HowToCode7

COLLABORATORS

	TITLE : HowToCode7		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		November 28, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

HowToCode7

1.1 HowToCode: Optimising

```
                         Optimising your Code
                         --------------------

Everyone wants their code to run as fast as possible, so here
are some speed-up tricks for you:

  1  68000 Optimisation
  2  68020 Optimisation
  3  Blitter Speed Optimisation
  4  General Speed-Up Notes
```

1.2 68000 Optimisation

```
68000 optimisation
------------------

Written by Irmen de Jong, march '93. (E-mail: ijdjong@cs.vu.nl)
Some notes added by CJ

NOTE! Not all these optimisations can be automatically applied. Make
sure they will not affect other areas in your code!

-----------------------------------------------------------------------------
Original    Possible optimisation   Examples/notes
-----------------------------------------------------------------------------
STANDARD WELL-KNOWN optimisATIONS
RULE: use Quick-type/Short branch! Use INLINE subroutines if they are small!
-----------------------------------------------------------------------------

BRA/BSR xx     BRA.s/BSR.s xx       if xx is close to PC

MOVE.X #0      CLR.X/MOVEQ/SUBA.X   move.l #0,count -> clr.l count
   move.l #0,d0    -> moveq #0,d0
   move.l #0,a0    -> sub.l a0,a0
```

```
CLR.L Dx        MOVEQ #0,Dx

CMP #0          TST

MOVE.L #nn,dx  MOVEQ #nn,dx         possible if -128<=nn<=127

ADD.X #nn       ADDQ.X #nn           possible if 1<=nn<=8
SUB.X #nn       SUBQ.X #nn           same...

JMP/JSR xx      BRA/BSR xx           possible if xx is close to PC

JSR xx;RTS      JMP xx               save a RTS
BSR xx;RTS      BRA xx               same...
                                     (assuming routine doesn't rely
                                     on anything in the stack)

LSL/ASL #1/2,xx ADD xx,xx [ADD xx,xx] lsl #2,d0 -> 2 times add d0,d0

MULU #yy,xx where yy is a power of 2, 2..256
   LSL/ASL #1-8,xx      mulu #2,d0 -> asl #1,d0 -> add d0,d0
   BEWARE: STATUS FLAGS ARE "WRONG"

DIVU #yy,xx where yy is a power of 2, 2..256
   LSR/ASR #.. SWAP  divu #16,d0 -> lsr #4,d0
            BEWARE: STATUS FLAGS ARE "WRONG",
            AND HIGHWORD IS NOT THE REMAINDER.

ADDRESS-RELATED OPTIMISATIONS
RULE: use short adressing/quick adds!
-----------------------------------------------------------------------------

MOVEA.L #nn     MOVEA.W #nn     Movea is "sign-extending" thus
                                possible if 0<=nn<=$7fff

ADDA.X #nn      LEA nn()        adda.l #800,a0 -> lea 800(a0),a0
                                possible if -$8000<=nn<=$7fff

LEA nn()        ADDQ.W #nn      lea 6(a0),a0 -> addq.w #6,a0
                                possible if 1<=nn<=8

$0000nnnn.l     $nnnn.w         move.l   4,a6 -> move.l 4.w,a6
                                possible if 0<=nnnn<=$7fff
                                (nnnn is SIGN EXTENDED to LONG!)

MOVE.L #xx,Ay  LEA xx,Ay       try xx(PC) with the LEA

MOVE.L Ax,Ay;
ADD #nnnn,Ay    LEA nnnn(Ax),Ay   copy&add in one

OFFSET-RELATED OPTIMISATIONS
RULE: use PC-relative addressing or basereg addressing!
      put your code&data in ONE segment if possible!
-----------------------------------------------------------------------------
MOVE.X nnnn     MOVE.X nnnn(pc)      lea copper,a0 -> lea copper(pc),a0..
LEA nnnn        LEA nnnn(pc)      ...possible if nnnn is close to PC

(Ax,Dx.l)       (Ax,Dx.w)            possible if 0<=Dx<=$7fff
```

```
If PC-relative doesn't work, use Ax as a pointer to your data block.
Use indirect addressing to get to your data: move.l Data1-Base(Ax),Dx etc.

TRICKY OPTIMISATIONS
-----------------------------------------------------------------------------
BSET #xx,yy    ORI.W #2^xx,yy       0<=xx<=15
BCLR #xx,yy    ANDI.W #~(2^xx),yy      "
BCHG #xx,yy    EORI.W #2^xx,yy         "
BTST #xx,yy    ANDI.W #2^xx,yy         "
               Best improvement if yy=a data reg.
               BEWARE: STATUS FLAGS ARE "WRONG".

SILLY OPTIMISATIONS (FOR OPTIMISING COMPILER OUTPUTS ETC)
-----------------------------------------------------------------------------
MOVEM (one reg.)  MOVE.l         movem  d0,-(sp) -> move.l d0,-(sp)

MOVE xx,-(sp)     PEA xx         possible if xx=(Ax) or constant.

0(Ax)             (Ax)

MULU/MULS #0      CLR.L          moveq #0,Dx with data-registers.

MULU #1,xx        SWAP CLR SWAP  high word is cleared with mulu #1
MULS #1,xx        SWAP CLR SWAP EXT.L  see MULU, and sign exteded.
                                    BEWARE: STATUS FLAGS ARE "WRONG"

LOOP OPTIMISATION.
-----------------------------------------------------------------------------
Example: imagine you want to eor 4096 bytes beginning at (a0).
Solution one:

      move.w   #4096-1,d7
.1    eori.b   d0,(a0)+
      dbra     d7,.1

Consider the loop from above. 4096 times a eor.b and a dbra takes time.
What do you think about this:

      move.w   #4096/4-1,d7
.1    eor.l    d0,(a0)+    ; d0 contains byte repeated 4 times
      dbra     d7,.1

Eors 4096 bytes too! But only needs 1024 eor.l/dbras.
Yeah, I hear you smart guys cry: what about 1024 eor.l without any loop?!
Right, that IS the fastest solution, but is VERY memory consuming (2 Kb).
Instead, join a loop and a few eor.l:

      move     #4096/4/4-1,d7
.1    eor.l    d0,(a0)+
      eor.l    d0,(a0)+
      eor.l    d0,(a0)+
      eor.l    d0,(a0)+
      dbra     d7,.1

This is faster than the loop before. I think about 8 or 16 eor.l's is just
fine, depending on the size of the mem to be handled (and the wanted
speed!). Also, mind the cache on 68020+ processors, the loop code must be
small enough to fit in it for highest speeds.
Try to do as much as possible within one loop (but considering the text
above) instead of a few loops after each other.

MEMORY CLEARING/FILLING.
---------------------------------------------------------------------------
A common problem is how to clear or fill some memory in a short time.
If it is CHIP-MEMORY, use the blitter (only D-channel, see below). In this
case you can still do other things with your 680x0 while the blitter is busy
erasing. If it is FAST-MEMORY, you can use the method from above, with
clr.l instead of eor.l, but there is a much faster way:

   move.l   sp,TempSp
   lea      MemEnd,sp
   moveq    #0,d0
;...for all 7 data regs...
   moveq    #0,d7
   move.l   d0,a0
;...for 6 address regs...
   move.l   d0,a6

After this, ONE instruction can clear 60 bytes of memory (15*4):

   movem.l  d0-d7/a0-a6,-(sp)    ;wham!

Now, repeat this instruction as often as required to erase the memory.
(memsize/60 times). You may need an additional movem.l to erase the last
few bytes. Get sp(=a7) back at the end with (guess..):

   move.l   TempSp,sp

If you are low on mem, put a few movem.l in a loop. But, now you need a
loop-counter register, so you'll only clear 56 bytes in one movem.l.

In the case of CHIP memory, you can use both the blitter and the processor
simultaneously to clear much CHIP mem in a VERY short time...
It takes some experimentation to find the best sizes to clear with the
blitter and with the processor.

BUT, ALWAYS USE A  WaitBlit()  AFTER CLEARING SIMULTANEOUSLY, even if you
think you know that the blitter is finished before your processor is.
```

1.3 General Speed-Up Notes

```
- When optimising programs first try to find the time-critical parts (inner
  loops, interrupt code, often called procedures etc.) In most cases 10%
  of the code is responsible for 90% of the execution time. Don't waste time
  doing needless optimising on startup and exit code when it's only
  called once!

- Often it is better not to set BLTPRI in DMACON (#10 in $dff09a) as this
  can keep your processor from calculating things while the blitter is busy.

- Use as many registers as possible! Store values in registers rather
```

  than in memory, it’s much faster!

- DON’T put your parameters on the stack before calling a routine! Instead,
  put them in registers!

- If you have enough memory, try to remove as many MULU/S and DIVU/S
  as possible by pre-calculating a multiplication or division table, and
  reading values from it, or rewrite multiply/divide code with simpler
  instructions if possible (eg ADD, LSR, etc.)