```
TITLE "M6800A - data_path.tdf - Data Path";

% This implements the entire data path %
% It currently occupies 278 LCs, and uses carry and cascade chains %
% %
% written 2000-Jun-04 by DCW, last modified 2000-Jun-05 16:42 %

INCLUDE "reg_pc.inc";
INCLUDE "reg_opadr.inc";
INCLUDE "reg_sp.inc";
INCLUDE "reg_x.inc";
INCLUDE "reg_a.inc";
INCLUDE "reg_b.inc";
INCLUDE "reg_cc.inc";
INCLUDE "data_mux.inc";
INCLUDE "alu.inc";
INCLUDE "daa.inc";
INCLUDE "cons_rom.inc";
INCLUDE "mux8_4_1.inc";

SUBDESIGN data_path
(
   % system clock %
   clk             : INPUT;     % 24 MHz system clock %

   % address bus and data busses to/from memory %
   mem_adr[15..0] : OUTPUT;    % memory address (64 Kbyte range) %
   smux_adr[1..0] : INPUT;     % control signals to select address source %
   mem_in[7..0]   : INPUT;     % input data bits (bits 7..0) %
   mem_out[7..0]  : OUTPUT;    % output data bits (bits 7..0) %

   % control signals to registers %
   cen_pch         : INPUT;     % PC %
   cen_pcl         : INPUT;     % PC %
   nload_pc        : INPUT;     % PC %
   s_pch           : INPUT;     % PC %
   s_pcl           : INPUT;     % PC %
   cen_oah         : INPUT;     % OP ADDR %
   cen_oal         : INPUT;     % OP ADDR %
   nload_oa        : INPUT;     % OP ADDR %
   nclr_oah        : INPUT;     % OP ADDR %
   nclr_oal        : INPUT;     % OP ADDR %
   cen_sph         : INPUT;     % SP %
   cen_spl         : INPUT;     % SP %
   nload_sp        : INPUT;     % SP %
   up_sp           : INPUT;     % SP %
   ce_xh           : INPUT;     % X %
   ce_xl           : INPUT;     % X %
   ce_a            : INPUT;     % A %
   ce_b            : INPUT;     % B %

   % control signals to ALU %
   s[1..0]         : INPUT;     % ALU channel select (0 = add, 1 = subtract, 2 = boolean, 3 = shift) %
   fs[1..0]        : INPUT;     % boolean function select (0 = AND, 1 = OR, 2 = XOR, 3 = NOT da) %
   left            : INPUT;     % shift direction (0 = right, 1 = left) %
   rot             : INPUT;     % type of shift (0 = shift, 1 = rotate) %
   arith           : INPUT;     % type of shift (0 = logical, 1 = arithmetic) %

   % control signals to ALU input muxes and constants ROM %
   smuxa[3..0]     : INPUT;     % %
   smuxb[1..0]     : INPUT;     % %
   scons[3..0]     : INPUT;     % %
   daa_out[1..0]   : OUTPUT;    % ?? %

   % control signals to condition code register, and CCR outputs %
   cenh            : INPUT;     % clock enable for H (bit 5) %
   ceni            : INPUT;     % clock enable for I (bit 4) %
   cenn            : INPUT;     % clock enable for N (bit 3) %
   cenz            : INPUT;     % clock enable for Z (bit 2) %
   cenv            : INPUT;     % clock enable for V (bit 1) %
   cenc            : INPUT;     % clock enable for C (bit 0) %
   cc[5..0]        : OUTPUT;    % the Condition Code Register %
)

VARIABLE

   result[7..0]    : node;      % result data bits from ALU %
   hin             : node;      % ?? %
   iin             : node;      % ?? %
   nin             : node;      % ?? %
```

```
    zero            : node;     % zero flag output from ALU %
    vin             : node;     % ?? %
    cout            : node;     % carry flag output from ALU %
    r_pc[15..0]     : node;     % output of Reg PC %
    r_opadr[15..0]  : node;     % output of Reg OP ADDR %
    r_sp[15..0]     : node;     % output of Reg SP %
    r_x[15..0]      : node;     % output of Reg X %
    r_a[7..0]       : node;     % output of Reg A %
    r_b[7..0]       : node;     % output of Reg B %
    r_cc[5..0]      : node;     % output of Reg CC %
    r_cons[7..0]    : node;     % output of constants ROM %
    mux_a[7..0]     : node;     % output of ALU input mux A %
    mux_b[7..0]     : node;     % output of ALU input mux B %

BEGIN

    % Register block %
    r_pc[]    = reg_pc ( result[], r_opadr[], s_pch, s_pcl, clk, cen_pch, cen_pcl, nload_pc );
    r_opadr[] = reg_opadr ( result[], clk, cen_oah, cen_oal, nload_oa, nclr_oah, nclr_oal );
    r_sp[]    = reg_sp ( result[], clk, cen_sph, cen_spl, nload_sp, up_sp );
    r_x[]     = reg_x ( result[], clk, ce_xh, ce_xl );
    r_a[]     = reg_a ( result[], clk, ce_a );
    r_b[]     = reg_b ( result[], clk, ce_b );

    r_cc[]    = reg_cc ( result[5..0], hin, iin, nin, zero, vin, cout, clk, cenh, ceni, cenn, cenz, cenv, cenc );
    hin       = GND;  % ?? %
    iin       = GND;  % ?? %
    nin       = GND;  % ?? %
    vin       = GND;  % ?? %
    cc[]      = r_cc[];  % %

    % ALU input muxes block %
    ( mux_a[], mux_b[] ) = data_mux ( r_pc[15..8], r_pc[7..0], r_sp[15..8], r_sp[7..0], r_x[15..8], r_x[7..0], r_a[], r_b[], r

    % Constants ROM block %
    r_cons[]  = cons_rom ( scons[] );
    daa_out[] = daa ( r_cc[0], r_cc[5], r_a[7..0] );

    % ALU block %
    ( result[], cout, zero )
              = alu ( mux_a[], mux_b[], r_cc[0], s[], fs[], left, rot, arith );

    % Memory address block %
    mem_adr[15..8] = mux8_4_1 ( VCC,VCC,VCC,VCC,VCC,VCC,VCC,VCC, r_pc[15..8], r_opadr[15..8], r_sp[15..8], smux_adr[] );
    mem_adr[7..0]  = mux8_4_1 ( VCC,VCC,VCC,VCC,VCC,VCC,VCC,VCC, r_pc[7..0],  r_opadr[7..0],  r_sp[7..0],  smux_adr[] );

    % Memory data bus block %
    mem_out[] = result[];

END;
```

```
TITLE "M6800A - data_mux.tdf - ALU input multiplexors";

% This implements the input multiplexors to the 8-bit Arithmetic/Logic Unit %
% %
% written 2000-Jun-04 by DCW, last modified 2000-Jun-04 14:55 %

INCLUDE "mux8_2_1.inc";
INCLUDE "mux8_4_1.inc";

SUBDESIGN data_mux
(
   dinpch[7..0]    : INPUT;     % register PC high byte %
   dinpcl[7..0]    : INPUT;     % register PC low byte %
   dinsph[7..0]    : INPUT;     % register SP high byte %
   dinspl[7..0]    : INPUT;     % register SP low byte %
   dinxh[7..0]     : INPUT;     % register X high byte %
   dinxl[7..0]     : INPUT;     % register X low byte %
   dina[7..0]      : INPUT;     % register A %
   dinb[7..0]      : INPUT;     % register B %
   dincc[5..0]     : INPUT;     % register CC (bits 7,6 always read '1') %
   dinmem[7..0]    : INPUT;     % memory %
   dincon[7..0]    : INPUT;     % constants ROM %
   sa[3..0]        : INPUT;     % input channel select for ALU input A %
                                % (0 = , ???????????? %
                                %  x = x) ?????????? %
   sb[1..0]        : INPUT;     % input channel select for ALU input B %
                                % (0 = constants ROM, 1 = memory, 2 = reg A, 3 = reg B ) %
   douta[7..0]     : OUTPUT;    % ALU input A %
   doutb[7..0]     : OUTPUT;    % ALU input B %
)

VARIABLE

   rmuxdx[7..0]    : node;      % result from mux for xh,xl,a,b %
   rmuxps[7..0]    : node;      % result from mux for pch,pcl,sph,spl %
   rmuxc0[7..0]    : node;      % result from mux for CC,00 %

BEGIN

   rmuxdx[] = mux8_4_1 ( dina[], dinb[], dinxh[], dinxl[], sa[1..0] );

   rmuxps[] = mux8_4_1 ( dinpch[], dinpcl[], dinsph[], dinspl[], sa[1..0] );

   rmuxc0[] = mux8_2_1 ( GND,GND,GND,GND,GND,GND,GND,GND, VCC,VCC,dincc[5..0], sa[0] );

   douta[]  = mux8_4_1 ( rmuxc0[], dinmem[], rmuxdx[], rmuxps[], sa[3..2] );

   doutb[]  = mux8_4_1 ( dincon[], dinmem[], dina[], dinb[], sb[] );

END;
```

```
TITLE "M6800A - alu.tdf - Arithmetic/Logic Unit";

% This implements the 8-bit Arithmetic/Logic Unit %
% It currently occupies 58 LCs, and uses carry and cascade chains %
% %
% written 2000-Jun-04 by DCW, last modified 2000-Jun-05 14:41 %

INCLUDE "alu_add.inc";
INCLUDE "alu_sub.inc";
INCLUDE "alu_bool.inc";
INCLUDE "alu_shft.inc";
INCLUDE "alu_zero.inc";
INCLUDE "mux8_4_1.inc";
INCLUDE "mux4_1.inc";

SUBDESIGN alu
(
   dina[7..0]     : INPUT;     % input data bits port a (bits 7..0) %
   dinb[7..0]     : INPUT;     % input data bits port b (bits 7..0) %
   cin            : INPUT;     % carry/borrow input %
   s[1..0]        : INPUT;     % ALU channel select (0 = add, 1 = subtract, 2 = boolean, 3 = shift) %
   fs[1..0]       : INPUT;     % function select (0 = AND, 1 = OR, 2 = XOR, 3 = NOT da) %
   left           : INPUT;     % shift direction (0 = right, 1 = left) %
   rot            : INPUT;     % type of shift (0 = shift, 1 = rotate) %
   arith          : INPUT;     % type of shift (0 = logical, 1 = arithmetic) %
   dout[7..0]     : OUTPUT;    % output data bits (bits 7..0) %
   cout           : OUTPUT;    % output carry/borrow bit %
   zout           : OUTPUT;    % output zero flag from zero detection %
)

VARIABLE

   radd[7..0]     : node;      % result data bits from add %
   rsub[7..0]     : node;      % result data bits from subtract %
   rbool[7..0]    : node;      % result data bits from boolean %
   rshft[7..0]    : node;      % result data bits from shift %
   rmux[7..0]     : node;      % result data bits from mux %
   coutadd        : node;      % output carry bit from alu_add %
   boutsub        : node;      % output borrow bit from alu_sub %
   coutshft       : node;      % output carry bit from alu_shft %

BEGIN

   ( radd[], coutadd )   = alu_add ( dina[], dinb[], cin );

   ( rsub[], boutsub )   = alu_sub ( dina[], dinb[], cin );

   ( rbool[] )           = alu_bool ( dina[], dinb[], fs[] );

   ( rshft[], coutshft ) = alu_shft ( cin, dina[], left, rot, arith );

   ( rmux[] )            = mux8_4_1 ( radd[], rsub[], rbool[], rshft[], s[] );

   ( cout )              = mux4_1 ( coutadd, boutsub, GND, coutshft, s[] );

   ( zout )              = alu_zero ( rmux[] );

   ( dout[] )            = rmux[];

END;
```

```
TITLE "M6800A - alu_add.tdf - ALU Adder";

% This implements the 8-bit ALU Adder %
% %
% written 2000-Jun-04 by DCW, last modified 2000-Jun-04 12:58 %

SUBDESIGN alu_add
(
   da[7..0]       : INPUT;     % input data bits, operand a (bits 7..0) %
   db[7..0]       : INPUT;     % input data bits, operand b (bits 7..0) %
   cin            : INPUT;     % carry input flag %
   s[7..0]        : OUTPUT;    % output data bits, sum (bits 7..0) %
   cout           : OUTPUT;    % carry output flag %
)

VARIABLE

   cc             : node;      % %
   c0             : node;      % %
   eqs0           : node;      % %
   eqc0           : node;      % %
   c1             : node;      % %
   eqs1           : node;      % %
   eqc1           : node;      % %
   c2             : node;      % %
   eqs2           : node;      % %
   eqc2           : node;      % %
   c3             : node;      % %
   eqs3           : node;      % %
   eqc3           : node;      % %
   c4             : node;      % %
   eqs4           : node;      % %
   eqc4           : node;      % %
   c5             : node;      % %
   eqs5           : node;      % %
   eqc5           : node;      % %
   c6             : node;      % %
   eqs6           : node;      % %
   eqc6           : node;      % %
   c7             : node;      % %
   eqs7           : node;      % %
   eqc7           : node;      % %

BEGIN

   % carry in %

   cc = CARRY(cin);

   % bit 0 %

   s0 = eqs0;
   c0 = CARRY(eqc0);

   eqs0 = (da0 $ db0) $ cc;
   eqc0 = da0 & cc
        # db0 & cc
        # da0 & db0;

   % bit 1 %

   s1 = eqs1;
   c1 = CARRY(eqc1);

   eqs1 = (da1 $ db1) $ c0;

   eqc1 = ( da1 & c0 )
        # ( db1 & c0 )
        # ( da1 & db1 );

   % bit 2 %

   s2 = eqs2;
   c2 = CARRY(eqc2);

   eqs2 = (da2 $ db2) $ c1;

   eqc2 = ( da2 & c1 )
        # ( db2 & c1 )
        # ( da2 & db2 );
```

```
    % bit 3 %

    s3 = eqs3;
    c3 = CARRY(eqc3);

    eqs3 = (da3 $ db3) $ c2;

    eqc3 = ( da3 & c2 )
         # ( db3 & c2 )
         # ( da3 & db3 );

    % bit 4 %

    s4 = eqs4;
    c4 = CARRY(eqc4);

    eqs4 = (da4 $ db4) $ c3;

    eqc4 = ( da4 & c3 )
         # ( db4 & c3 )
         # ( da4 & db4 );

    % bit 5 %

    s5 = eqs5;
    c5 = CARRY(eqc5);

    eqs5 = (da5 $ db5) $ c4;

    eqc5 = ( da5 & c4 )
         # ( db5 & c4 )
         # ( da5 & db5 );

    % bit 6 %

    s6 = eqs6;
    c6 = CARRY(eqc6);

    eqs6 = (da6 $ db6) $ c5;

    eqc6 = ( da6 & c5 )
         # ( db6 & c5 )
         # ( da6 & db6 );

    % bit 7 %

    s7 = eqs7;
    c7 = CARRY(eqc7);

    eqs7 = (da7 $ db7) $ c6;

    eqc7 = ( da7 & c6 )
         # ( db7 & c6 )
         # ( da7 & db7 );

    % carry out %

    cout = lcell(c7);

END;
```

```
% TITLE "M6800A - daa.mif - DAA Constants ROM Data Values" %

% This defines the data values in the DAA constants ROM %
% %
% written 2000-Jun-05 by DCW, last modified 2000-Jun-05 16:25 %

DEPTH = 1024;   % Memory depth, in words %
WIDTH = 2;      % Memory word width, in bits %

ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;

CONTENT

   BEGIN

         0  : 00;
      [1..3FF] : 11;

   END;
```

[000..009] : 00
00A.. 00F 01

M6800A INSTRUCTION SEQUENCING DETAILS

- Opcodes 0x,1x,3x,4x,5x are 1-byte inherent instructions, so inhibit PC increment after opcode+1 fetched. Refer to nodes controlu.tdf:1byt and controlu.tdf:???? for the implementation.

- There are 32 basic instruction operation cycle-by-cycle types. Use 5 bits of the 256x8 decode EAB to encode the type code. The first cycle of all instructions is the same - fetching the instruction opcode. The second cycle of all instructions appears to be the same when looking at the address and data busses, but the internal operation differs. The minimum number of cycles for any instruction is 2 cycles. Since the opcode is only decoded during the second cycle, the state machine does not branch to any sub-state for any 2-cycle instructions, i.e., there is only one state for the first cycle, shared by all instructions, and only one state for the second cycle, shared by all instructions. However, for instructions that have 3 or more cycles, (29 types), different states are branched to. Node controlu.tdf:2cyc indicates which instructions use 2 cycles so the state machine knows to return to the first state in the next cycle, and to set up the address mux select signals to be latched at the end of the 2nd cycle to select the PC register.

- immediate 8-bit reads (2 cycles): ADDx, ADCx, SUBx, SBCx, ANDx, ORAx, EORx, BITx, CMPx, LDAx
- immediate 16-bit reads (3 cycles): LDX, LDS, CPX

- direct 8-bit reads (3 cycles): ADDx, ADCx, SUBx, SBCx, ANDx, ORAx, EORx, BITx, CMPx, LDAx
- direct 16-bit reads (4 cycles): LDX, LDS, CPX
- direct 8-bit stores (4 cycles): STAx
- direct 16-bit stores (5 cycles): STX, STS

- indexed 8-bit reads (5 cycles): ADDx, ADCx, SUBx, SBCx, ANDx, ORAx, EORx, BITx, CMPx, LDAx
- indexed 16-bit reads (6 cycles): LDX, LDS, CPX
- indexed 8-bit stores (6 cycles): STAx
- indexed 16-bit stores (7 cycles): STX, STS
- indexed 8-bit read/modify/write (7 cycles): ASL, ASR, CLR, COM, DEC, INC, LSR, NEG, ROL, ROR, TST
- indexed jump (4 cycles): JMP
- indexed subroutine call (8 cycles): JSR

- extended 8-bit reads (4 cycles): ADDx, ADCx, SUBx, SBCx, ANDx, ORAx, EORx, BITx, CMPx, LDAx
- extended 16-bit reads (5 cycles): LDX, LDS, CPX
- extended 8-bit stores (5 cycles): STAx
- extended 16-bit stores (6 cycles): STX, STS
- extended 8-bit read/modify/write (6 cycles): ASL, ASR, CLR, COM, DEC, INC, LSR, NEG, ROL, ROR, TST
- extended jump (3 cycles): JMP
- extended subroutine call (9 cycles): JSR

- relative branch (4 cycles): BCC, BCS, BEQ, BGE, BGT, BHI, BLE, BLS, BLT, BMI, BNE, BPL, BRA, BVC, BVS
- relative subroutine call (8 cycles): BSR

- inherent subroutine return (5 cycles): RTS
- inherent 8-bit register operations (2 cycles): NOP
  ASLx, ASRx, CLRx, COMx, DECx, INCx, LSRx, NEGx, ROLx, RORx, TSTx
  ABA, SBA, CBA, DAA, TAB, TBA, TPA, TAP
- inherent condition code operations (2 cycles): CLC, CLI, CLV, SEC, SEI, SEV
- inherent 8-bit push (4 cycles): PSHx
- inherent 8-bit pop (4 cycles): PULx
- inherent 16-bit increment/decrement (4 cycles): DES, DEX, INS, INX
- inherent 16-bit transfer (4 cycles): TSX, TXS
- inherent software interrupt (12 cycles): SWI
- inherent return from interrupt (10 cycles): RTI
- inherent wait for interrupt (9 cycles): WAI

(x = A or B)

170 cycles
32 types

2 cycles: 3
3 cycles: 3
4 cycles: 9
5 cycles: 5
6 cycles: 4
7 cycles: 2

8 cycles: 2
9 cycles: 2
10 cycles: 1
12 cycles: 1

>>>> ???? But how to handle the divide-by-4 clock ???? <<<<

401/576 = 69%

Instruction sequencing.doc

M6800A INSTRUCTION SEQUENCING DETAILS

>>>> ???? But how to handle the divide-by-4 clock ???? <<<<

Description

Signals latched by clock rising edge
Signals after clock rising edge,
Signals set up to be latched by next clock

RESET* asserted (only recognized at instruction boundaries??)

next MEMADDR mux select signals = $FFFF, next RD/WR* signal = RD, next RESET state = RESET0,
next RUN state = RESET, next CC.I value = '1'

MEMADDR mux select signals latched, RD/WR* signal latched, RESET state latched, RUN state latched,
CC.I value latched
MEMADDR bus = $FFFF, RD/WR* = RD, RESET state = RESET0, RUN state = RESET, CC.I = '1',
next MEMADDR mux select signals = $FFFF, next RD/WR* signal = RD, next RESET state = RESET0,
next RUN state = RESET, next CC.I value = '1'
... (repeats as long as RESET* asserted) ...

RESET* negated

CONSTANTS ROM = $FE, ALU B MUX = CONSTANTS, BOOL = PASS, RESULT MUX = BOOL,
next MEMADDR mux select signals = $FFFF, next RD/WR* signal = RD, next RESET state = RESET1,
next RUN state = RESET, next OP ADDR function = LOAD / LO
MEMADDR mux select signals latched, RD/WR* signal latched, RESET state latched, RUN state latched,
OP ADDR low byte latched
OP ADDR = $??FE, MEMADDR bus = $FFFF, RD/WR* = RD, RESET state = RESET1, RUN state = RESET,
CONSTANTS ROM = $FF, ALU B MUX = CONSTANTS, BOOL = PASS, RESULT MUX = BOOL,
next MEMADDR mux select signals = OP ADDR, next RD/WR* signal = RD, next RESET state = RESET2,
next RUN state = RESET, next OP ADDR function = LOAD / HI,
MEMADDR mux select signals latched, RD/WR* signal latched, RESET state latched, RUN state latched,
OP ADDR high byte latched
OP ADDR = $FFFE, MEMADDR bus = $FFFE, RD/WR* = RD, RESET state = RESET2, RUN state = RESET,
ALU B MUX = MEM DBIN, BOOL = PASS, RESULT MUX = BOOL, PC HI MUX = RESULT,
next MEMADDR mux select signals = OP ADDR, next RD/WR* signal = RD, next RESET state = RESET3,
next RUN state = RESET, next OP ADDR function = INCREMENT, next PC function = LOAD / HI
MEMADDR mux select signals latched, RD/WR* signal latched, RESET state latched, RUN state latched,
PC high byte latched, OP ADDR incremented
OP ADDR = $FFFF, MEMADDR bus = $FFFF, RD/WR* = RD, RESET state = RESET3, RUN state = RESET,
PC = $VV??, ALU B MUX = MEM DBIN, BOOL = PASS, RESULT MUX = BOOL, PC LO MUX = RESULT,
next MEMADDR mux select signals = PC, next RD/WR* signal = RD, next RESET state = RUN,
next RUN state = RUN0, next PC function = LOAD / LO
MEMADDR mux select signals latched, RD/WR* signal latched, RESET state latched, RUN state latched,
PC low byte latched
PC = $VVVV, MEMADDR bus = $VVVV, RD/WR* = RD, RESET state = RUN, RUN state = RUN0,
next MEMADDR mux select signals = PC, next RD/WR* signal = RD, next RESET state = RUN,
next RUN state = RUN1, next PC function = INCREMENT, next IR function = LATCH
MEMADDR mux select signals latched, RD/WR* signal latched, RESET state latched, RUN state latched,
PC incremented, opcode latched into IR
PC = $VVVV+1, MEMADDR bus = $VVVV+1, RD/WR* = RD, RESET state = RUN, RUN state = RUN1,
etc., etc. ... (at this point we are already in normal instruction processing mode) ...

Normal Instruction Processing

PC = address of opcode, MEMADDR bus = address of opcode, RD/WR* = RD, RUN state = RUN0,
next MEMADDR mux select signals = PC, next RD/WR* signal = RD, next RUN state = RUN1,
next PC function = INCREMENT, next IR function = LATCH
MEMADDR mux select signals latched, RD/WR* signal latched, RUN state latched, PC incremented,
opcode latched into IR
PC = addr. of opcode + 1, MEMADDR bus = addr. of opcode + 1, RD/WR* = RD, RUN state = RUN1,
??????????????????, ??????????????????

Initial test program:

```
                        ORG     $FF80
FF80 01         START   NOP
FF81 20FD               BRA     START

                        ORG     $FFFE
FFFE FF80       RESETV  FDB     START

                        END
```

??????????????????

ADDA $1234 (3 bytes - 4 cycles)

R/W* set to "1", ...etc.
MEMADDR loaded with PC, PC incremented by 1, MEMDIR loaded with R/W*
byte of data (opcode) output by SRAM to
IR loaded with MEMDATAIN

BRA $1234 (2 bytes - 3 cycles)

next MEMADDR mux control signals = PC
MEMADDR mux control signals latched
PC value output to address bus, next MEMADDR mux control signals = PC
MEMADDR mux control signals latched, PC incremented by 1
PC value output to address bus, next MEMADDR mux control signals = $FFFF
MEMADDR mux control signals latched, PC incremented by 1
$FFFF value output to address bus

JMP $1234 (3 bytes - 3 cycles)

JMP 8,X (2 bytes - 3 cycles)

RTS (1 byte - 5 cycles)

* Cycle-by-cycle description:

- immediate 8-bit reads (2 bytes, 2 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch immediate operand byte - ADDR = PC, R/W = R
    + -

- immediate 16-bit reads (3 bytes, 3 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch immediate HI operand byte - ADDR = PC, R/W = R
    3 - fetch immediate LO operand byte - ADDR = PC, R/W = R
    + -

- direct 8-bit reads (2 bytes, 3 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch LO address byte - ADDR = PC, R/W = R
    3 - read operand byte - ADDR = OP, R/W = R
    + -

- direct 16-bit reads (2 bytes, 4 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch LO address byte - ADDR = PC, R/W = R
    3 - read HI operand byte - ADDR = OP, R/W = R
    4 - read LO operand byte - ADDR = OP, R/W = R
    + -

- direct 8-bit stores (2 bytes, 4 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch LO address byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - write operand byte - ADDR = OP, R/W = W
    + -

- direct 16-bit stores (2 bytes, 5 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch LO address byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - write HI operand byte - ADDR = OP, R/W = W
    5 - write LO operand byte - ADDR = OP, R/W = W
    + -

- indexed 8-bit reads (2 bytes, 5 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch offset byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - FFFF - ADDR = FF, R/W = R
    5 - read operand byte - ADDR = OP, R/W = R
    + -

- indexed 16-bit reads (2 bytes, 6 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch offset byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - FFFF - ADDR = FF, R/W = R
    5 - read HI operand byte - ADDR = OP, R/W = R
    6 - read LO operand byte - ADDR = OP, R/W = R
    + -

- indexed 8-bit stores (2 bytes, 6 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch offset byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - FFFF - ADDR = FF, R/W = R
    5 - FFFF - ADDR = FF, R/W = R
    6 - write operand byte - ADDR = OP, R/W = W
    + -

- indexed 16-bit stores (2 bytes, 7 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch offset byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - FFFF - ADDR = FF, R/W = R
    5 - FFFF - ADDR = FF, R/W = R
    6 - write HI operand byte - ADDR = OP, R/W = W
    7 - write LO operand byte - ADDR = OP, R/W = W
    + -

- indexed 8-bit read/modify/write (2 bytes, 7 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch offset byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - FFFF - ADDR = FF, R/W = R
    5 - read operand byte - ADDR = OP, R/W = R
    6 - FFFF - ADDR = FF, R/W = R
    7 - write operand byte - ADDR = OP, R/W = W
    + -

- indexed jump (2 bytes, 4 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch offset byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - FFFF - ADDR = FF, R/W = R
    + -

- indexed subroutine call (2 bytes, 8 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch offset byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - write LO byte return address - ADDR = SP, R/W = W
    5 - write HI byte return address - ADDR = SP, R/W = W
    6 - FFFF - ADDR = FF, R/W = R
    7 - FFFF - ADDR = FF, R/W = R
    8 - FFFF - ADDR = FF, R/W = R
    + -

- extended 8-bit reads (3 bytes, 4 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch HI address byte - ADDR = PC, R/W = R
    3 - fetch LO address byte - ADDR = PC, R/W = R
    4 - read operand byte - ADDR = OP, R/W = R
    + -

- extended 16-bit reads (3 bytes, 5 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch HI address byte - ADDR = PC, R/W = R
    3 - fetch LO address byte - ADDR = PC, R/W = R
    4 - read HI operand byte - ADDR = OP, R/W = R
    5 - read LO operand byte - ADDR = OP, R/W = R
    + -

- extended 8-bit stores (3 bytes, 5 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch HI address byte - ADDR = PC, R/W = R
    3 - fetch LO address byte - ADDR = PC, R/W = R
    4 - FFFF - ADDR = FF, R/W = R
    5 - write operand byte - ADDR = OP, R/W = W
    + -

- extended 16-bit stores (3 bytes, 6 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch HI address byte - ADDR = PC, R/W = R
    3 - fetch LO address byte - ADDR = PC, R/W = R
    4 - FFFF - ADDR = FF, R/W = R
    5 - write HI operand byte - ADDR = PP, R/W = W
    6 - write LO operand byte - ADDR = OP, R/W = W
    + -

- extended 8-bit read/modify/write (3 bytes, 6 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch HI address byte - ADDR = PC, R/W = R
    3 - fetch LO address byte - ADDR = PC, R/W = R
    4 - read operand byte - ADDR = OP, R/W = R
    5 - FFFF - ADDR = FF, R/W = R
    6 - write operand byte - ADDR = OP, R/W = W
    + -

- extended jump (3 bytes, 3 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch HI address byte - ADDR = PC, R/W = R
    3 - fetch LO address byte - ADDR = PC, R/W = R
    + -

- extended subroutine call (3 bytes, 9 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch HI address byte - ADDR = PC, R/W = R
    3 - fetch LO address byte - ADDR = PC, R/W = R
    4 - FFFF - ADDR = FF, R/W = R
    5 - write LO byte return address - ADDR = SP, R/W = W
    6 - write HI byte return address - ADDR = SP, R/W = W
    7 - FFFF - ADDR = FF, R/W = R
    8 - FFFF - ADDR = FF, R/W = R
    9 - FFFF - ADDR = FF, R/W = R
    + -

- relative branch (2 bytes, 4 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch offset byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - FFFF - ADDR = FF, R/W = R
    + -

- relative subroutine call (2 bytes, 8 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch offset byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - write LO byte return address - ADDR = SP, R/W = W
    5 - write HI byte return address - ADDR = SP, R/W = W
    6 - FFFF - ADDR = FF, R/W = R
    7 - FFFF - ADDR = FF, R/W = R
    8 - FFFF - ADDR = FF, R/W = R
    + -

- inherent subroutine return (1 byte, 5 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch dummy byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - read HI return address byte - ADDR = SP, R/W = R
    5 - read LO return address byte - ADDR = SP, R/W = R
    + -

- inherent 8-bit register operations (1 byte, 2 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch dummy byte - ADDR = PC, R/W = R
    + -

- inherent condition code operations (1 byte, 2 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch dummy byte - ADDR = PC, R/W = R
    + -

- inherent 8-bit push (1 byte, 4 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch dummy byte - ADDR = PC, R/W = R
    3 - write operand byte - ADDR = SP, R/W = W
    4 - FFFF - ADDR = FF, R/W = R
    + -

- inherent 8-bit pop (1 byte, 4 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch dummy byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - read operand byte - ADDR = SP, R/W = R
    + -

- inherent 16-bit increment/decrement (1 byte, 4 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch dummy byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - FFFF - ADDR = FF, R/W = R
    + -

- inherent 16-bit transfer (1 byte, 4 cycles):
    1 - fetch opcode byte - ADDR = PC, R/W = R
    2 - fetch dummy byte - ADDR = PC, R/W = R
    3 - FFFF - ADDR = FF, R/W = R
    4 - FFFF - ADDR = FF, R/W = R
    + -

```
- inherent software interrupt (1 byte, 12 cycles):
     1 - fetch opcode byte - ADDR = PC, R/W = R
     2 - fetch dummy byte - ADDR = PC, R/W = R
     3 - write LO byte return address - ADDR = SP, R/W = W
     4 - write HI byte return address - ADDR = SP, R/W = W
     5 - write LO byte index register - ADDR = SP, R/W = W
     6 - write HI byte index register - ADDR = SP, R/W = W
     7 - write Reg A byte - ADDR = SP, R/W = W
     8 - write Reg B byte - ADDR = SP, R/W = W
     9 - write Reg CC byte - ADDR = SP, R/W = W
    10 - FFFF - ADDR = FF, R/W = R
    11 - read HI byte of vector from FFFA - ADDR = OP, R/W = R
    12 - read LO byte of vector from FFFB - ADDR = OP, R/W = R
     + -

- inherent return from interrupt (1 byte, 10 cycles):
     1 - fetch opcode byte - ADDR = PC, R/W = R
     2 - fetch dummy byte - ADDR = PC, R/W = R
     3 - FFFF - ADDR = FF, R/W = R
     4 - read Reg CC byte - ADDR = SP, R/W = R
     5 - read Reg B byte - ADDR = SP, R/W = R
     6 - read Reg A byte - ADDR = SP, R/W = R
     7 - read HI byte index register - ADDR = SP, R/W = R
     8 - read LO byte index register - ADDR = SP, R/W = R
     9 - read HI byte return address - ADDR = SP, R/W = R
    10 - read LO byte return address - ADDR = SP, R/W = R
     + -

- inherent wait for interrupt (1 byte, 9 cycles):
     1 - fetch opcode byte - ADDR = PC, R/W = R
     2 - fetch dummy byte - ADDR = PC, R/W = R
     3 - write LO byte return address - ADDR = SP, R/W = W
     4 - write HI byte return address - ADDR = SP, R/W = W
     5 - write LO byte index register - ADDR = SP, R/W = W
     6 - write HI byte index register - ADDR = SP, R/W = W
     7 - write Reg A byte - ADDR = SP, R/W = W
     8 - write Reg B byte - ADDR = SP, R/W = W
     9 - write Reg CC byte - ADDR = SP, R/W = W
     + -
```