example simulation

* input file

94010005

9402000c

90430800

00000000

* output using -t flag

instruction trace:

at 00, adds 5,r0,r1

at 04, adds c,r0,r2

at 08, adds r1,r2,r3

at 0c, halt

r0: 00000000 r8: 00000000 r10: 00000000 r18: 00000000

r1: 00000005 r9: 00000000 r11: 00000000 r19: 00000000

r2: 0000000c ra: 00000000 r12: 00000000 r1a: 00000000

r3: 00000011 rb: 00000000 r13: 00000000 r1b: 00000000

r4: 00000000 rc: 00000000 r14: 00000000 r1c: 00000000

r5: 00000000 rd: 00000000 r15: 00000000 r1d: 00000000

r6: 00000000 re: 00000000 r16: 00000000 r1e: 00000000

r7: 00000000 rf: 00000000 r17: 00000000 r1f: 00000000

cc: 0

execution statistics (in decimal):

instruction fetches = 4

data words read = 0

data words written = 0

branches executed = 0

branches taken = 0