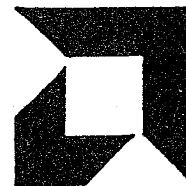


# ED29116

## DESIGNING WITH THE Am29116 16-BIT BIPOLAR MICROPROCESSOR

LECTURE
VOLUME II

AMD CUSTOMER EDUCATION

ED29116

"Designing with the Am29116 16-Bit Bipolar Microprocessor"

by

Barbara Albert
Arndt Bode, Dr. rer. nat.
J.W. Locke, Ph.D.

March 1983

Customer Education Center
Advanced Micro Devices, Inc.

Copyright by

ADVANCED MICRO DEVICES, INC.

9Ø1 Thompson Place

Sunnyvale

California 94Ø86

1983

# INDEX to VOLUME II


Day 2 — page

3. Exercises on the Am29116, part 1 — 3.1Ø
   - Exercises — 3.2Ø
   - Solutions — 3.8Ø

4. Am29116 Bit-Mapped Graphics Controllers — 4.1Ø
   - Introduction — 4.2Ø
   - Drawing a Vector — 4.3Ø
   - Fast Vector-Plotting Algorithm — 4.5Ø
   - Am29116 Microcode for Vector Generation — 4.8Ø
   - - Improving the Vector Algorithm — 4.11Ø

5. Application of the Am29116 for intelligent controllers — 5.1Ø
   - Intelligent controller structures — 5.15
     - ● low speed version — 5.3Ø
     - ● high speed version — 5.11Ø
     - ● comparison with an Am29Ø1 based solution — 5.17Ø
     - ● very high speed solution — 5.22Ø
       - - Am952Ø burst error processor — 5.24Ø
       - - Microprogramming the controller — 5.425

6. Application of the Am29116 for general purpose CPUs — 6.1Ø
   - A Microprogrammed CPU using Am29116 — 6.2Ø
   - System overview — 6.3Ø
   - System Organization — 6.4Ø
   - Instruction Formats — 6.6Ø
   - Timing analysis — 6.13Ø
   - Pipelining at the Macro level — 6.19Ø
   - Comparison with Super-16 — 6.27Ø
     - o Macro Instruction Execution — 6.34Ø
   - Comparison for 29Ø1-292Ø3-29116 solutions — 6.43Ø
   - Performance Analysis — 6.45Ø

7. Exercises on the Am29116, part 2 — 7.1Ø
   - Exercises (Microprogramming the Am29116) — 7.2Ø
   - Solutions — 7.3Ø

# DAY 2

# CHAPTER 3

Exercises, Part 1

Exercises - Part 1

True or false:

1. The Am29116 is externally TTL compatible, but uses ECL ciruitry internally.
2. The Am29116 is expandable (i.e. two can be hooked together).
3. The Am29116 is for 8-bit or 16-bit intelligent controllers.
4. The Am29116 can perform conditional testing on its status register.
5. The barrel shifter rotates 1 to 15 bits up or down in one microcycle.
6. The Am29116 must be used with an Am29∅4.
7. The Am29116 can perform immediate operations.
8. The Am29116 has a choice of four input sources to its data MUX's which in turn provide three ALU inputs, R, S, U.
9. The Am29116 can perform three-address instructions.
1∅. Fast clock speed is synonymous with high throughput.
11. The Am29116 can generate remainders up to 16 bits long from CRC polynomials.

Exercises - Part 1 (continued)

12. The Am29116 always has its ALU output at $Y_i$.

13. The ALU destinations are RAM, ACC, D-Latch.

14. Single-operand instructions are PASS, COMPLEMENT, INCREMENT, and TWO's COMPLEMENT.

15. D(ØE) (D with zero extend) is used for two's-complement arithmetic.

16. "$\overline{R}$ --> Dest" calculates one's-complement, and "$\overline{R}$ + 1 --> Dest" calculates two's complement.

17. The Am29116 can perform NAND, NOR, EXOR in one microcycle.

18. Shift up can use Ø, 1 or the QLINK bit as input to the LSB.

19. Shift down uses Ø, 1 or the QLINK bit as the only input choices to the MSB.

2Ø. Rotate operates in byte or word mode.

21. Rotate uses the U-input to the ALU.

22. Load $2^n$ causes a mask (1 in a field of Ø's) to be generated and can be used for loading RAM, ACC.

23. Read Y-bus, change a bit, output to Y-bus is possible in one microcycle with the Am29116.

24. If you perform a bit-oriented instruction on the ACC, the destination is the ACC or the RAM.

25. There are 17 possible results when priority encoding a word.

26. Byte-mode prioritizing ignores bits 8-15.

Exercises - Part 1 (continued)

27. The Am29116 can perform operations on polynomials of degree 16 or less.

28. 95% of CRC calculations use polynomials with 16-bit remainders.

29. The CRC calculations can be done in a forward or reverse mode. (i.e. either transmit bit Ø first or transmit bit 15 first)

3Ø. CRC remainders can be calculated in byte or word mode.

31. The status word can be loaded from D, RAM, or ACC.

32. The Z, C, N and OVR status bits can be loaded without affecting LINK or the user flags.

33. You can set or reset the entire status word.

34. You can set or reset the ALU flags individually.

35. On the Am29116, you can load both the status register and the ACC in the same microcycle if the RAM is the source.

36. If the status-register enable is high, the status register is "frozen". That is, no operation can alter status.

37. All conditional testing is performed on the stored values in the status register.

Exercises - Part 1 (continued)

Fill in or answer:

38. How many bits are in the Am29116 status register?

39. How many conditional tests can be made?

4Ø. Can you perform a conditional test during another instruction? If so, how?

41. Can byte operations be performed in either the upper byte or the lower byte of a register?

42. Can the Am29116 support a 1ØØ ns microcycle time?

43. List three possible sources for single operand instructions.

44. Can you load a byte into the D-latch?

45. List three possible source pairs for two-operand instructions.

46. Show the content of this register after a word-mode rotate with n=2.

| 15 | 14 | 13 | 12 | 11 | 1Ø | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | Ø |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | Ø | Ø | 1 | Ø | 1 | Ø | Ø | Ø | 1 | 1 | Ø | 1 | Ø | 1 |

47. Does the Am29116 allow you to rotate R3 and place the result back into R3?

48. Does the Am29116 allow you to rotate R3 and place the result in R7?

49. If there is no active priority request, what result is produced by a prioritize instruction in word mode?

Exercises - Part 1 (continued)

5Ø. If bit 15 is active, what result is produced by a prioritize instruction in word mode?

51. If bit 15 is active, what result is produced by a prioritize instruction in byte mode?

52. Is it true that the Am29116 is an order of magnitude faster than the Am29Ø1 which in turn is an order of magnitude faster than the AmZ8ØØØ for certain controller-oriented operations?

53. Can the Am29116 be used to do multiply? If so, outline the required code.

54. Can the Am29116 be used for bit operations?

55. Can the Am29116 be used for rotate operations?

56. Does the Am29116 have an ALU?

57. Can the Am29116 be used to build a CPU?

58. If a mask bit is zero in a ROTATE-and-MERGE instruction, from which operand is the corresponding bit passed to the destination?

59. If
```
       U = ØØ11  ØØØ1  Ø1Ø1  Ø11Ø
       R = 1Ø1Ø  1Ø1Ø  1Ø1Ø  1Ø1Ø
    mask = Ø1Ø1  1Ø1Ø  Ø11Ø  1ØØ1
```
what bit pattern is produced by a word-mode ROTATE-and-MERGE instruction with n=4?

6Ø. If the highest bit position with a one is position 7 and the mask is 1Ø1Ø $_{HEX}$, what is the result of a word-mode prioritize instruction?

61. If the highest bit position with a one is position 7 and the mask is 1Ø1Ø $_{HEX}$, what is the result of a byte-mode prioritize instruction?

Exercises - Part 1 (continued)

62. What is the result of loading the complement of $2^n$? What type of instruction allows you to do this?

63. Suppose you want to do a word-rotate down five bit positions. How do you do this on an Am29116?

64. What is QLINK?

65. Can you set and reset the ALU status bits individually as you can with the Am29Ø4 micro-status register?

66. Can you set & reset the LINK and FLAG status bits individually?

67. In byte mode, are all 8 bits of the status register loaded?

68. Which instructions do not cause the ALU status bits to be updated?

69. When are the upper four status bits (LINK, FLAG1, FLAG2, FLAG3) changed?

Exercises - Part 1

SOLUTIONS

Solutions for Exercises - Part 1 (continued)

1. True

2. False

3. True - almost every instruction operates in byte or word mode.

4. True - using either the instruction lines $I_1$ - $I_4$ or the $T_1$ - $T_4$ lines.

5. False - the barrel shifter only rotates up 1 to 15 bits.
   An effective down rotate is achieved by choosing an appropriate number of bits to rotate up such that the result is equivalent to the desired down rotation.
   (16-i in word mode; 8-i in byte mode)

6. False - but an Am29Ø4 is useful for emulations because of its bit-settable status register.

7. True - requires two microwords and two microcycles.

8. True - ACC, D-Latch, RAM and the instruction lines (for immediate data).

9. False - the Am29116 can be extended only to a two-register-address structure (using an external MUX, an additional 5-bit microinstruction field and extended timing).

1Ø. False! - recall the problems that occur on branching with a double-pipelined CPU based on Am291Ø & Am29Ø1 or Am29Ø3 and the need for extra NOP instructions.

11. True

Solutions for Exercises - Part 1 (continued)

12. True if $\overline{OE}_y$ is enabled. Otherwise, false.

13. True with reservations - the D-latch can be used but requires some tricky timing. You could generate a race condition if the D-latch is also a source for the operation. The D-latch is not intended to be a destination, and its use as such is not recommended. "Normal" destinations are RAM, ACC or NONE.

14. True. (PASS is another term for MOVE).

15. False - D(SE) (D sign-extended from bit 7) is used for two's complement arithmetic

16. True

17. True

18. True - shift uses the Ø, 1 or QLINK. The inputs to the carry MUX should not be confused with the inputs to the shift MUX.

19. False - QC, QN⊕QOVR and QLINK are also available. Do not confuse the use of Q in the Am29116 context (designating status-register contents) with the Q-shifter or Q-register of Am29Ø1, Am29Ø3 and Am292Ø3.

2Ø. True

21. True

22. True - also goes to internal Y-bus

23. True - be careful with the timing

24. False - ACC only. BONR instruction has common source/destination field.

25. True - none and 1 thru 16.

26. True

Solutions for Exercises - Part 1 (continued)

27. True

28. True - according to an AMD survey.

29. True

3Ø. False - Word mode only. But short polynomials that produce an 8-bit remainder can be used.

31. True - the status can also be loaded from immediate data as well.

32. True

33. True

34. False - if you need this use an Am29Ø4.

35. False - this is the one source which precludes loading both the ACC and the status register.

36. True

37. True

Answers to Fill-in or Answer Section:

38. Eight - Z, C, N, OVR, LINK, FLAG1, FLAG2, FLAG3

39. There are 12 condition code test signals -
You can test all 8 status bits individually. You can force a LOW.
And you can test 3 combinations: $Z+\overline{C}$, $N \oplus OVR$, $(N \oplus OVR)+Z$.

4Ø. Yes - by using the T-bus lines as input. This requires a wider microword to control the T-bus.

Solutions for Exercises - Part 1 (continued)

41. In byte mode, instructions alter only the lower byte.

42. Yes - Very carefully!
    This requires a register between the sequencer and the control store as well as the pipeline register at the output of the control store. 125 ns is more easily achieved.

43. RAM, ACC, D, I, Zero.

44. No.

45. RAM - ACC, RAM - I, D - RAM, D - ACC, ACC - I, D - I

46. Rotate is up only -

    1100 1010 0011 0101 n = 2 becomes
    0010 1000 1101 0111

47. Yes

48. Yes - with an external MUX and care with timing.

49. Zero.

50. One.

51. Bit 15 does not participate in the byte mode.
    The result will depend on the content of the lower byte.

52. Yes it is true. Especially when multi-bit rotation, priority determination or CRC remainder calculations are needed.

Solutions for Exercises - Part 1 (continued)

53. Yes - but it was not optimized for this application.

Consider a 16 x 16-bit unsigned multiplication:

i) Initialize the partial product (PP) and a counter to zero.

ii) Increment the counter.

iii) Test the MSB of the multiplier -
If the MSB is a one then: PP+multiplicand --> PP

iv) If count equals 16 then END
else upshift the PP and the multiplier.
(remember to carry from the LSH to the MSH of the PP)

v) Goto ii).

Look at the Am29116 instruction set! We can write a better procedure for the Am29116 than the above.

With the use of the prioritize and the rotate instructions you can reduce the number of microcycles.
(But now the number of microcycles depends on the multiplier!)

An improved algorithm is as follows:

i) Initialize PP and counter to zero.

ii) Prioritize the multiplier & call the result 'prio'.
If (prio = Ø) OR (prio > (16-counter))
then rotate PP by (16-counter) and END.

iii) count+prio --> count

iv) Rotate PP and multiplier by prio.

v) PP+multiplicand --> PP

vi) Goto ii).

Solutions for Exercises - Part 1 (continued)

54. Yes

55. Yes - The barrel shifter works in the byte or word mode.

56. Yes - The Am29116 has a 16-bit three-operand ALU.

57. Yes - but it was not optimized for this application.

58. The R-operand bit is passed to the destination.

59. The result is 1011 000 1110 0011.

60. The result is 9.
    Note: <mask bit i equal to zero> allows participation of the source bit i in the priority determination.

61. The result is one.

62. This clears the $n^{th}$ bit and sets all of the other bits in the destination. The instruction is:
    'BOR1, LDC2NR' or 'BONR, LDC2NA' or 'BONR, LDC2NY'.

63. Rotate up with n=11. ... (16-5=11)

    example: 0000 1111 0101 1010
             1101 0000 0111 1010

Solutions for Exercises - Part 1 (continued)

64. QLINK is the linkage bit for shift operations. It is also used by CRC instructions to bring in each bit of serial data.

65. You cannot set/reset the ALU status bits one by one. If you need this feature, use the Am2904 as an additional external status logic.

66. Yes

67. In byte mode, the lower 4 bits (ALU status) on the status register are loaded.

68. NOOP, save-status, test-status or any instruction if either of $\overline{IEN}$ or $\overline{SRE}$ are high.

69. The upper four status bits (LINK, FLAG 1, FLAG 2, FLAG 3) are changed during status set/reset; status load (word mode only); plus QLINK is updated after each shift

# CHAPTER 4

## Am29116 Bit-Mapped Graphics Controllers

Am29116 Bit-Mapped Graphics Controllers

The Am29116 has proved to be very popular amongst designers of graphics controllers. This is a natural result of the fact that the Am29116 has a very suitable instruction set for the kinds of algorithms that arise in graphics applications.

We will first discuss this subject in general using the article* by Chu and Miller as a reference. Then we will examine in detail one of the most useful procedures for bit-mapped raster scan graphics, the efficient drawing of a straight vector, and show how this is coded for the Am29116. By taking both an overall look at the requirements of bit-mapped graphics and by following a particular algorithm in detail, we are going to demonstrate why the Am29116 has been so successful as a graphics controller.

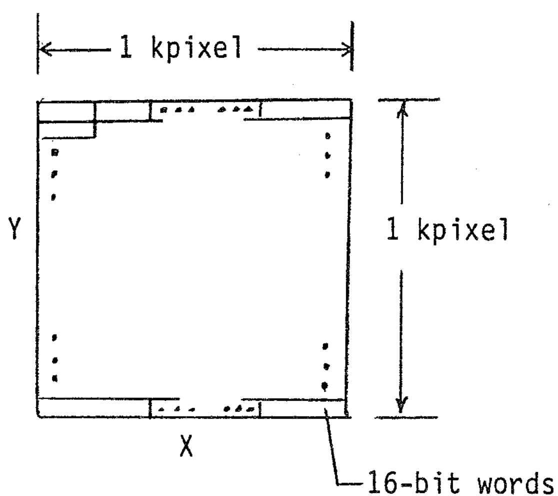
* "Microprocessor Architecture Suits Bit-Mapped Graphics"

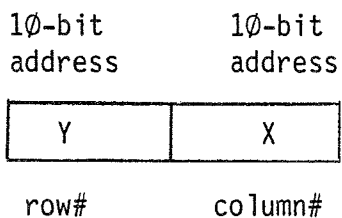
by Paul Chu and Warren Miller, Electronic Design, Jan.2Ø,1983 p.143

Am29116 Bit-Mapped Graphics Controllers (continued)

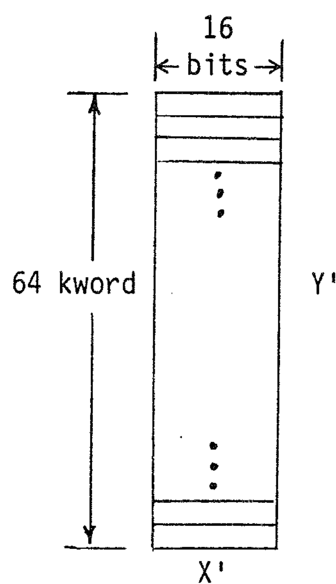
Logical Address to Physical Address Mapping

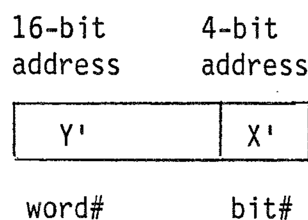
Display Memory (logical)




Bit-Map Memory (physical)

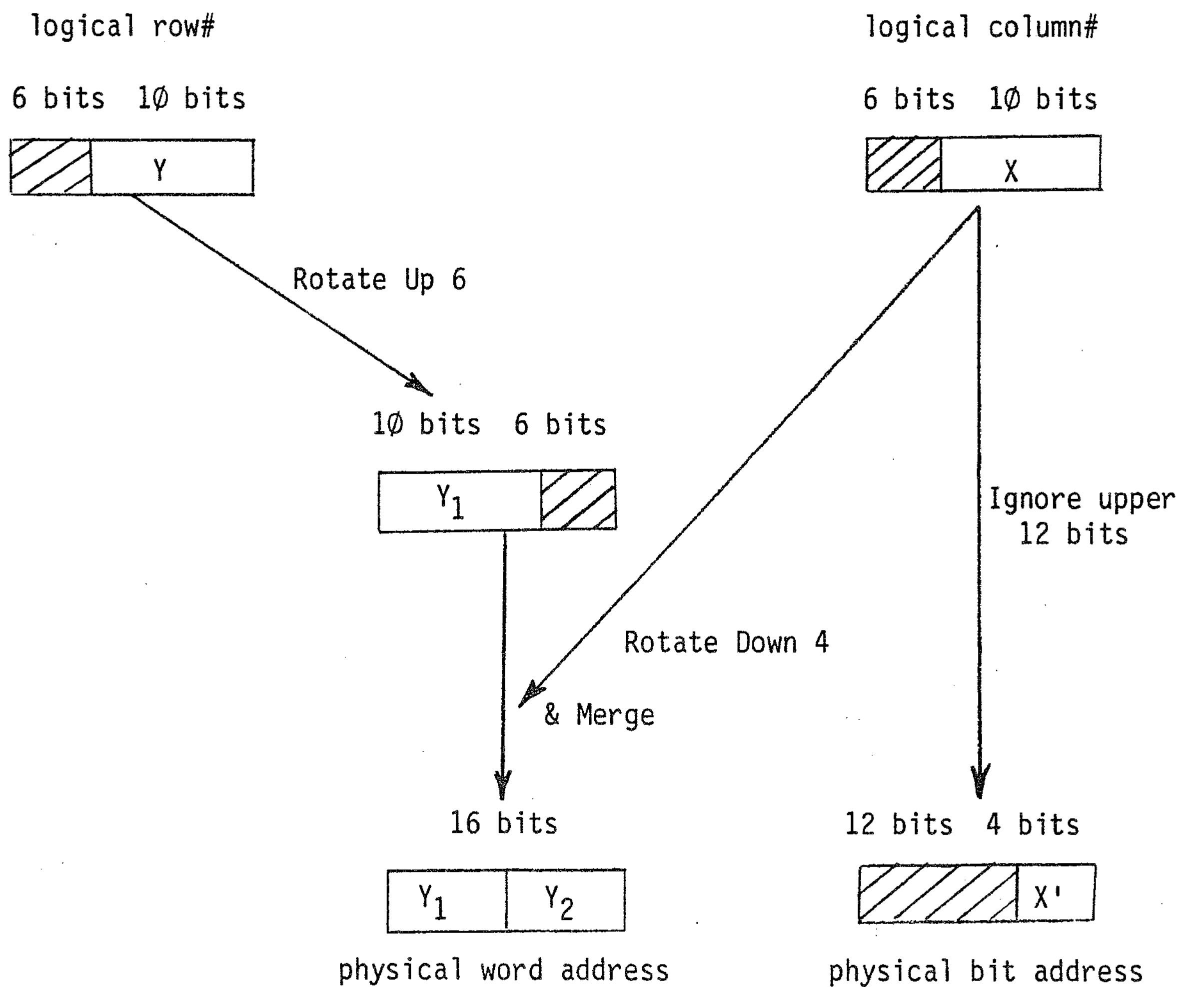


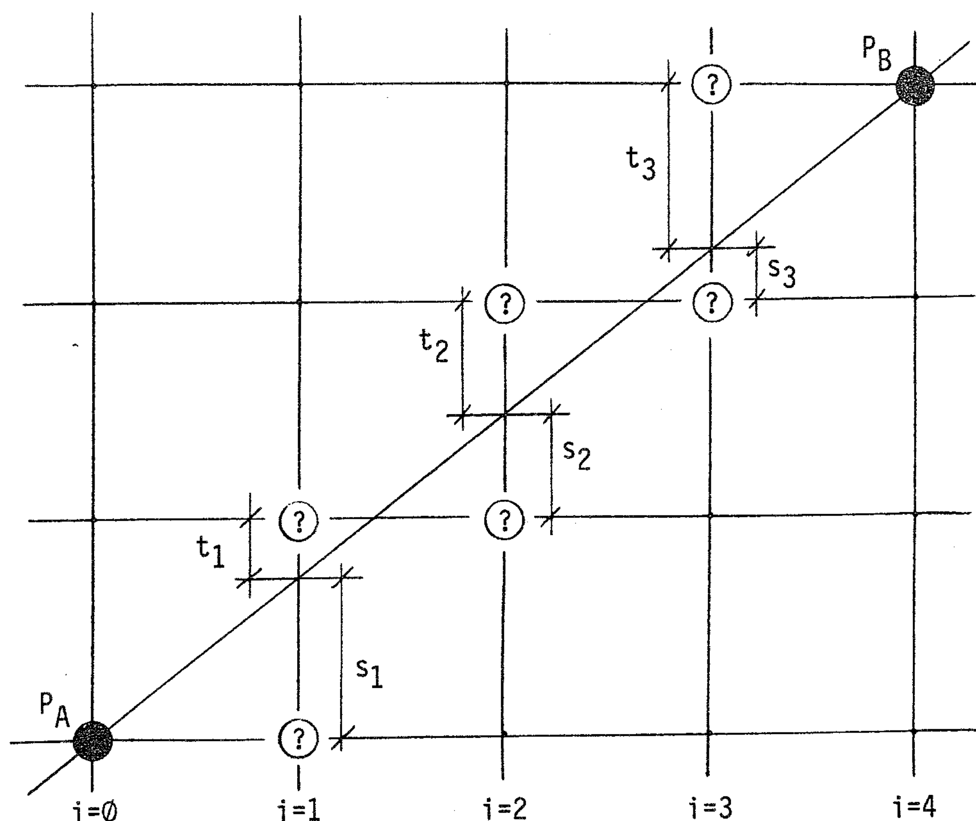
Am29116 Bit-Mapped Graphics Controllers (continued)

Logical Address to Physical Address Mapping (continued)

The mapping process:

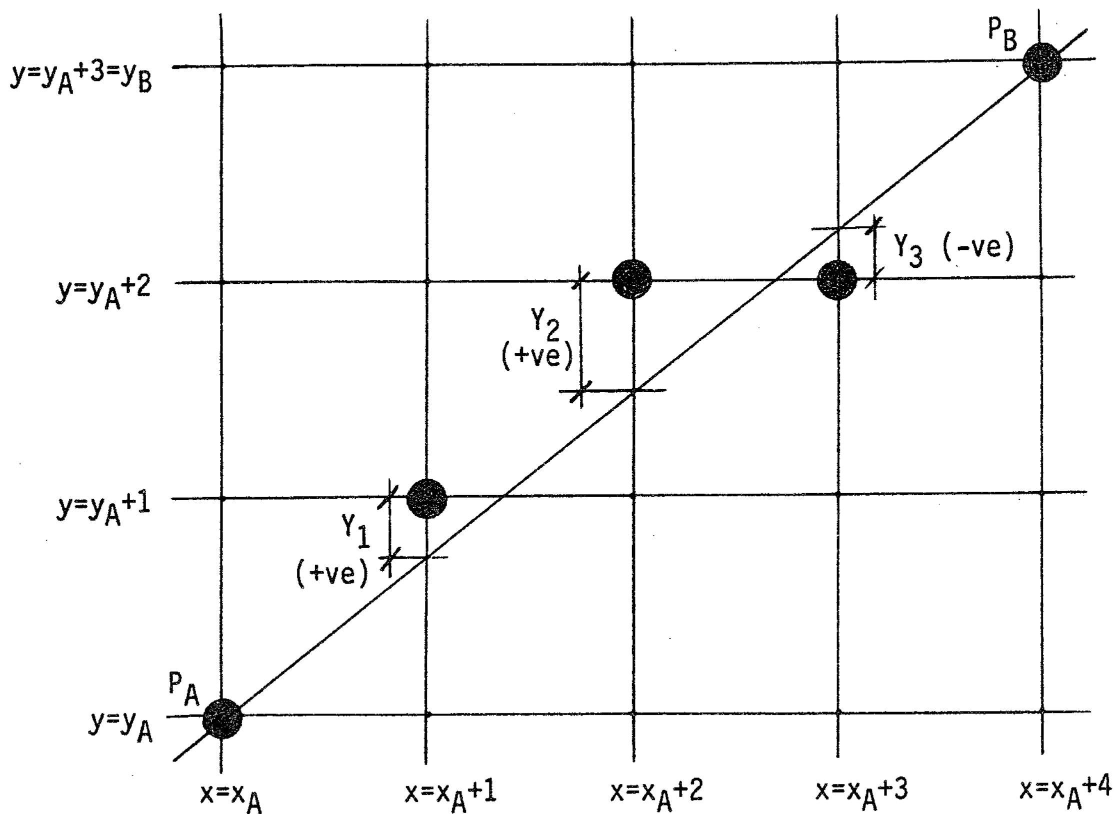


Drawing a Vector on a Two-Dimensional Raster Grid




We would like to find an efficient procedure to draw a straight vector between two points. When given the coordinates of two end points, $P_A$ and $P_B$, located on a raster grid, the procedure should set the bits in screen memory associated with these end points and with a set of intermediate points closely approximating a straight line. We will describe a procedure that works for lines of unit or less slope that selects the closer of the two candidate points for each increment of 'x'. That is, for each 'i' in the above figure, the procedure will compare the lengths of '$s_i$' and '$t_i$' and will select the point associated with the shorter length. This procedure can be generalized to handle lines with slope exceeding unity.

Drawing a Vector on a Two-Dimensional Raster Grid (continued)




- of course at $x=x_A$ and $x=x_B$, $Y=\emptyset$

In the above figure we see a successful application of such a procedure. Since the intermediate points are selected from a finite grid, they do not lie precisely on the ideal straight line linking $P_A$ with $P_B$. A measure of the success of the procedure is the value of the $Y_i$'s, the differences between the y-coordinates of the intermediate points and the corresponding vertical position of the ideal line.

An Algorithm for Fast Vector Plotting with an Am29116

For a straight line linking two points $P_A(x_A,y_A)$ and $P_B(x_B,x_B)$:

$$y = y(x) = \frac{(y_B-y_A)x+x_B y_A-x_A y_B}{x_B-x_A}$$

Let us define $dy=(y_B-y_A)$ and $dx=(x_B-x_A)$. For an arbitrary point $P(x_p,y_p)$, not necessarily on this line:

$$Y_p=(y_p-y(x_p))\cdot dx = -x_p\cdot dy+y_p\cdot dx-x_B y_A+x_A y_B$$

is proportional to the vertical distance of P from the line. We multiplied the distance by 'dx' to obtain a measure of vertical distance that can be calculated without performing a division. In all further discussion we will use this scaled measure for vertical distances. Since we will apply the same scale factor, 'dx', to all vertical distances, this tactic will not affect our ability to determine which of two points is closer to an ideal line.

A simple vector-drawing procedure is as follows:

1. Plot a point at $x_A,y_A$.
2. Increment x by one.
3. If $x > x_B$ then quit.
4. Calculate Y, the vertical distance error, scaled by 'dx', for two points: x,y and x,y+1.
5. If ABS(Y(x,y)) < ABS(Y(x,y+1)) then goto 6 else y=y+1.
6. Plot a point at x,y.
7. Goto 2.

This procedure may be referred to as a <u>digital differential analyser</u> by analogy with numerical methods for solving differental equations. This procedure has avoided division and multiplication but it still has redundant arithmetic. Two additions are required for the two Y's, we may have to perform a two's-complement operation on either or both of the Y's to produce absolute values and finally a subtraction is required to compare the Y's. We can compress this procedure further.

We have been choosing the next point based on the sign of

$$d = ABS(Y_{lower})-ABS(Y_{upper})$$

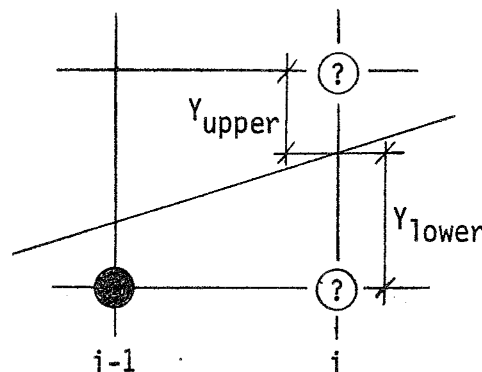
(We will now call 'd' the "discriminant" for our problem).

An Algorithm for Fast Vector Plotting with an Am29116 (continued)

But in the situation shown below, $Y_{lower}$ is negative and $Y_{upper}$ is positive. In this case:

$$d = -Y_{lower} - Y_{upper}$$

and the absolute-value operation is not required.




Further the $i^{th}$ discriminant can be obtained efficiently from the $(i-1)^{th}$ discriminant. To discover how, consider the Y's involved:

$$Y_{i-1} = -x_{i-1} \cdot dy + y_{i-1} \cdot dx - x_B y_A + x_A y_B$$

$$Y_{lower_i} = -(x_{i-1}+1) \cdot dy + y_{i-1} \cdot dx - x_B y_A + x_A y_B = Y_{i-1} - dy$$

$$Y_{upper_i} = -(x_{i-1}+1) \cdot dy + (y_{i-1}+1) \cdot dx - x_B y_A + x_A y_B = Y_{i-1} - dy + dx$$

Therefore: $$d_i = -Y_{lower_i} - Y_{upper_i} = -2Y_{i-1} + 2 \cdot dy - dx$$

And since $Y_Ø = Ø$: $$d_1 = 2 \cdot dy - dx$$

An Algorithm for Fast Vector Plotting with an Am29116 (continued)

An efficient algorithm for plotting a straight vector from given end points with minimal arithmetic can now be based on the use of a sequence of values of the above discriminant, d. We start with $x=x_A$, $y=y_A$, $d_1=2\cdot dy-dx$. For each point we check to see if $x>x_B$. If so we terminate the procedure. Otherwise we set the bit corresponding to x,y in the screen memory. Then we check the sign of $d_i$ and proceed to select the next point and the next discrimininant, $d_{i+1}$ as follows:

**If $d_i$ is -ve**

$x_i = x_i+1$
$y_i = y_i$
(i.e. select lower point)

$d_{i+1} = -2Y_i+2\cdot dy-dx$
$= -2(Y_{i-1}-dy)+2\cdot dy-dx$
$= d_i+2\cdot dy$

i.e. $d_{i+1} = d_i+incr1$
where $incr1 = 2\cdot dy$ (always +ve)

**If $d_i$ is +ve**

$x_i = x_i+1$
$y_i = y_i+1$
(i.e. select upper point)

$d_{i+1} = -2Y_i+2\cdot dy-dx$
$= -2(Y_{i-1}-dy+dx)+2\cdot dy-dx$
$= d_i+2\cdot dy-2\cdot dx$

i.e. $d_{i+1} = d_i+incr2$
where $incr2 = 2\cdot dy-2\cdot dx$ (always -ve)

---

Thus for each point plotted, we require only one addition plus one increment for 'x' and possibly one increment for 'y'. This improved procedure is Bresenham's[1] algorithm for a straight line. Variations on this algorithm can be devised to plot circular and elliptical arcs.

Reference: Bresenham,J.E. "Algorithm for Computer Control of a Digital Plotter" IBM Syst. J., Vol. 4, No. 1 (1965), pp.25-3Ø

## Am29116 Microcode for Vector Generation

Let us show the code that implements Bresenham's algorithm in two sections. First let us calculate the various quantities needed before the main loop begins.

We start with:

| Reg | Content |
|---|---|
| RØØ | $x_A$ |
| RØ1 | $y_A$ |
| RØ2 | $x_B$ |
| RØ3 | $y_B$ |

We want:

| Reg | Content |
|---|---|
| RØØ | $x_A$ |
| RØ1 | $y_A$ |
| RØ2 | dx=(pixel count)-1 |
| RØ3 | incr1 = 2·dy --- always +ve |
| RØ4 | incr2 = 2·dy-2·dx --- always -ve |
| ACC | $d_1$ = 2·dy-dx |

The code required to achieve this result is:

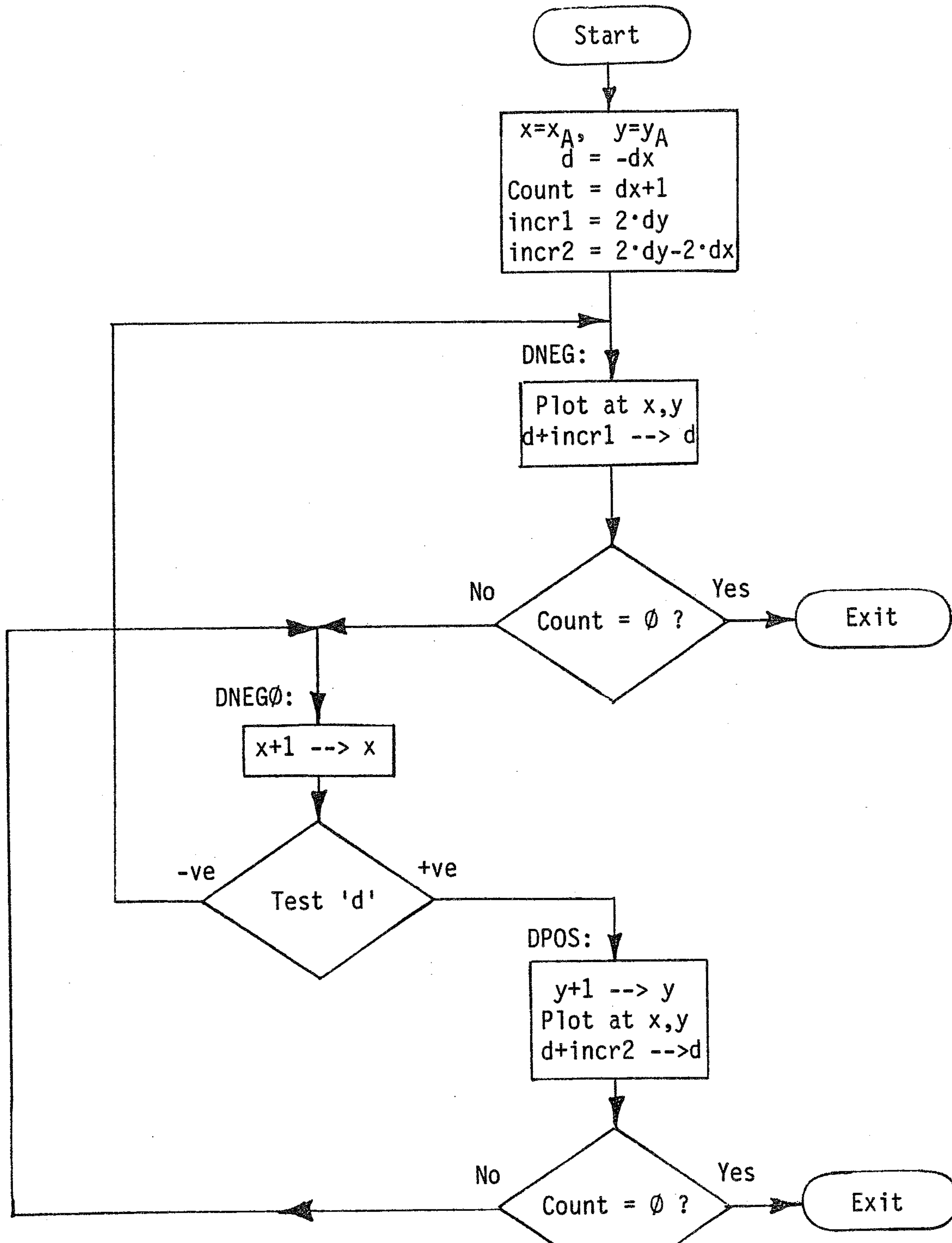
```
SOR      W,MOVE,SORA,RØØ     & CONT    ; xA          --> ACC
TOR1     W,SUBS,TORAA,RØ2    & CONT    ; dx=xB-xA    --> ACC
SOR      W,MOVE,SOAR,RØ2     & CONT    ; dx          --> RØ2 final dx
SOR      W,MOVE,SOAR,RØ4     & CONT    ; dx          --> RØ4
SOR      W,MOVE,SORA,RØ1     & CONT    ; yA          --> ACC
TOR1     W,SUBS,TORAA,RØ3    & CONT    ; dy=yB-yA    --> ACC
SHFTNR   W,SHA,SHUPZ,NRA     & CONT    ; 2·dy        --> ACC
SOR      W,MOVE,SOAR,RØ3     & CONT    ; 2·dy        --> RØ3 final incr1
TOR1     W,SUBR,TORAA,RØ2    & CONT    ; 2·dy-dx     --> ACC final d1 **
TOR1     W,SUBR,TORAR,RØ4    & CONT    ; 2·dy-2·dx   --> RØ4 final incr2
```

* Note: "SHFTR" is used to multiply by two.
The alternative of adding RAM to RAM is not available.

** Note: The particular implementation of the main loop we are about to discuss requires that '$d_1$' be adjusted further.

Am29116 Microcode for Vector Generation (continued)

Program Flow Chart for Main Loop of Bresenham's Vector Algorithm:



Am29116 Microcode for Vector Generation (continued)

The main loop section of the code is as follows:

```
       TOR1   W,SUBR,TORAA,RØ3                    ; -dx --> ACC   (final d1)
              & CONT

       SOR    W,INC,SORY,RØ2                      ; dx+1 --> Am291Ø counter
              & IFNOT CT16 & CT LOW & LDCT JUNK   ; Forced pass loads counter
              & JMPI     & OEY                    ; via "JUMP INDIRECT" path.

DNEG:         ; d is -ve
       YYYY   Instruction #1 of subroutine PLOT   ; avoid waste of 1 cycle
              & IFNOT CT16 & CT LOW & CJS PLOT+1  ; unconditional CALL

       TOR    W,ADD,TORAA,RØ3                     ; d+incr1 --> d
              & RPCT DNEGØ                        ; Last pt plotted? Jmp if not.

       AAAA   Any instruction                     ; last point has been plotted
              & IFNOT CT16 & CT LOW & CJP BRXIT   ; unconditional jmp to exit

DNEGØ: SOR    W,INC,SORR,RØØ                      ; x+1 --> x
              & IF CT16 & CT N & CJP DNEG         ; Test sign of d, branch if -ve,
                                                  ; continue if +ve.

DPOS:         ; d is +ve
       SOR    W,INC,SORR,RØ1                      ; y+1 --> y
              & IFNOT CT16 & CT LOW & CJS PLOT    ; unconditional CALL

       TOR    W,ADD,TORAA,RØ4                     ; d+incr2 --> d
              & RPCT DNEGØ                        ; Last pt plotted? Jmp if not.

BRXIT: XXXX   ; The vector is now completely drawn. Exit from Bresenham.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
PLOT:         ; Hardware-dependant routine to plot point at RØØ,RØ1
       YYYY   First instruction of PLOT subroutine
              & CONT

PLOT+1:ZZZZ   Second instruction of PLOT subroutine
       ....
       ....
       ....
       LLLL   Last instruction of PLOT subroutine
              & IFNOT CT16 & CT LOW & CRTN   ; return to caller of PLOT/PLOT+1
```

Am29116 Microcode for Vector Generation (continued)

Improving the Vector Algorithm:

The above implementation of Bresenham's algorithm can still be improved. As it stands, the code manipulates the logical addresses, 'x' and 'y', and then plots each point by a procedure ("PLOT") that must convert these address coordinates to physical coordinates. It would be better to work with the physical addresses directly and hence shorten "PLOT".

Suppose the physical address consists of a word address, 'W', and a bit address, 'b'. Suppose further that 'b' is represented by a word, 'B', that has a single bit set in the bit position corresponding to 'b'. Then the algorithm can be modified as follows:

1. Replace 'y+1 --> y' by:

   W+(# of pixels per line) --> W ... usually add $2^n$ via 'BOR2 A2NR'
   ('b' does not need to be altered)

2. Replace 'x+1 --> x' by:

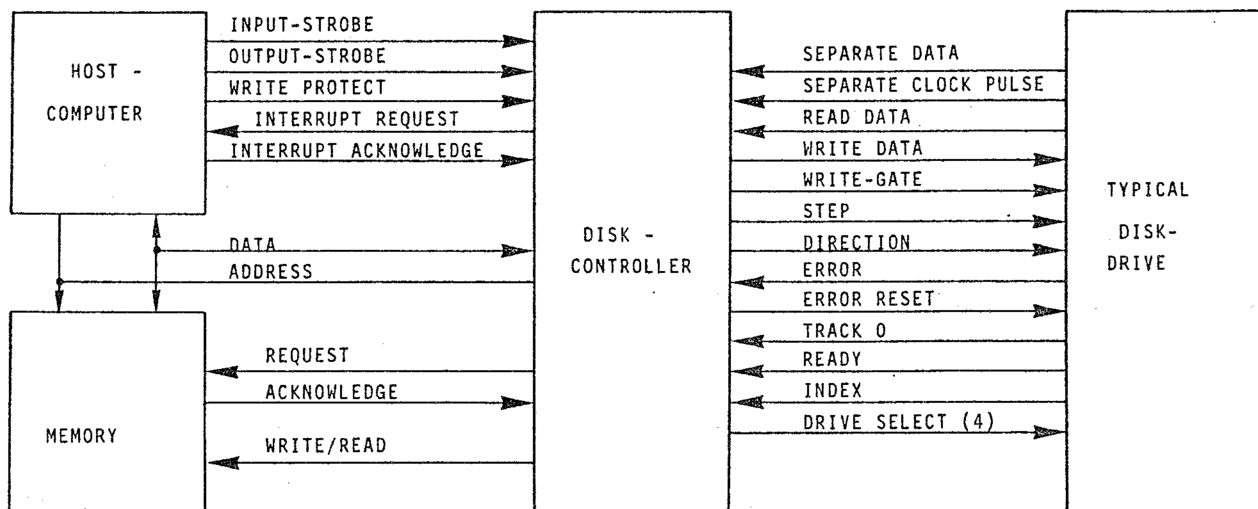
   Rotate B up one.
   Then if this rotation sets the sign bit,
   conditionally: 'W+1 --> W'

If the code is altered to manipulate 'W' and 'B' in this manner, then the subroutine, "PLOT", receives the address of the word in the screen memory to be modified, along with the bit bit mask with which to set that word. This shortens "PLOT" by at least 4 cycles. The implementation of this improvement is left as an exercise for the reader.

# CHAPTER 5

Intelligent Controllers Based on Am29116

## A Typical Peripheral Controller




BLOCK DIAGRAM OF A TYPICAL PERIPHERAL CONTROLLER

## Peripheral Controllers

The functions of a peripheral controller may include:

- Parallel transmission of data and address information between the host computer and the device being controlled.
- Detection and execution of commands from the host.
- Provision of status information to the host indicating the state of the controller and the controlled device.
- Serial transmission of data to and from the controlled device.
- Generation and testing of status, command and timing bits that coordinate controller/controlled-device interaction.
- Execution of calculations or algorithms related to the control of the peripheral or to processing data from it.

Intelligent Controller

Low Speed Version

Intelligent Controller - Low Speed

Devices Needed for the Minimum Configuration:

To control the Am29116:

- Am291Ø sequencer
- Microprogram Memory
- Pipeline Register (could be incorporated in a registered PROM that contains the microprogram)

To control the sequencer:

- 1 1/2 octal D-type flip-flops (Am292Ø's) forming a 12-bit register acting as a latch for the direct input to the Am291Ø sequencer

To interface with the host:

- 2 8-bit bidirectional I/O ports (Am295Ø's) connected to the data bus and providing the data path to and from the host
- 2 DMA address generators (Am294Ø's) to drive the host address bus
- 1 bidirectional I/O port (Am295Ø) interfacing with the host control bus

To interface with the peripheral device:

- 2 bidirectional I/O ports (Am295Ø's) conveying status and command signals to and from the peripheral device
- 1 serial-to-parallel converter for the serial data stream to and from the peripheral device
- 1 scratchpad or buffer RAM

And, of course the Am29116

# Low Speed Disk Controller

$\overline{OE}$*-and $\overline{CE}$*-
control for peripheral devices

*$\overline{OE}$ = output enable
*$\overline{CE}$ = clock enable

block diagram of a minimal configuration of a disk controller

Intelligent Controller - Low Speed

Brief Description of Some of the Elements

Am291Ø Sequencer

- an address sequencer intended for controlling the sequence of the execution of microinstructions stored in microprogram memory
- capabilities:
  - fixed width of 12 address bits
  - simple sequential access
  - conditional branching to any microinstruction within its 12-bit 4Ø96-word address range
  - incorporates a 12-bit counter for loop control
  - provides a 5-level stack for microsubroutine linkage
- can accept an address from one of 4 sources:
  - its microprogram address counter
  - a direct input from an external source
  - its internal register
  - its stack
- executes 16 instructions as specified by 4 instruction inputs
- has 5 control pins to control branching, loading of its register, incrementing of its program counter and enabling its output bus drivers

Intelligent Controller - Low Speed (continued)

Brief Description of Some of the Elements (continued)

Am2920 Octal D-Type Flip-Flop

- provides eight edge-triggered D-type flip-flops with
  - a buffered common clock
  - a buffered common clock enable
  - a buffered common asynchronous clear input
  - three-state output control
- the clear input, $\overline{CLR}$, resets all eight flip-flops independent of all other inputs
- With the three-state output-enable LOW, all eight outputs appear as normal TTL outputs. Otherwise the outputs are in a high impedance state.
- The clock-enable input, $\overline{E}$, is used to selectively load data into the register. When $\overline{E}$ is HIGH the register will retain its current data. When $\overline{E}$ is LOW, new data is entered on the LOW-to-HIGH transition of the clock input.

Intelligent Controller - Low Speed (continued)

Brief Description of Some of the Elements (continued)

Am295Ø Eight-Bit Bidirectional I/O Port

- designed for use as a parallel data I/O port
- provides 2 back-to-back registers to store data moving in both directions between 2 bidirectional three-state busses
- provides a handshake-flag flip-flop for each data direction to allow coordination of demand-response data transfer:
  - Each flag flip-flop is set automatically when a register is loaded.
  - Each flag flip-flop has an edge-sensitive clear input.
- provides for each register:
  - a clock input
  - a clock enable
  - a three-state output enable

Intelligent Controller - Low Speed (continued)

Brief Description of Some of the Elements (continued)

The DMA Address Generator - Am2940

- High speed, cascadable, eight-bit wide Direct Memory Access address generator slice

- Generates sequential memory addresses for use in the sequential transfer of data to or from a memory

- Equipped with

  - address counter (increment/decrement)
  - address register (saves the initial address)
  - word counter (increment/decrement)
  - word counter register (terminal count)
  - three-state address ouput buffers

- 4 control modes

- 8 different instructions (3 instruction pins)

  - write/read control register (control mode)
  - read word/address counter
  - reinitialize counters
  - load address/word count
  - enable counters

- 3 control pins

Intelligent Controller - Low Speed (continued)

Brief Description of Some of the Elements (continued)

## The Scratch Pad Memory

- Not always necessary for a peripheral controller
- Improves system performance by allowing the host CPU and the peripheral device to respond at different rates.
- May also be useful in improving the execution speed of a controller algorithm by providing quick-access storage for variables or look-up tables. This traffic can thus be kept off of the main bus.

Intelligent Controller - Low Speed (continued)

Microword

- 16 instruction bits for the Am29116 that are shared with the 12 data bits to the Am291Ø that provide for loading the counter and providing branch addresses

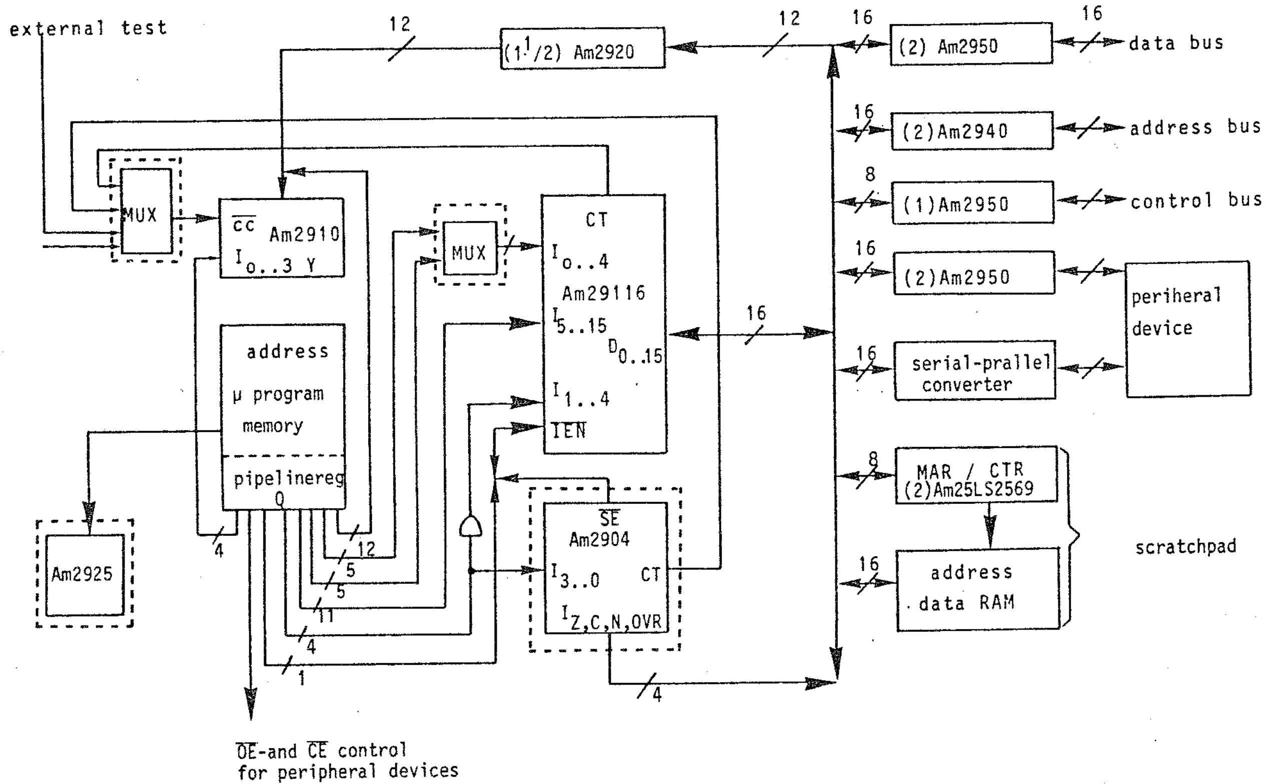
- 1 bit to determine the destination of the above lines (to either the Am29116 or the Am291Ø). Use the $\overline{\text{IEN}}$ input of the Am29116 to disable it on those microcycles when the data on the instruction lines is intended for the Am291Ø.

- 4 instruction bits for the Am291Ø

- 3 instruction bits for the Am294Ø

- Using this technique of sharing microinstruction fields, a microword of 28 bits is possible. Of course, there is a speed penalty caused by the need to halt the Am29116 when data is passed to the sequencer.

## Peripheral Controller with Maximal Performance




block diagram of a maximal configuration of a disk controller

Intelligent Controller - High Speed (continued)

Additional Elements

- 2 multiplexers:
  - to define different Am29116 source and destination registers
  - to select the branch condition for the Am2910
- Am2925 clock generator and microcycle length controller. For extended timing required by Am29116 two-address operation.
- Am2904 status and shift control unit:
  - to add more flexibility to the Am29116 status tests
  - use only the Am2904 micro status register and its condition code instructions

Intelligent Controller - High Speed (continued)

## Brief Description of the Additional Elements

### Am2925 Clock Generator and Microcycle Length Controller

- General purpose crystal-controlled clock generator/driver
- Has a microprogrammable clock cycle length:
  - provides significant speed-up over fixed clock cycle
  - meets a variety of system speed requirements
- Generates four different simultaneous clock output waveforms
- One of eight cycle lengths can be selected by the microprogram
- System control functions include:
  - run
  - halt
  - single-step
  - initialize
  - ready/wait
  - inputs can determine:
    - where a halt occurs
    - the end point timing of wait cycles
- Up to 12 pins can be controlled by the microword.

Intelligent Controller - High Speed (continued)

## Brief Description of the Additional Elements (continued)

### Am29Ø4 Status and Shift Control Unit

- Designed to perform all the miscellaneous functions which are usually performed in MSI around an ALU

- It contains three nearly independent blocks of logic:
  - multiplexer to generate the carry-in
  - 4 three-state multiplexers for shift linkage
  - 2 status registers for storing carry, overflow, zero and negative status flags.
    These status registers control a condition-test output via a condition code multiplexer.
    A wide selection of condition-code test logic is provided.

In our application only the status registers and condition-test logic is used.

Intelligent Controller - High Speed (continued)

Brief Description of the Additional Elements (continued)

Am29Ø4 Condition Code Output

| $I_3 - I_\emptyset$ | Condition Code Output | |
|---|---|---|
| Ø Ø Ø Ø | $(N \oplus OVR)+Z$ | |
| Ø Ø Ø 1 | $\overline{(N \oplus OVR)} \cdot \overline{Z}$ | * |
| Ø Ø 1 Ø | $N \oplus OVR$ | |
| Ø Ø 1 1 | $\overline{N \oplus OVR}$ | * |
| Ø 1 Ø Ø | $Z$ | |
| Ø 1 Ø 1 | $\overline{Z}$ | * |
| Ø 1 1 Ø | $OVR$ | |
| Ø 1 1 1 | $\overline{OVR}$ | * |
| 1 Ø Ø Ø | $C+Z$ | * |
| 1 Ø Ø 1 | $\overline{C} \cdot \overline{Z}$ | * |
| 1 Ø 1 Ø | $C$ | |
| 1 Ø 1 1 | $\overline{C}$ | * |
| 1 1 Ø Ø | $\overline{C}+Z$ | |
| 1 1 Ø 1 | $C+\overline{Z}$ | * |
| 1 1 1 Ø | $N$ | |
| 1 1 1 1 | $\overline{N}$ | * |

* 9 Condition codes not available at the Am29116 CT-output.

Intelligent Controller - High Speed (continued)

## Reasons for the Better Performance of the Second Solution

- Instruction inputs of the Am29116 and the $D_{\emptyset-11}$ inputs of the Am291∅ are driven from separate microcode bits.
  - allows simultaneous instruction execution in the Am29116 and direct-address branching in the Am291∅
  - requires an additional 12 bits in the microinstruction
- Multiplexer at the CC-input of the Am291∅ (controlled by 2 bits)
  - allows testing of conditions without loading the signals into the Am29116
- $T_{1-4}$ inputs of the Am29116 driven by 4 additional microword bits
  - allows simultaneous testing and execution of an Am29116 instruction
- Use of the Am29∅4 (controlled by 4 bits)
  - improved flexibility in status testing
- Multiplexer at the $I_{\emptyset-4}$ inputs of the Am29116 (needs 6 bits)
  - allows different source and destination addresses in RAM in the same microcycle
- The Am2925 clock generator/driver
  - able to dynamically alter the length of the microcycle

Intelligent Controller

Comparison with an Am29Ø1 - Based Solution

## Alternate Means of Obtaining Am29116 Features

| Am29116 | Am29Ø1-Based Solution |
|---|---|
| • 16-bit data path | • Use 4 Am29Ø1 microprocessor slices and 1 Am29Ø2 carry-look-ahead generator. |
| • 32 registers | • Am29Ø1 has 16 registers but is not readily expandable. For additional registers use Am29Ø3 or Am292Ø3 processor with 4 Am297Ø5 16 x 4-bit register file extenders. |
| • 16-bit barrel shifter | • Use n cycles to shift data by n bits. |

Alternate Means of Obtaining Am29116 Features (continued)

| Am29116 | Am29Ø1-Based Solution |
| --- | --- |
| • Status register and condition code generator/MUX | • Use 2 Am29Ø4 status and shift control units<br>- One Am29Ø4 for the ALU flags (C,N,OVR,Z) and condition code generation<br>- One Am29Ø4 for the 3 user-definable flags and the linkage flag |
| • Byte/Word Mode | • Use two output enable bits ($\overline{OE_Y}$). Then in byte operations force $\overline{OE_Y}$ high for slices 3 and 4.<br>• Multiplex status lines at slices 2 and 4 |

Note: This will permit external byte-mode functions. Operations on internal registers will still alter all 16-bits. You should choose Am292Ø3 rather than Am29Ø1 if Byte/Word operations are important. When used with a status-flag MUX, Am292Ø3 fully supports Byte/Word operations.

Alternate Means of Obtaining Am29116 Features (continued)

| Am29116 | Am29Ø1-Based Solution |
|---|---|
| • Built-in 16-bit priority encoder | • Use two 8-bit Am25LS2513 priority encoders |
| • Masking capability of the ALU | • Use three microcycles:<br>- mask the first operand<br>- mask the second operand<br>- operate on the masked operands |
| • Bit oriented instructions | • Emulate with several microinstructions |
| • CRC instructions | • Can be emulated by a lengthy microcode sequence. Will execute too slowly for many applications. |

# Intelligent Controller - Using Am2901

## Brief Description of the Additional Devices

### Am2901 4-bit Microprocessor Slice

- High-speed cascadable element for use in
  - CPU's
  - peripheral controllers
  - programmable microprocessors
  - numerous other applications
- Consists of:
  - a 16-word by 4-bit two-port RAM
  - a high-speed 8-function ALU plus shifting, decoding and multiplexing sections
- Cascadable with either:
  - simple ripple carry propagation
  - the Am2902 look-ahead carry generator
- Produces 4 status flags (N, OVR, C, Z)
- accepts 9-bit microinstructions:
  - 3 bits select the ALU operand source
  - 3 bits select the ALU function
  - 3 bits select the ALU destination
- accepts 8 bits to select the two RAM addresses
- accepts 2 control bits ($\overline{OE}$ and carry-in)

## Intelligent Controller - Using Am29∅1

### Brief Description of the Additional Devices

### Am29∅2 High-Speed Look-Ahead Carry Generator

- Provides anticipated carries across a group of four binary ALU's
- Accepts up to 4 pairs of carry Propagate and carry Generate signals from an ALU and one carry input.

With ALU operands A and B and an addition operation:

$$P = \overline{P_3 \cdot P_2 \cdot P_1 \cdot P_\emptyset} \qquad G = \overline{G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_\emptyset}$$

$$P_\emptyset = A_\emptyset + B_\emptyset \qquad G_\emptyset = A_\emptyset \cdot B_\emptyset$$

$$P_1 = A_1 + B_1 \qquad G_1 = A_1 \cdot B_1$$

$$P_2 = A_2 + B_2 \qquad G_2 = A_2 \cdot B_2$$

$$P_3 = A_3 + B_3 \qquad G_3 = A_3 \cdot B_3$$

- Also provides carry-propagate and carry-generate signals to use for further levels of look-ahead.
- Logic results provided at the outputs are:

$$C_{n+x} = \overline{G_\emptyset + P_\emptyset \cdot C_n}$$

$$C_{n+y} = \overline{G_1 + P_1 \cdot G_\emptyset + P_1 \cdot P_\emptyset \cdot C_n}$$

$$C_{n+z} = \overline{G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_\emptyset + P_2 \cdot P_1 \cdot P_\emptyset \cdot C_n}$$

$$\overline{G} = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_\emptyset$$

$$\overline{P} = P_3 \cdot P_2 \cdot P_1 \cdot P_\emptyset$$

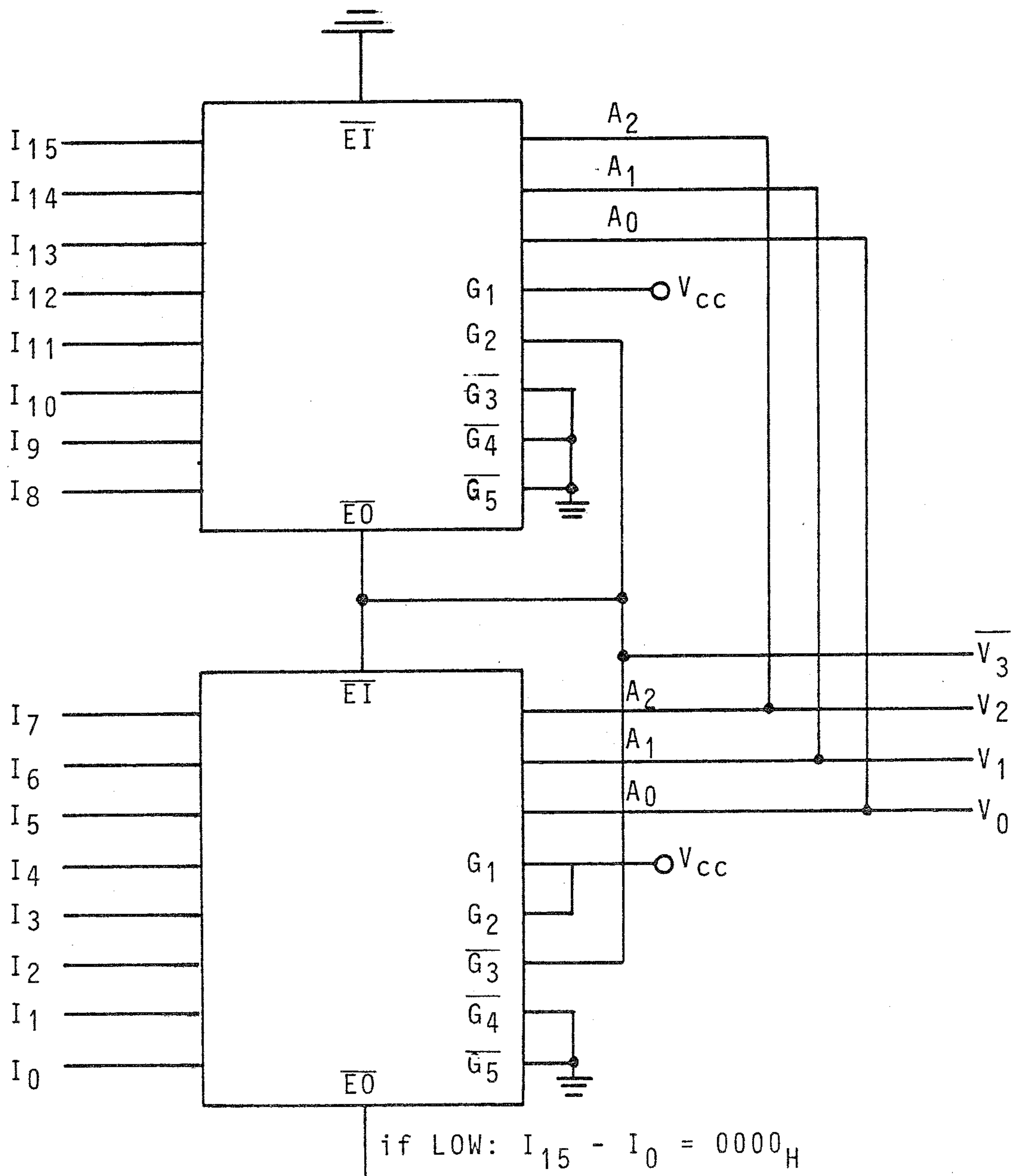
## Intelligent Controller - Using Am29Ø1

### Brief Description of the Additional Devices

### Am25LS2513 Three-State Priority Encoder

- Encodes eight lines to three-line binary
- Three-state outputs
  - controlled by three active LOW and two active HIGH inputs
- Cascadable
  - provides an input enable and an output enable to permit cascading without additional circuitry

Building a 16-bit Priority Encoder from 2 AM25LS2513

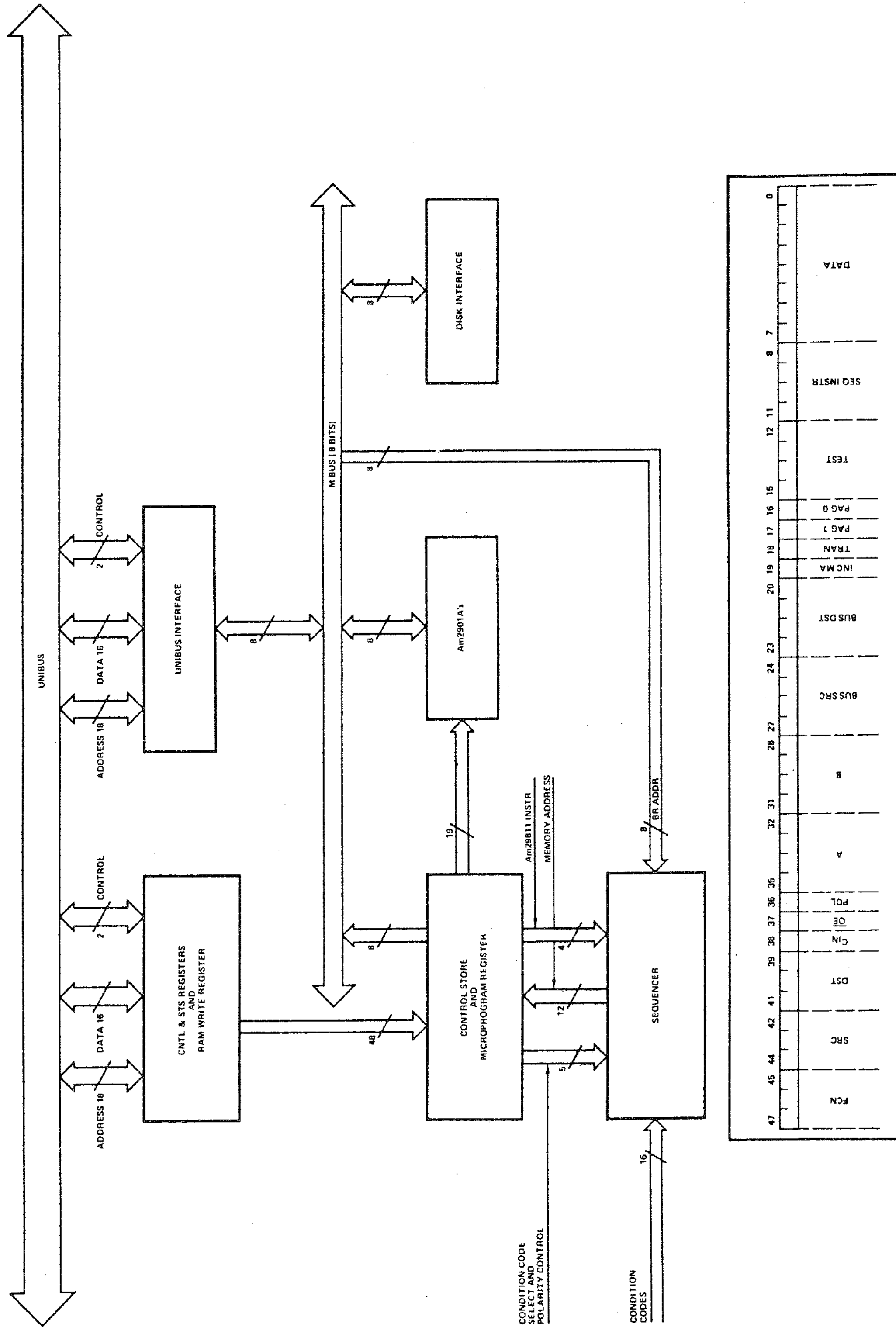


Intelligent Controller - Using Am2901

Thus we have seen that it is rather hard to build a controller with the same features as the AM29116 from parts with a lower level of integration.

On the next page we see a simple disk controller based on a pair of Am2901A's.

## A Disk Controller built around Am2901's



## The Microword for the 2901 - Based Solution

- 48 bits wide
  - . 19 bits Am2901 instruction (M47 - M37, M35 - M28)
    - 3 bits function select (FCN)
    - 3 bits source (SRC)
    - 3 bits destination (DST)
    - 4 bits A port register address
    - 4 bits B port register address
    - 1 bit carry in ($C_N$)
    - 1 bit output enable (OE)
  - . 8 bits M-bus control (M27 - M20)
    - 4 bits bus source (BUS SRC)
    - 4 bits bus destination (BUS DST)
  - . 11 bits sequencer (M36, M17 - M8)
    - 4 bits Am2911 instruction (SEQ INSTR)
    - 4 bits condition code select (TEST)
    - 1 bit polarity of CC (POL)
    - 2 bits to determine the page of microprogram memory
  - . 8 bits data (M7 - M0)
  - . 2 bits additional control (M19, M18)
    - 1 bit increments MAR (INC MA)
    - 1 bit initiate data transfer (TRAN)

Intelligent Controller

Very High Speed Solution

## Intelligent Controller - Very High Speed

### Some General Considerations

- Interface-signal names, polarities and functions used here are similar to those used in the current ANSI standard for hard-disk drives.

- The methods and functions discussed here can be used for most current hard- or flexible-disk drives.

- With minimal external logic, this controller uses an Am29116 and an Am952Ø burst error processor to perform all of the functions needed to:

  - write and read at 3Ø Mbits per second

  - format a disk

  Including:

  - searching a track for a specific header and sector

  - managing data flow through a high-speed buffer memory

  - generating modified Fire-code check bits while writing

  - detecting and correcting single and burst errors on reading

  - generating and checking of CRC's in sector headers

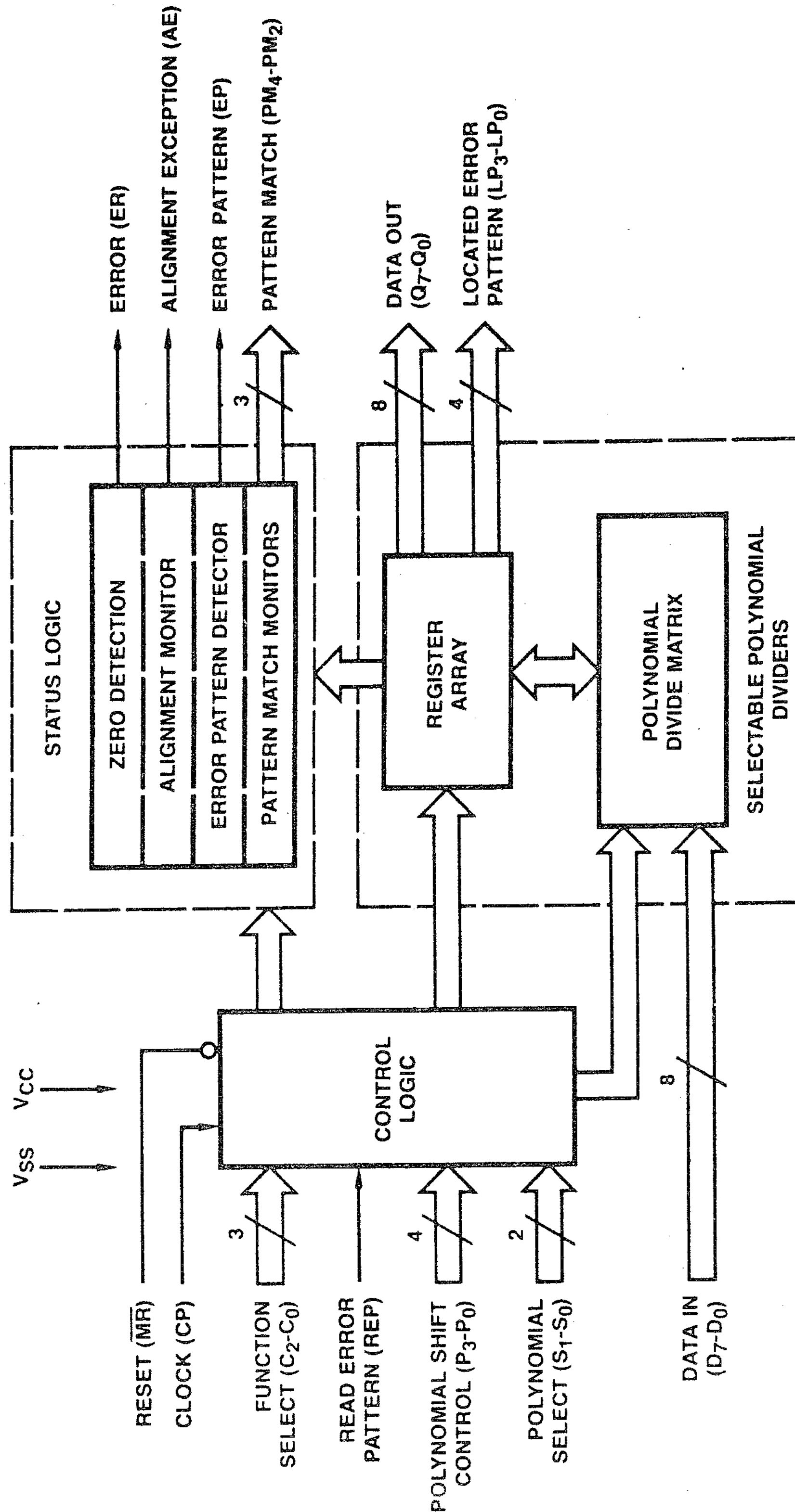
Intelligent Controller - Very High Speed (continued)

Am9520 Burst Error Processor

Distinctive Characteristics:

- Provides for detection and correction of burst errors:
  - detects errors in serial-data blocks up to 585K bits long
  - allows correction of error bursts of up to 12 bits long
- Effective data rates up to 20 Mbits per second:
  - fast enough for high-performance hard disk systems
- Four selectable industry-standard polynomials:
  - popular IBM 56- and 48-bit polynomials
  - also 35-bit and 32-bit polynomials
- Three correction algorithms provide flexibility:
  - full-period clock-around method compatible with current practice
  - Chinese Remainder Theorem method reduces correction time by orders of magnitude
  - reciprocal polynomial method for correction with the 48-bit IBM code
- Designed for use in disk controllers and communication systems based on fixed-instruction-set or microprogrammed processors.

## Am9520 Burst Error Processor



Intelligent Controller - Very High Speed (continued)

Am952Ø Burst Error Processor (continued)

Functional Description:

- Register Array
  - consists of 56 flip-flops used for
    - . check-bit computation during write operations
    - . syndrome computation during read operations
    - . error pattern extraction during error correction operations

- Polynomial-Divide Matrix:
  - establishes interconnections and feedback for a group of shift registers such that an entire byte of data is divided in a parallel operation by the selected polynomial
  - the matrix is controlled by 2 Polynomial Select and 2 Function Select inputs
  - the data is presented to the matrix a byte at a time on 8 data lines

- When correction operations are complete, the error pattern is available on 12 outputs:
  - eight bits on the $Q_{\emptyset}$-$Q_7$ outputs
  - four bits on the $LP_{\emptyset}$-$LP_3$ outputs

Intelligent Controller - Very High Speed (continued)

Am9520 Burst Error Processor (continued)

- Write:
  - While the data is being written, the Am9520 is in the Compute-Check-Bits mode, calculating the polynomial remainder without affecting the flow of data to the disk
  - After the last data byte, the Am9520 is switched into the Write-Check-Bits mode, outputting the 4, 5, 6 or 7 check bytes.
  - These check bytes constitute additional information to be appended to the data stream to allow detection and correction of errors on reading.
- Read:
  - Two modes are available when reading: Normal and High Speed
  - The two modes use different correction algorithms.
  - After the last information byte has been read, the state of the ER output signal indicates whether an error has occured.

Intelligent Controller - Very High Speed (continued)

Am9520 Burst Error Processor (continued)

- Correction:
  - After the read operation, the syndrome in the register array contains information specifying:
    - the location of the error
    - the bit pattern of the error

- Normal Correction Mode:
  - The error location is found by counting the number of clock pulses required to make the EP output go HIGH.
  - The error pattern available on $LP_0$-$LP_3$ and $Q_0$-$Q_7$ can be Exclusive ORed with the data to effect the correction.

- High-Speed Correction Mode:
  - The error location is found by counting the number of clock pulses required to generate an indicator for each of the 2 or 4 factors of the polynomial.
  - The error pattern available on $LP_0$-$LP_3$ and $Q_0$-$Q_7$ can be Exclusive ORed with the data to effect the correction.
  - While there are more steps to the high-speed-correction procedure than are required by the "Normal" procedure, the correction is accomplished far faster.
  - The high-speed correction method is not available for the 48-bit polynomial.
  - For the 56-, 35- and 32-bit polynomials, the high-speed method should be preferred over the "normal method".

Intelligent Controller - Very High Speed (continued)

Am952Ø Burst Error Processor (continued)

Computing Check Bits:

- The Polynomial Divide Matrix and the Register Array implement the familiar serial form of feedback shift register arrangement in an 8-bit parallel form.

- The $S_1$ and $S_Ø$ inputs select one of 4 polynomials:

| Polynomial | Degree & Number of Check Bits | Period (Bits) | Correctable Burst Error Length (Bits) |
|---|---|---|---|
| $(x^{22}+1)\cdot(x^{11}+x^7+x^6+x+1)\cdot(x^{12}+x^{11}+x^{1Ø}+ \ldots +x+1)\cdot(x^{11}+x^9+x^7+x^6+x^5+x+1)$ | 56 | 585,442 | 11 |
| $(x^{21}+1)\cdot(x^{11}+x^2+1)$ | 32 | 42,987 | 11 |
| $(x^{23}+1)\cdot(x^{12}+x^{11}+x^8+x^7+x^3+x+1)$ | 35 | 94,185 | 12 |
| $(x^{13}+1)\cdot(x^{35}+x^{23}+x^8+x^2+1)$ | 48 | $13\cdot(2^{35}-1)$ $=4.466\ldots x1Ø^{11}$ | 7 |

- When the last data byte has been read or written, the Register Array contains the check bits.

- Remember to select the same polynomial for reading as was used for writing.

Intelligent Controller - Very High Speed (continued)

Am952Ø Burst Error Processor (continued)

Computing Check Bits (continued)

- Sequence of events to compute the check bits

  i) The clock input,CP, should be in the quiescent HIGH state.

  ii) Initialize by activating the master reset input, $\overline{MR}$, LOW and return it to HIGH.

  iii) Specify the desired polynomial via $S_Ø$, $S_1$.
  Apply zeroes to $C_2$-$C_Ø$ to select Compute-Check-Bits mode.

  iv) Establish a byte of data on $D_Ø$-$D_7$ inputs.

  v) Make the clock input go LOW then HIGH.

  vi) Repeat from step iv) until all data bytes are entered.

Intelligent Controller - Very High Speed (continued)

Am952Ø Burst Error Processor (continued)

Write Check Bits:

- When the Write-Check-Bits mode is established, the feedback paths of the register array are disabled and the check bits may be shifted out.

- Checkbits are available on the $Q_{\emptyset}$-$Q_7$ outputs one byte at a time.

- Sequence of events to obtain the check bits:

  i) The clock input, CP, should be in the quiescent HIGH state.

  ii) Select the polynomial via the $S_{\emptyset}$,$S_1$ inputs.

  iii) Force $C_2$,$C_1$,$C_{\emptyset}$ to LOW LOW HIGH (Write Check Bits).

  iv) After a propagation delay the $Q_{\emptyset}$-$Q_7$ outputs will contain the first check byte.

  v) Make the clock input go LOW then HIGH. The next check byte will be available on the $Q_{\emptyset}$-$Q_7$ outputs.

  vi) Repeat from step v) until all check bytes are read out.

Intelligent Controller - Very High Speed (continued)

Am9520 Burst Error Processor (continued)

Read - General:

- The input stream (data and check bytes) is divided by the selected polynomial to obtain the syndrome.

- A non-zero syndrome indicates an error has been detected. If the syndrome is not zero the ER output will be HIGH.

- Two methods for error correction are available:

  - full-period clock-around ("Normal")

  - Chinese Remainder Theorem ("High Speed Method")

- There is a different read procedure for each of these methods:

  - Read Normal: produces one syndrome

  - Read High Speed: produces as many syndromes as the number of factors in the polynomial.

    The input stream is simultaneously divided by all of the factors of the polynomial. The ER output indicates whether or not all syndromes are zero.

- Read High Speed is not available for the 48-bit polynomial

Intelligent Controller - Very High Speed (continued)

Am9520 Burst Error Processor (continued)

Read Normal:

- Sequence of events for Read Normal:

  i) The clock input should be in the quiescent HIGH state.

  ii) Initialize by pulling the master reset, $\overline{MR}$, LOW and then returning it to HIGH.

  iii) Select the required polynomial via $S_0$,$S_1$ inputs.

  iv) Apply LHL to $C_2$,$C_1$,$C_0$ to select Read Normal.

  v) Present a byte of information as read from the disk to the $D_0$-$D_7$ inputs.

  vi) Make the clock go LOW then HIGH.

  vii) Repeat from step v) until the last check byte read from the disk is processed.

  viii) Test the ER output:

  - ER HIGH: an error has been detected.

  - ER LOW: no error has been detected.

Intelligent Controller - Very High Speed (continued)

Am952Ø Burst Error Processor (continued)

High Speed Read:

- Sequence of events for Read High Speed:

  i) The clock input, CP, should be in the quiescent HIGH state.

  ii) Specify the polynomial via the $S_Ø$,$S_1$ inputs.

  iii) Apply LHH to $C_2$,$C_1$,$C_Ø$ to select Read High Speed.

  iv) Initialize by pulling the master reset, $\overline{MR}$, LOW and then returning it to HIGH.

  v) Present a byte as read from the disk to the $D_Ø$-$D_7$ inputs.

  vi) Make the clock go LOW then HIGH.

  vii) Repeat from step v) until the last check byte has been read.

  viii) Test the ER output:

  - ER HIGH: an error has been detected.
  - ER LOW: no error has been detected.

Intelligent Controller - Very High Speed (continued)

Am9520 Burst Error Processor (continued)

Function Select Codes

| $C_2$ | $C_1$ | $C_0$ | Function |
|---|---|---|---|
| L | L | L | Compute check bits |
| L | L | H | Write check bits |
| L | H | L | Read normal |
| L | H | H | Read high speed |
| H | L | L | Load |
| H | L | H | Reserved |
| H | H | L | Correct normal |
| H | H | H | Correct high speed |

Intelligent Controller - Very High Speed (continued)

Am952Ø Burst Error Processor (continued)

Correct Normal:

- The Am952Ø manipulates the syndrome to yield:
  - error pattern (at $Q_Ø$-$Q_7$ and $LP_Ø$-$LP_3$ outputs)
  - error location (needs further external computation)
- Syndrome is repeatedly divided by the polynomial until the error pattern is located:
  - done by repeatedly clocking without regard to the $D_Ø$-$D_7$ inputs until EP goes HIGH
- If the AE output goes HIGH while the EP output remains LOW, an alignment exception has been detected.
- Count clock cycles (#C) until EP goes HIGH.
- If #C > (period of polynomial) then the error is uncorrectable.

Intelligent Controller - Very High Speed (continued)

Am9520 Burst Error Processor (continued)

Correct Normal: (continued)

- Use two external counters (R1, R2)
  - R1 counts the number of cycles until AE goes HIGH
  - R2 counts the number of cycles from AE going HIGH to EP going HIGH. (= 0 if no alignment exception)
- If R1+R2 > (period of polynomial) then the error is uncorrectable.
- (N·K - 8·R1 - R2) gives the location of the first bit in the error burst counting from the last check bit of the record for the 56-bit and 32-bit polynomials.

  (N·K - 8·R1 - R2+5) gives the location of the first bit in the error burst counting from the last check bit of the record for the 35-bit polynomial.

  where:
  K is the smallest +ve integer that makes this expression +ve.
  and N is the period of the polynomial.

Note: The 48-bit polynomial uses another correction algorithm. See specification sheet for details.

Intelligent Controller - Very High Speed (continued)

Am952Ø Burst Error Processor (continued)

Burst Error Processor (continued)

Correct High Speed:

- This mode allows you to determine the error pattern in far fewer clock cycles than does the "Normal" mode.
  - A polynomial with m factors with periods $P_1, P_2, \ldots, P_m$ will correct in no more than the following number of of clock cycles:
    - . In Normal mode: $P_1 \cdot P_2 \cdot \ldots \cdot P_m$ i.e. the product of the P's
    - . In High Speed mode: $P_1 + P_2 + \ldots + P_m$ i.e. the sum of the P's
- Number of syndromes equals the number of factors of the polynomial.
- Refer to the table a few pages back which shows the factorization of the polynomials.

As we have written the factors, the first factor has a special signifigance. The degree of this first factor determines the maximum length of an error burst that is still correctable.

Intelligent Controller - Very High Speed (continued)

Am952∅ Burst Error Processor (continued)

- The error location is given by:

$$L = N \cdot K - (A1M1 + A2M2 + \ldots + A_m M_m)$$

Where N is the natural period of the polynomial.
K is the smallest integer that makes L positive.
$M_i$ are the numbers of clock cycles required to match the error pattern of each factor.
and $A_i$ are the Chinese Remainder Theorem coefficients:

| Polynomials | A1 | A2 | A3 | A4 |
|---|---|---|---|---|
| 56-bit | 452,387 | 578,864 | 2,521,9∅4 | 2,647,216 |
| 32-bit | 311,144 | 32,76∅ | -- | -- |
| 35-bit | 32,76∅ | 72∅,728 | -- | -- |

Intelligent Controller - Very High Speed (continued)

Am952Ø Burst Error Processor (continued)

Correct High Speed (continued)

- To determine $M_i$ and the error pattern use:
  - $P_Ø$-$P_3$ inputs to select the register section to be clocked
  - $PM_2$-$PM_4$, the "pattern match" outputs:

    When $PM_i$ is HIGH then the $syndrome_i$ shows a match with the error pattern.

- For M1: use the polynomial-shift controls to select register-section #1 ($P_Ø$,$P_1$,$P_2$,$P_3$ - HLLL).

  Use the same procedure as in the correct normal mode.

  - If R1+R2 > (period of factor #1) then error is uncorrectable.
  - If error is correctable then M1 = 8R1 + R2

- The $M_i$ will be determined after M1 but slightly differently:
  - Select the $i^{th}$ register section via $P_3$-$P_Ø$
  - While $PM_i$ is LOW clock the 952Ø and increment a counter
  - If the count exceeds the period of this factor the error is not correctable.

- When EP and all $PM_i$'s associated with this polynomial are HIGH, then the error pattern and error location are determined.

Intelligent Controller - Very High Speed (continued)

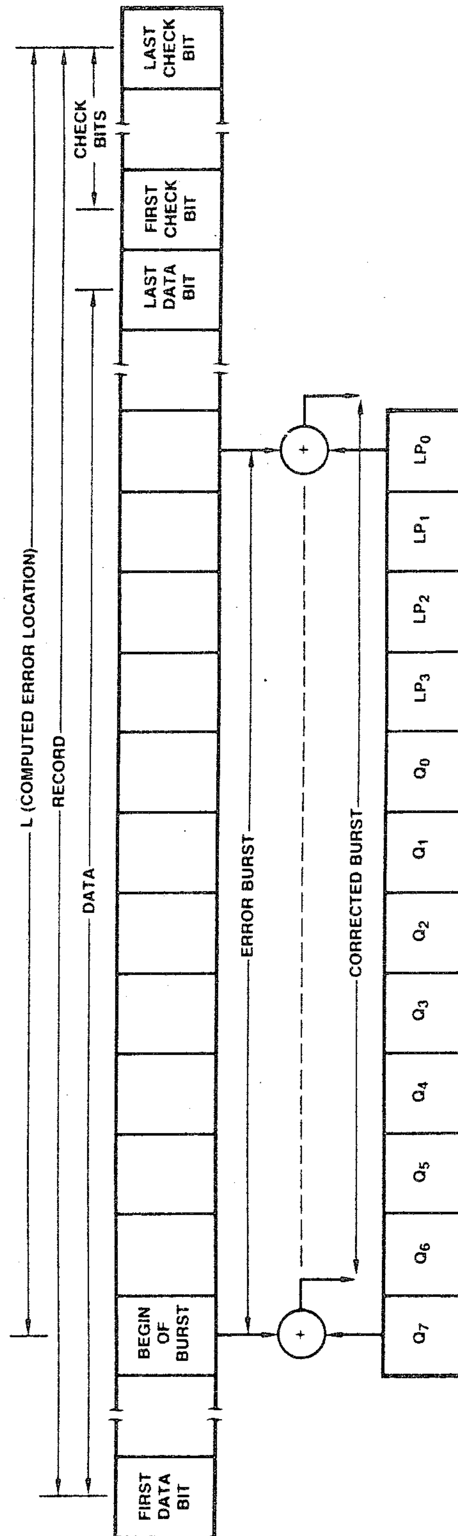
Am952Ø Burst Error Processor (continued)

Am952Ø Polynomial Periods

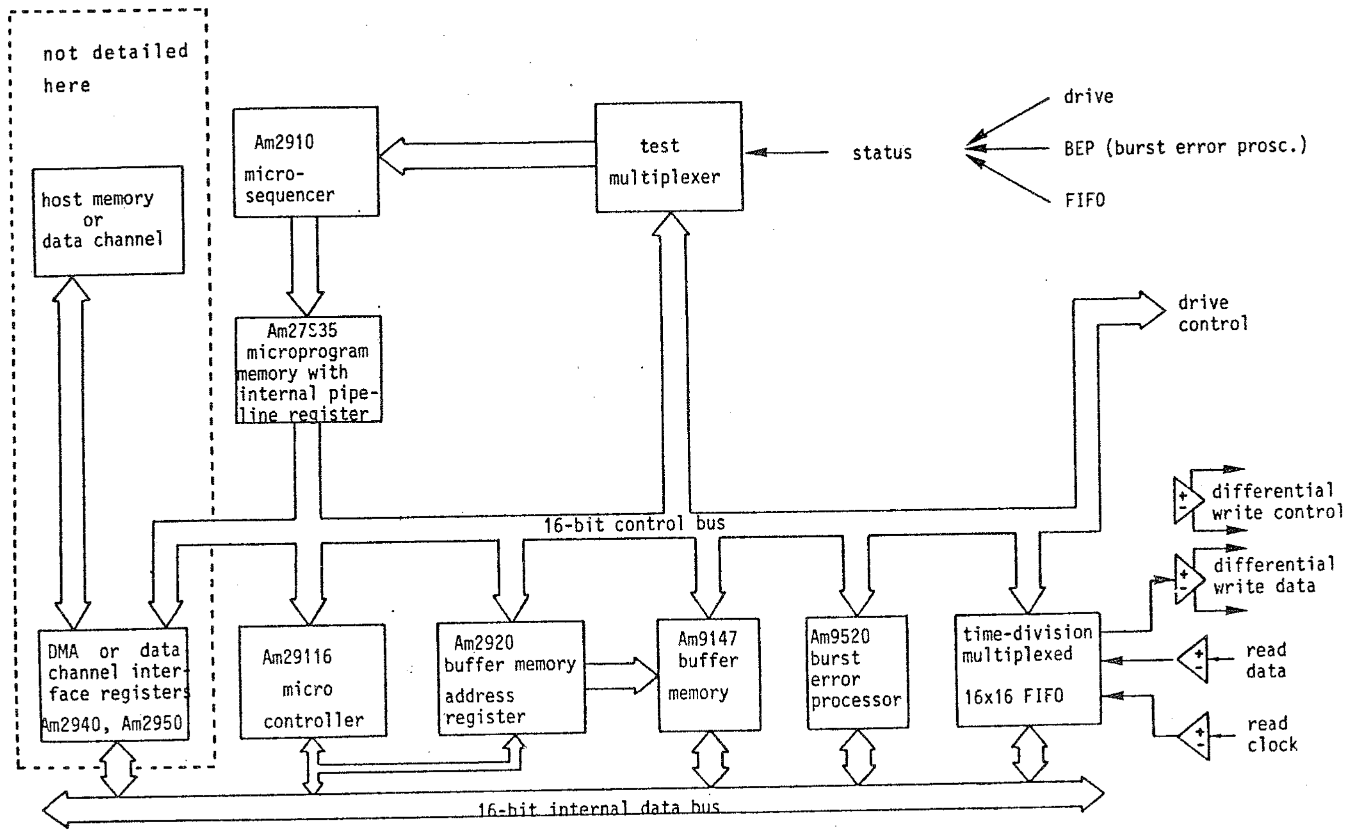
| Polynomial | Period Factor 1 | Period Factor 2 | Period Factor 3 | Period Factor 4 | Composite Period (N) |
|---|---|---|---|---|---|
| 56-bit | 22 | 13 | 89 | 23 | 585,442 |
| 32-bit | 21 | 2Ø47 | -- | -- | 42,987 |
| 35-bit | 23 | 4Ø95 | -- | -- | 94,185 |

Note: The 48-bit polynomial requires the use of a different correction procedure and is not shown here.

Error Pattern Format for 56-Bit, 35-Bit and 32-Bit Polynomials



Data Path for Very High Speed Controller



Intelligent Controller - Very High Speed (continued)

System Organisation:

- Interface to the disk drives:
  - bit-serial data paths for read and write
  - byte-parallel paths for
    - . commands
    - . disk addresses
    - . disk status

- Am291∅ sequencer and Am27S35 registered PROM microprogram memory drives 76-bit control bus:
- Data flow is:
  - asynchronous & serial at 3∅ Mbits/sec from drive to a FIFO array
  - 16-bit clocked & parallel from the FIFO array to the buffer memory at 2∅ Mbits per second via the internal data bus.
- The BEP is located on the internal data bus.
- Disk read:
  - Data is read into the FIFO array at 3∅ Mbit per second.
  - Concurrently, the data is transferred from the buffer memory to the BEP at a rate of 15 MHz.
- Disk write:
  - The BEP pre-calculates the check bits before the write. There is not enough time to overlap BEP/buffer and buffer/FIFO transfers.

Intelligent Controller - Very High Speed (continued)

## Microinstruction Format

- 8Ø-bit wide microinstruction
  - This is not a minimum width, but it demonstrates microcoding in a straightforward manner.
- Sample microcode for uncompressed sector read/write operations is available including:
  - header and sector searching
  - error checking of the header via Am29116 CRC instructions
  - error checking and correction of the data segments via Am952Ø and its 56-bit modified Fire code polynomial
- The sector input/output microroutine (SECTIO) performs input or output of a single 256-byte sector.

## Microinstruction for the Very High Speed Controller

Microinstruction Field

| Bits | Width | Mnemonic | Comment |
|---|---|---|---|
| 79-64 | 16 | $I_{15}-I_{\emptyset}$ | Am29116 Instruction |
| 63-6Ø | 4 | $T_4-T_1$ | Am29116 Conditional Test Select |
| 59 | 1 | $\overline{SRE}$ | Am29116 Status Register Enable |
| 58 | 1 | $\overline{OE}_Y$ | Am29116 Output Enable Y-Bus |
| 57 | 1 | $\overline{IEN}$ | Am29116 Instruction Enable |
| 56 | 1 | DLE | Am29116 Data Latch Enable |
| 55-52 | 4 | $I_3-I_{\emptyset}$ | Am291Ø Instruction |
| 51-42 | 1Ø | $D_9-D_{\emptyset}$ | Am291Ø Direct Input |
| 41-36 | 6 | --- | Condition-Code MUX Selection & Polarity |
| 35 | 1 | $\overline{ADMC}$ | Address Mark Control |
| 34 | 1 | $\overline{BFCB}$ | (Enable Memory) Bus from (Disk Drive) Control Bus |
| 33 | 1 | $\overline{BFTP}$ | (Enable Memory) Bus from Translate PROM (Translate PROM needed in data compression operations) |
| 32 | 1 | $\overline{BF\emptyset 3}$ | (Enable Memory) Bus from 94Ø3A FIFO array |

Microinstruction for the Very High Speed Controller (continued)

| Microinstruction Field | | | |
|---|---|---|---|
| Bits | Width | Mnemonic | Comment |
| 31 | 1 | $\overline{BF16}$ | (Enable Memory) Bus from Am29116 Y-Bus |
| 3Ø | 1 | $\overline{BF2L}$ | (Enable Memory) Bus Lower Byte from Am952Ø Q-Bus |
| 29 | 1 | $\overline{BF2U}$ | (Enable Memory) Bus Upper Byte from Am952Ø Q-Bus |
| 28 | 1 | $\overline{BOUT}$ | Bus Direction OUT |
| 27 | 1 | $\overline{BTØ3}$ | (Enable Memory) Bus to 94Ø3A FIFO Array |
| 26 | 1 | $\overline{BT16}$ | (Enable Memory) Bus to Am29116 Y-Bus |
| 25 | 1 | $\overline{BT2L}$ | (Enable Memory) Bus Lower Byte to Am952Ø D-Bus |
| 24 | 1 | $\overline{BT2U}$ | (Enable Memory) Bus Upper Byte to Am952Ø D-Bus |
| 23 | 1 | $\overline{BT2Ø}$ | (Enable Memory) Bus to Am952Ø REP, P3-PØ, C2-CØ |
| 22 | 1 | $\overline{CE2L}$ | Clock Enable Am952Ø to Lower-Byte Bus Interface Register |
| 21 | 1 | $\overline{CE2Ø}$ | Clock Enable Memory Bus to Am952Ø Interface Registers |
| 2Ø | 1 | $\overline{CP2Ø}$ | Clock Pulse for 952Ø (Microcoded Waveform) |
| 19 | 1 | $\overline{CREQ}$ | Command Request |
| 18 | 1 | $\overline{INPT}$ | (Enable Serial Data) Input to 94Ø3A FIFO Array |

## Microinstruction for the Very High Speed Controller (continued)

Microinstruction Field

| Bits | Width | Mnemonic | Comment |
|---|---|---|---|
| 17-16 | 2 | $\overline{\text{JMPI}}$ | (Enable) Jump Indirect Am29116 Y-Bus (double rail) |
| 15 | 1 | $\overline{\text{MADR}}$ | (Enable Loading of Buffer) Memory Address Register |
| 14 | 1 | $\overline{\text{MREA}}$ | (Enable Buffer) Memory Read |
| 13 | 1 | $\overline{\text{MWRT}}$ | (Enable) Memory Write Operation |
| 12 | 1 | $\overline{\text{OUPT}}$ | (Enable Serial Data) Output from 94Ø3A FIFO array |
| 11 | 1 | $\overline{\text{PENB}}$ | Parameter Enable |
| 1Ø | 1 | $\overline{\text{PFPM}}$ | (Enable Setting of Am952Ø) P Bits from Am952Ø PM Bits |
| 9 | 1 | $\overline{\text{PFØ3}}$ | (Enable) Parallel Fetch from 94Ø3A FIFO array |
| 8 | 1 | $\overline{\text{PLØ3}}$ | (Enable) Parallel Load of 94Ø3A FIFO array |
| 7 | 1 | $\overline{\text{PREQ}}$ | Parameter Request |
| 6 | 1 | $\overline{\text{RDGA}}$ | Read Gate |
| 5 | 1 | $\overline{\text{RFIF}}$ | Reset 94Ø3A FIFO array |
| 4 | 1 | $\overline{\text{SAST}}$ | Select/Attention Strobe |
| 3 | 1 | $\overline{\text{WRGA}}$ | Write Gate |
| 2-Ø | 3 | XLAT | Translate Table Select for Data Compression PROM |

Intelligent Controller - Very High Speed (continued)

## Conclusion

What makes this controller so fast?

- The Am29116 microprocessor has been combined with the Am952Ø burst error processor.

  - This provides the powerful Am29116 instruction set and the very effective hardware elements of the Am29116:

    . its CRC logic
    . 32 registers
    . barrel shifter
    . priority encoder

  - The Am952Ø is a relevant specialized device which:

    . generates the checksum

    . checks the data together with the checksum and produces both the error pattern and error location

    . is very fast (due to its special hardware).

    You can calculate a CRC remainder -

    on an Am952Ø at 5Ønsec per data byte.
    on an Am29116 at 2ØØnsec per data bit.

    (The Am952Ø is 32x faster than the Am29116!).

- The Am952Ø is unique in supporting the very fast Chinese Remainder Theorem method that greatly speeds the correction of a faulty sector.

  This method is fully supported by hardware for use while reading and in the correction calculation itself.

Intelligent Controller - Very High Speed (continued)

Conclusion (continued)

- We can write special microcode: for example for packing ASCII fields.
  This is not possible with a fixed-instruction-set processor or a specialized LSI disk-controller chip.

- We are making use of the parallel hardware of the 9520.
  For example, the Am9520 can perform up to four simultaneous polynomial division operations and produce four independant syndromes concurrently.

- We are concurrently reading from the disk and generating the polynomial remainders for error detection and correction.

  (We were not able to support checkbit calculation in parallel with writing, however).

- We have used a FIFO array to allow us to read a data stream that is faster than even an Am9520 can check it.

- We have incorporated a buffer memory that:

  - holds images of the last eight sectors read from or written to disk

  - holds I/O request queues to maximize throughput

  - holds additional house keeping tables

- We have used a PROM to translate from EBCDIC data to packed-BCD or ASCII

# CHAPTER 6

Application of Am29116 to General Purpose CPUs

A Microprogrammed CPU Using Am29116

The following pages provide an introduction to
AMD Application Note MPR-1712

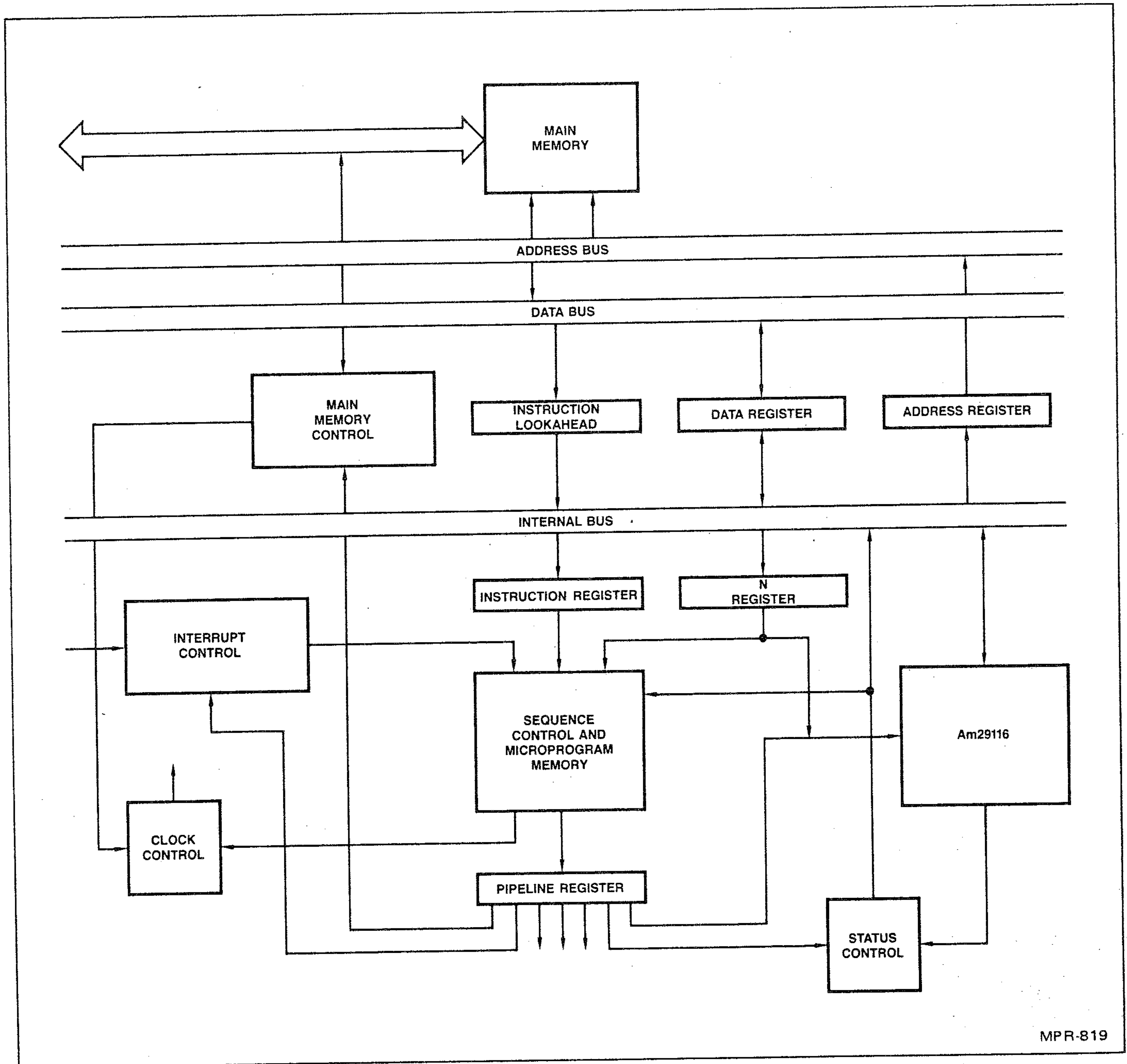
# A Microprogrammed CPU Using Am29116

## Introduction and System Overview

- Am29116 is optimized for peripheral controller applications
- However, Am29116 is also an ideal choice for CPU's as well.
  - it has a powerful instruction set for:
    - . arithmetic operations
    - . data movement
    - . multiple-bit shifts
    - . bit manipulation
    - . status manipulation
  - it has high speed (1ØØ nsec cycle time)
  - it can reduce power requirements
  - it save area on PC boards
- We will describe a CPU built with the Am29116 which maintains architectural and software compatibility with the Super-16.*
- This CPU incorporates pipelining at
  - the microprogram level
  - the macroinstruction level

* As described in detail in
"Bitslice Microprocessor Design" by Mick and Brick, Chapter 9

Central Processing Unit Block Diagram



A Microprogrammed CPU Using Am29116 (continued)

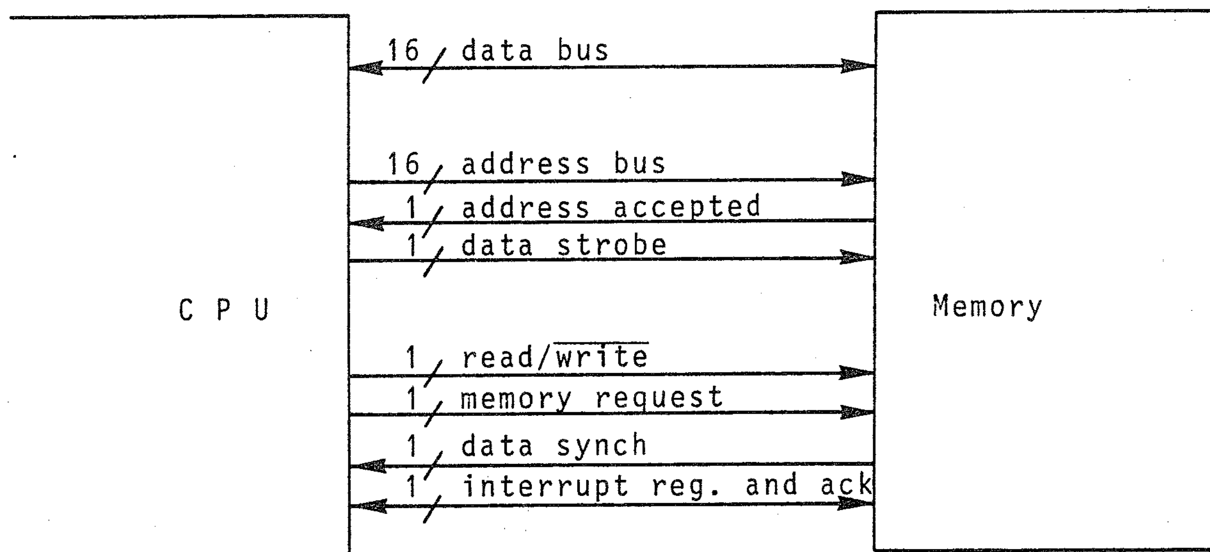
System Organization

This is a simple system comprised of:

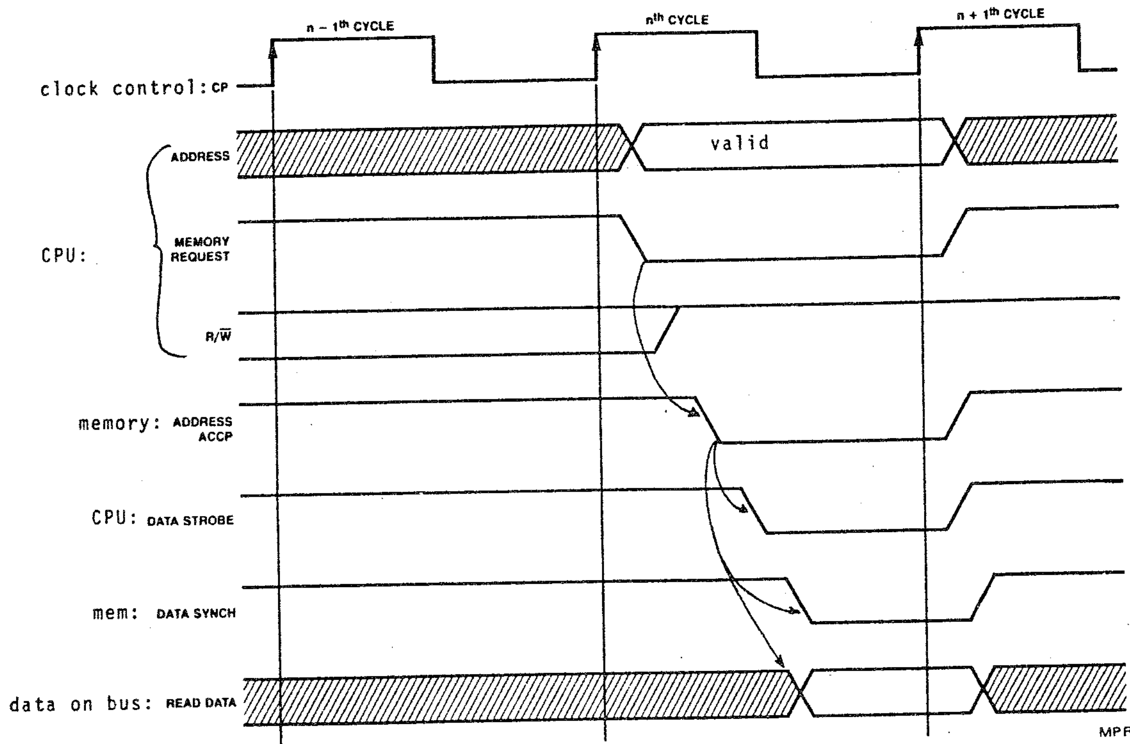
- 16-bits wide main memory built from static RAM chips
- Am29116 processor and CCU
- A simple bus structure:
  - can be modified to accomodate interface signals
  - but to add other I/Ø devices, a bus controller is needed

System Organization (continued)

Interface Signals between the Memory and the CPU




- handshaking over three busses
  - 16-bit-wide address bus
  - 16-bit-wide bidirectional data bus
  - 7-bit-wide control bus
    - memory request
    - read/$\overline{\text{write}}$
    - address accepted
    - data strobe
    - data synch
    - interrupt control lines


n – 1th CYCLE
nth CYCLE
n + 1th CYCLE
clock control: CP
ADDRESS
valid
CPU:
MEMORY REQUEST
R/W
memory: ADDRESS ACCP
CPU: DATA STROBE
mem: DATA SYNCH
data on bus: READ DATA
MPR-821

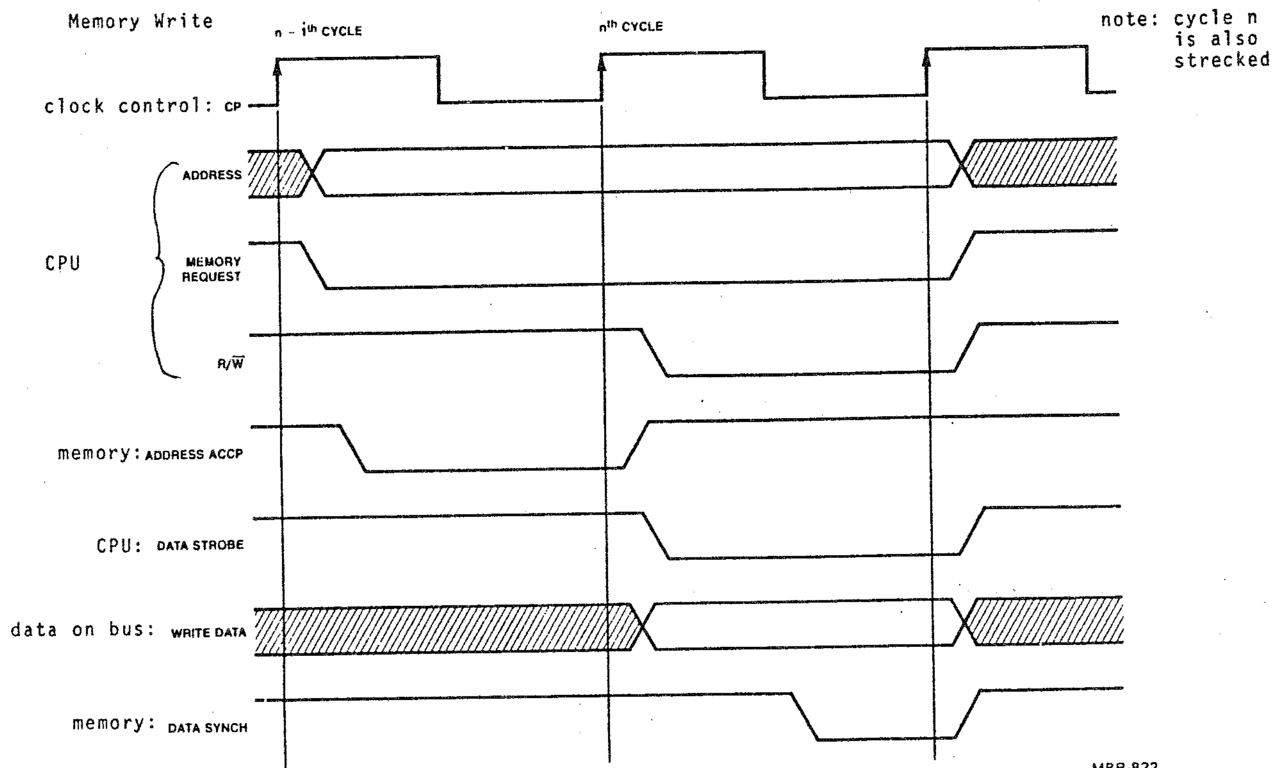
A Microprogrammed CPU Using Am29116 (continued)

Memory Read (continued)

- To use the data in the n+1$^{th}$ cycle, the Am29116 generates the main memory address during the (n-1)$^{th}$ cycle.
- Data is read during the n$^{th}$ cycle.
- The n$^{th}$ cycle must be stretched to accomodate main-memory READ timing.
- The signal to stretch the n$^{th}$ cycle is provided to the Am2925 clock generator during the (n-1)$^{th}$ cycle.

System Organization (continued)

Memory Write

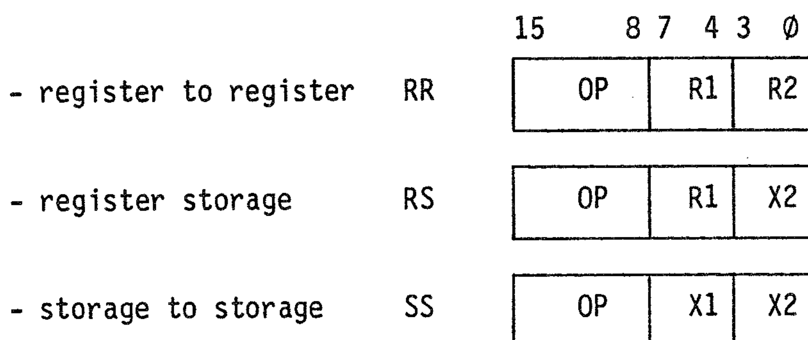


A Microprogrammed CPU Using Am29116 (continued)

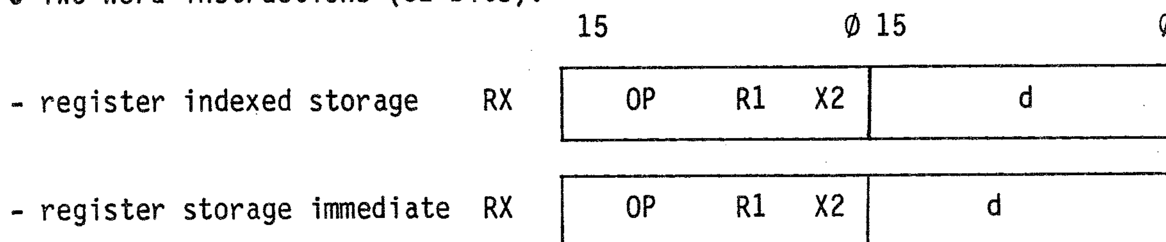
Instruction Formats

(same as for Super-16)

- One-word instructions (16 bits):




- Two-word instructions (32 bits):




- 4-bit register address defines one of 16 registers:

- uses lower half (RØ - R15) of the 32 registers in the Am29116 as user registers

- upper half (R16 - R31) used by the operating system (stack pointer, counter, etc.)

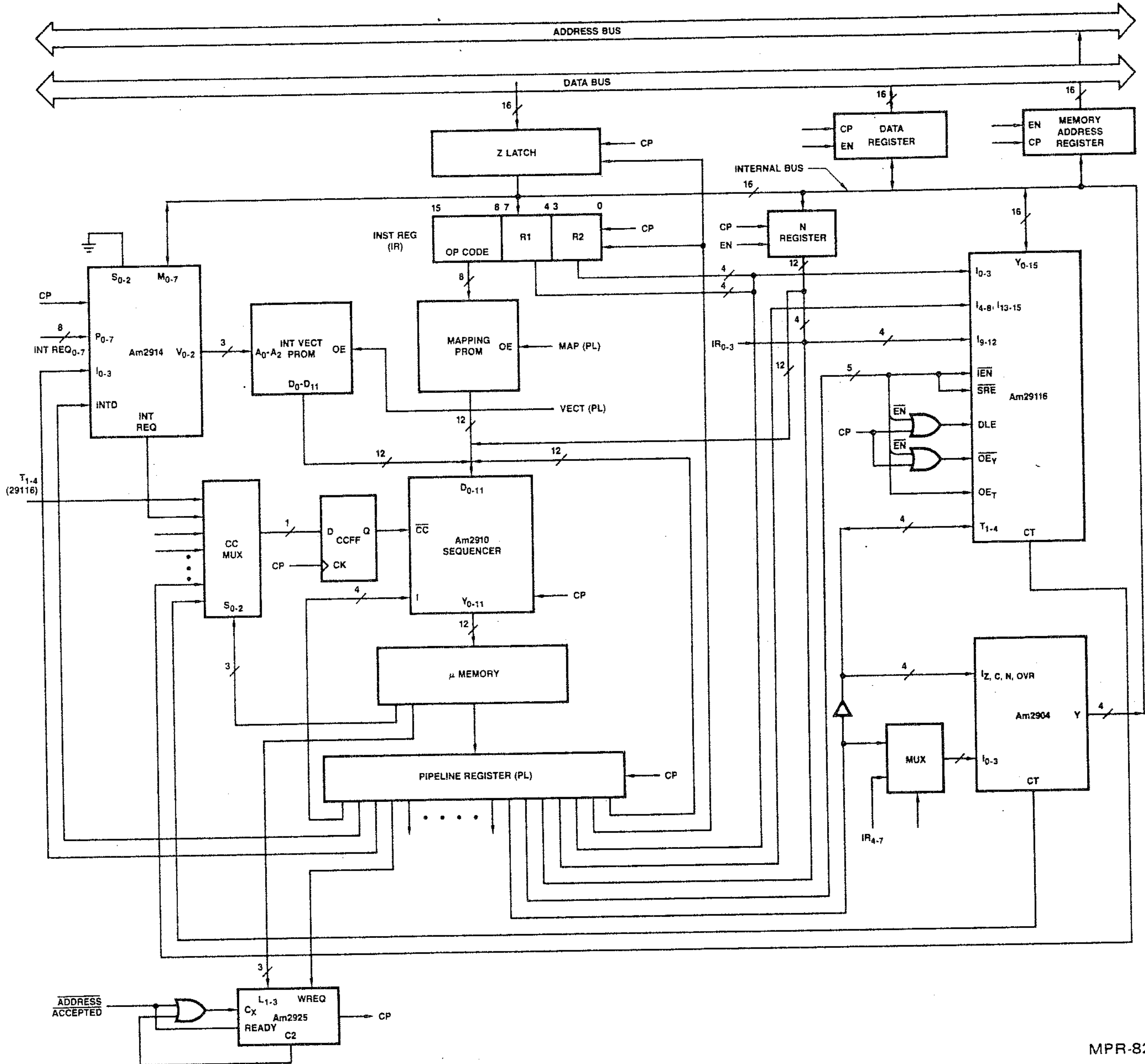
A Microprogrammed CPU Using Am29116 (continued)

Instruction Formats (continued)

- 8-bit opcode specifies 256 instructions
  - includes information about the addressing mode
  - PUSH/POP operations
  - I/O instructions
  - decimal and binary integer arithmetic
- Data types
  - bit
  - nibble
  - byte
  - word

A Microprogrammed CPU Using Am29116 (continued)

## Central Processing Unit




MPR-824

A Microprogrammed CPU Using Am29116 (continued)

## CPU - Architecture

- Internal data transfers use 16-bit wide internal bus

- Data is transferred between the system bus and the internal CPU bus on three paths:

  - data register

  - address register

  - instruction lookahead register (Z-latch)

- Pipelining

  - microlevel: pipeline register at the output of the microprogram memory

    (for overlapped instruction fetch and execution)

  - macrolevel: instruction register (IR) and Z-latch

    . decodes the macroinstruction in the IR

    . next macroinstruction, displacement field or data in the Z-latch

A Microprogrammed CPU Using Am29116 (continued)

## Macroinstruction Data Path

- Macroinstruction can move from main memory to
  - the Z-latch or
  - immediately to the IR (by making the Z-latch transparent)
- Load IR directly:
  - during pipeline-fill operation
  - on instruction after a two-word instruction
- Decoding of (IR) determines the meaning of (Z):
  - next instruction if (IR) is a RR, RS or SS instruction
  - displacement if (IR) is a RX instruction:

    A displacement is moved into the Am29116 via its Y-bus in the cycle in which the operand address is formed.
  - immediate data (used in the execution cycle)
- Am29116 can input and output data in the same microcycle:
  - data passes into the D-latch in $1^{st}$ half of cycle
  - In the second half of the cycle, the D-latch is disabled & the ALU result goes out to internal bus via the Y-bus.
- N-register can be used for:
  - N-way branching or for normalization (use prioritize instr'n)
  - "n" in the rotate-by-n instruction

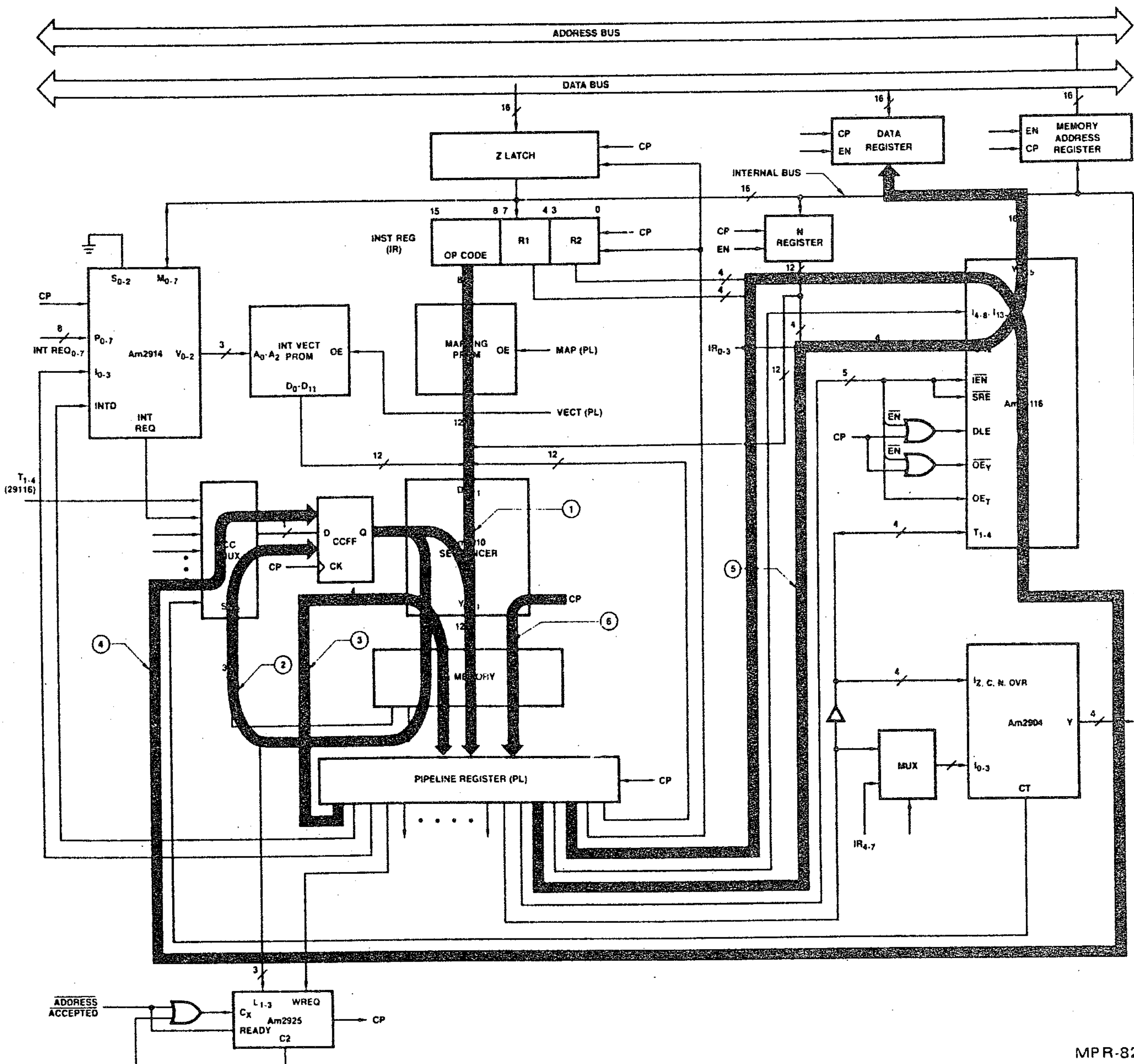
A Microprogrammed CPU Using Am29116 (continued)

Microprogram Control

- Am291Ø sequencer generates address for next microinstruction
  - branch address sources:
    - . pipeline register
    - . interrupt vector decoder
    - . macroinstruction decoder (mapping PROM)
    - . N-register
  - Conditions for branching can come from:
    - . Am29116 condition-test output
    - . Am29Ø4 condition-test output. Provides:
      - for extended set of branching conditions
      - for saving conditions generated previously (a useful result of having 2 status registers)
      - Am29116 T-bus (carry, overflow, negative, zero)
      - Interrupt request accepted by an Am2914 interrupt controller request

A Microprogrammed CPU Using Am29116 (continued)

Timing Analyses



## A Microprogrammed CPU Using Am29116 (continued)

### Timing Analyses (continued)

### Path Computations

Path 1 (IR --> pipeline)

| | From | To | |
|---|---|---|---|
| Instruction register | CP | Q | 13 ns |
| Mapping PROM | ADD | Y | 40 ns |
| Sequencer | D | Y | 20 ns |
| Micromemory | ADD | Y | 40 ns |
| Pipeline register | set up | | 5 ns |
| | | | 118 ns |

Path 2 (CC-flip flop --> CC-MUX control)

| | From | To | |
|---|---|---|---|
| CC flip flop | CP | Q | 13 ns |
| Sequencer | CC | Y | 43 ns |
| Micromemory | ADD | Y | 40 ns |
| CC-MUX | sel | Y | 15 ns |
| CC-flip-flop | set up | | 5 ns |
| | | | 116 ns |

A Microprogrammed CPU Using Am29116 (continued)

Timing Analysis (continued)

Path Computations (continued)

Path 3 (pipeline register --> pipeline register)

| | From | To | |
|---|---|---|---|
| Pipeline register | CP | Y | 13 ns |
| Sequencer | I | Y | 7Ø ns |
| Micromemory | ADD | Y | 4Ø ns |
| Pipeline register | set up | | 5 ns |
| | | | 128 ns |

Path 4 (pipeline register --> CC-flip-flop)

| | from | to | |
|---|---|---|---|
| Pipeline register | CP | Y | 13 ns |
| Three-state gate | enable | Y | 29 ns |
| Am29116 (preliminary) | I | CT | 47 ns |
| CC - MUX | $D_{in}$ | Y | 15 ns |
| CC flip flop | set up | | 5 ns |
| | | | 1Ø9 ns |

A Microprogrammed CPU Using Am29116 (continued)

Timing Analysis (continued)

Path Computations (continued)

Path 5 (pipeline register --> data register)

| | From | To | |
|---|---|---|---|
| Pipeline register | CP | Y | 13 ns |
| Three-state gate | enable | Y | 29 ns |
| Am29116 (preliminary) | I | Y | 88 ns |
| Data register | set up | | 5 ns |
| | | | 135 ns |

Path 6 (PC and STACK --> pipeline register)

| | From | To | |
|---|---|---|---|
| Am291∅ (PC and STACK) | CP | Y | 1∅∅ ns* |
| Micromemory | ADD | Y | 4∅ ns |
| Pipeline register | set up | | 5 ns |
| | | | 145 ns |

* It is assumed that the previous instruction could produce no change in the counter or could only decrement the counter

A Microprogrammed CPU Using Am29116 (continued)

Timing Analysis (continued)

- Most critical path was

| | From | To | |
|---|---|---|---|
| Pipeline register | CP | Y | 13 ns |
| Am29116 | I | CT | 47 ns |
| CC MUX | D | Y | 15 ns |
| Sequencer | $\overline{CC}$ | Y | 43 ns |
| Micromemory | ADD | Y | 4Ø ns |
| Pipeline register | set up | | 5 ns |
| | | | 163 ns |

- Introducing the CC-Flip-Flop separates the cycle time for that path into two non-critical paths (4 and 2).
- Since the CC-FF delays the $\overline{CC}$ signal by one clock, the CC-MUX-select lines are driven directly from the microprogram memory rather than the pipeline register. This tactic re-aligns the selection of the condition code with the execution of the microinstruction.

A Microprogrammed CPU Using Am29116 (continued)

## Macroinstruction Execution

- 4 basic sequences of operations:
  - Form Memory Address of Instruction: (1 microcycle)

    PC, MAR <-- PC + 2
  - Fetch Instruction: (1 microcycle)

    generate Main-Memory-Request and Read-Strobe

    bus <-- ((MAR))

    IR <-- (bus) at the next rising edge
  - Decode Instruction: (1 microcycle)

    IR <-- Z-latch

    PROM generates starting address for the microprogram
  - Execute:

    Am29116 performs the specified operation on the operands.
    The number of microcycles depends upon the operation.

- 2 extra steps are needed for instructions with memory operands
  - Form Operand Address: (1 microcycle)

    MAR <-- (X) + d using the 'd' in the Z-latch
  - Fetch Operand: (1 microcycle)

    Z <-- ((MAR)) or D <-- ((MAR))

A Microprogrammed CPU Using Am29116 (continued)

## Pipelining at the Macrolevel

- There is very little scope in this configuration for additional macrolevel pipelining:
  - You could use a second Am29116 as the basis for a Program Control Unit.
    That is, you would use one Am29116 as an ALU and the second Am29116 as a PCU.
  - You could use some other processor as a basis for this PCU.
- A PCU would increase throughput by introducing parallelism in the advancing of the program counter, and in stack operations.
- The present configuration already incorporates some concurrency, however.

## A Microprogrammed CPU Using Am29116 (continued)

### Pipelining at the Macrolevel (continued)

Overlapping of Register to Register Instructions

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Form Instruction Address | A | B | | | | C | | | D | | | E | | | | | | | | | | PC + 2 → PC<br>PC + 2 → MAR |
| Fetch Instruction | | IR<br>A | Z<br>B | | | | Z<br>C | | | Z<br>D | | | Z<br>E | | | | | | | | | PC + 2 → MAR and Load Z Latch or IR |
| Decode | | | A | | | IR<br>B | | | IR<br>C | | | IR<br>D | | | | | | | | | | Decode Instruction and Load Pipeline Register |
| Form Operand Address | | | | | | | | | | | | | | | | | | | | | | Z + Index Register → MAR |
| Fetch Operand | | | | | | | | | | | | | | | | | | | | | | Load Data Register |
| Execute | | | | A | A | | B | B | | C | C | | D | D | | | | | | | | |
| | Y | Y | | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | | | | | | | | The Am29116 Usage |

A, B, C, D are Register to Register type instructions.
Z = Z Latch
IR = Instruction Register

- Cycle 2: pipefill operation (directly load IR) from memory while Am29116 forms next instruction address
- Cycle 4,5: in single-port Am29116, RR instr. needs 2 cycles
- Cycle 6: IR <-- (Z) and map into microaddress with a PROM while Am29116 forms next instruction address
- Cycle 3 is the only cycle in which the Am29116 is idle.
- After the first 5 cycles, every third cycle produces a result.

## A Microprogrammed CPU Using Am29116 (continued)

### Pipelining at the Macrolevel (continued)

Overlapping of Register to Index Storage Instructions

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Form Instruction Address | A | $A_D$ | | | B | | $B_D$ | | | C | | $C_D$ | | | | | | | | | | PC + 2 → MAR<br>PC + 2 → PC |
| Fetch Instruction | | IR<br>A | Z<br>$A_D$ | | | IR<br>B | | Z<br>$B_D$ | | | IR<br>C | | Z<br>$C_D$ | | | | | | | | | (PC + 2 → MAR and PC)*<br>Load IR or Z Latch |
| Decode | | | A | | | | B | | | | | C | | | | | | | | | | PC + 2 → MAR and PC<br>Decode and Load Pipeline Register |
| Form Operand Address | | | | A | | | | | B | | | | | C | | | | | | | | Z + Index Register → MAR |
| Fetch Operand | | | | | A | | | | | B | | | | | C | | | | | | | PC + 2 → PC and MAR<br>Load Operand in Data Register |
| Execute | | | | | | A | | | | | B | | | | | C | | | | | | |
| | Y | Y | | Y | Y | Y | Y | | Y | Y | Y | Y | | Y | Y | Y | | | | | | The Am29116 Usage |

*For pipefill operation only.

A, B C are Register to Index storage type instructions.
$A_D$, $B_D$, $C_D$ are displacement.
Z = Z Latch
IR = Instruction Register

- Cycle 2: Form $A_D$ creates address to fetch displacement, 'd'.
- Cycle 3: Decoding determines if Z contains a displacement.
- Cycle 6: Two-word instructions use both Z-latch and IR:
  - Instruction is directly loaded into IR.
  - Execution of A prevents formation of address of displacement, 'd'. (a PCU would save a cycle here).
- Every fifth cycle the Am29116 is idle.
- Every fifth cycle produces a result

## A Microprogrammed CPU Using Am29116 (continued)

### Pipelining of the Macrolevel (continued)

Overlapping of Branch-on-Condition RX Type Instructions

M is the condition
(X2)+d is the address

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Form Instruction Address | A | $A_D$ | | | B or K | $B_D$ $K_D$ | | | B+2 K+2 | | $B+2_D$ $K+2_D$ | | | | | | | | | | | PC + 2 → MAR<br>PC + 2 → PC |
| Fetch Instruction | | IR<br>A | Z<br>$A_D$ | | | IR<br>B or K | Z<br>$B_D$ $K_D$ | | | IR<br>B+2 K+2 | | Z<br>$B+2_D$ $K+2_D$ | | | | | | | | | | (PC + 2 → MAR and PC)*<br>Load IR or Z Latch |
| Decode | | | A | | | | B K | | | | B+2 K+2 | | | | | | | | | | | PC + 2 → MAR and PC<br>Decode and Load Pipeline Register |
| Form Operand Address | | | | | | | | B K | | | | | B+2 K+2 | | | | | | | | | Z + Index Register → MAR |
| Fetch Operand | | | | | | | | | B K | | | | | B+2 K+2 | | | | | | | | PC + 2 → PC and MAR<br>Load Operand In Data Register |
| Execute | | | | A* | | | | | | B K | | | | | B+2 or K+2 | | | | | | | |
| | Y | Y | | Y | Y | Y | | Y | Y | Y | Y | | Y | Y | Y | | | | | | | The Am29116 Usage |

*During this cycle decision to branch takes place
If condition is true, Address = K = Index Reg + $A_D$
If condition is false, Address = B = A+1

Z = Z Latch
IR = Instruction Register

- Cycle 5: the result of the execution in cycle 4 held by the Am29Ø4 determines whether the Am29116 will issue the inline or the branch address

A Microprogrammed CPU Using Am29116 (continued)

Microword Format

- 78-bit wide microword
- Control bits for each functional unit are grouped

| Field Width | Mnemonic | Description |
|---|---|---|
| ALU | | |
| 16 | $I_{\emptyset}$-$I_{15}$ | 29116 Instruction |
| 1 | DLE | 29116 Data Latch Enable |
| 1 | IEN | 29116 Instruction Enable |
| 1 | OEY | 29116 Output Enable Y-bus |
| 1 | SRE | 29116 Status Register Enable |
| 1 | OET | 29116 Output Enable T-bus |
| 3 | RAMSRC | 29116 $I_{\emptyset}$-$I_4$ Source Select |
| 2 | NSRC | 29116 $I_9$-$I_{12}$ Source Select |
| Data Path | | |
| 4 | DSEL | Data Register Source/Destination Select |
| 1 | DLD | Data Register Enable |
| 1 | MARLD | Memory Address Enable |
| 1 | IRLD | Instruction Register Enable |
| 1 | ZLD | Z-Latch Enable |
| 1 | NLD | N-Register Enable |
| 1 | MAP | Mapping PROM Output Enable |
| 1 | VECT | Interrupt Vector PROM Output Enable |

A Microprogrammed CPU Using Am29116 (continued)

Microword Format (continued)

| Field Width | Mnemonic | Description |
|---|---|---|
| Memory Control | | |
| 1 | R/W | Memory READ/WRITE Pulse |
| 1 | WREO | Wait Request |
| 1 | DATASTB | Data Strope |
| 1 | MEMREQ | Memory Request |
| Interrupt Control | | |
| 4 | $I_0$-$I_3$ | 2914 Instruction |
| 1 | INTD | 2914 Interrupt Disable |
| Clock Select | | |
| 3 | $L_1$-$L_3$ | 2925 Clock Length Select |
| Status | | |
| 1 | EZ | 2904 Enable Zero |
| 1 | EC | 2904 Enable Carry |
| 1 | ES | 2904 Enable Sign |
| 1 | EOVR | 2904 Enable Overflow |
| 1 | OEM | 2904 Enable Machine Status |
| 1 | OEMICRO | 2904 Enable Micro Status |

A Microprogrammed CPU Using Am29116 (continued)

Microword Format (continued)

| Field Width | Mnemonic | Description |
|---|---|---|
| Test | | |
| 4 | $T_1$-$T_4$ | 29116 or 29Ø4 Test Status Instruction |
| CCSEL | | |
| 3 | CCSEL | Condition Code MUX Select |
| Sequence Control | | |
| 4 | $I_Ø$-$I_3$ | 291Ø Instruction |
| Branch Address | | |
| 12 | BA | Next Micro Address |
| 78 | | |

A Microprogrammed CPU Using Am29116 (continued)

## Conclusion

- Microprogrammability makes for easy and quick design of a customized architecture.
- The powerful instruction set of the Am29116 is very suitable for CPU applications:
  - bit mampulation
  - multiple bit rotate
  - rotate and merge
  - rotate and compare
  - prioritize functions
- We have shown a minimal configuration:
  - a PCU unit based on another Am29116, Am29Ø1's
    or Am293Ø PCU slices could increse the throughput.

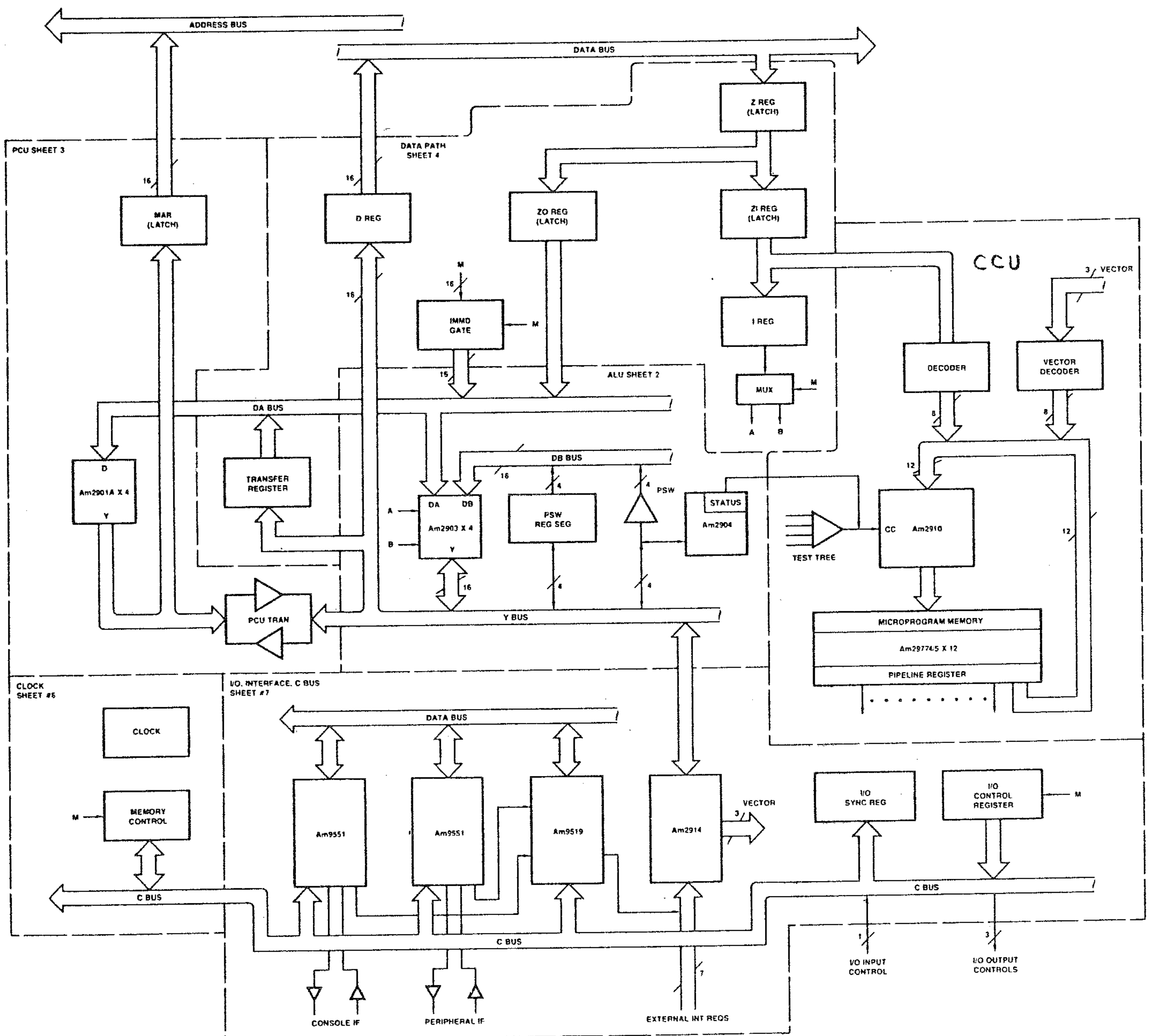
A Microprogrammed CPU Using Am29116 (continued)

Comparison with Super-16

## A Microprogrammed CPU Using Am29116 (continued)

### The Super-16

A 16-bit computer built from Am29ØØ Family parts to illustrate design techniques. Incorporates pipelining at the macro- and microprogram level.



A Microprogrammed CPU Using Am29116 (continued)

Super-16 (continued)

Processor Organization

- Distinct sections
  - program control unit (PCU)
  - arithmetic and logic unit (ALU)
  - computer control unit (CCU)
  - data paths
  - memory control, clock control
  - I/O interface
  - interrupt section

There is one important distinguishing feature as compared with the Am29116 CPU described in the last section:

- The ALU and the PCU are separate.

We will restrict our attention to this point.

A Microprogrammed CPU Using Am29116 (continued)

## Comparison with the Super-Sixteen

The PCU:

- Incorporates:
  - MAR (latch)
  - 4 Am29Ø1 microprocessor slices
  - a 16-bit transfer register
  - a 16-bit bidirectional buffer ... called a transfer driver
- Controls the macroprogram sequencing
- Is controlled by the microprogram
- Register assignments

  RØ- program counter PC

  R1- stack pointer

  R2- stack lower limit

  R3- stack upper limit

  R4- +2 ... a useful constant

  R5- +4 ... a useful constant

  R6,R7- not used but are available

  R8,R15- not used (wired to be disabled)

A Microprogrammed CPU Using Am29116 (continued)

Comparison with the Super-Sixteen (continued)

PCU (continued)

- Function
  - updating of the PCU
  - MAR <-- (PC) for reading instructions or data from main memory
  - updating of the stack pointer
    checking stack limits
- Communication
  - data to PCU from ALU via transfer register
  - data from PCU to Y-bus of the ALU via PCU
    transfer drivers

A Microprogrammed CPU Using Am29116 (continued)

Comparison with the Super-Sixteen (continued)

ALU

- Designed from
  - 4 Am29Ø3 superslices
  - Am29Ø4 status and shift control logic
  - PSW register
  - 3 buses

    DA: ALU input from $Z_Ø$ register (memory data)

    IMMD gate (immediate field of microcode)

    DB: ALU input from PSW

    Y: ALU output, input to RAM registers

- RAM register selection from instruction register (I)
  - $I_{Ø-3}$: A address on Am29Ø3
  - $I_{4-7}$: B address on Am29Ø3 or

    control for CCMUX (in Am29Ø4) in conditional

    branch instructions (macrolevel!)

A Microprogrammed CPU Using Am29116 (continued)

Comparison with the Super-Sixteen (continued)

ALU (continued)

- Byte-wide operations are possible as well as word-wide:
  - only the lower 8 bits are affected
  - by disabling write-enable & output-enable for slices 3,4
  - word/byte MUX selects C,N,OVR from slice 2 (MSS)
  - force $Y_{8-15}$ to zero for Z-signal
- Control for the ALU
  - A & B addresses, Am29Ø4 CCMUX from macrolevel $I_{7-\emptyset}$
  - ALU operations from microlevel $M_{78-86}$
  - WORD signal $M_{9\emptyset}$ (byte/word)

A Microprogrammed CPU Using Am29116 (continued)

Comparison with the Super-Sixteen (continued)

(Macro) Instruction Execution

- Form instruction address (PCU)
  - publish bus request at the beginning of the cycle
  - PC, MAR <-- (PC) + 2
  - address bus <-- (MAR) 5Øns prior to beginning of next cycle
- Instruction fetch (main memory)
  - the main memory is fast enough to all reading in one cycle
- Decode (mapping PROM)
  - fetched instruction is routed through Z and Z1 registers to the instruction decoder (mapping PROM)
  - 8-bit opcode is address for the PROM giving the starting address for the microprogram to execute this instruction
- Displacement fetch (main memory)
  - every instruction fetch is followed by another read cycle:
    . next instruction fetch for one word instructions
    . displacement fetch for two word instructions
  - decoding of the previous instruction determines how this data should be interpreted.

A Microprogrammed CPU Using Am29116 (continued)

Comparison with the Super-Sixteen (continued)

(Macro) Instruction Execution (continued)

- Form operand address (ALU)
  - fetched displacement is sent through the Z and ZØ register to the ALU (29Ø3)
  - MAR <-- ($X_2$) + d (5Ø ns before the end of the cycle)

    (See note 1)

- Operand fetch (memory)

- Execute (ALU) (See note 2)
  - 29Ø3's perform specified operation on the operand's
  - simultaneous with the last execution cycle the instruction decoder is enabled

Notes: 1) Remember the instruction format for the CPU in the last section. This application is an emulation of the Super-16, so that the instruction formats match.

2) Execution may take more than one microcycle.

## Pipelining at the Macrolevel (continued)

### Register-to-Register Pipeline Operation

A, B, C, D are RR instructions

| Action | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Form Instruction Address | A | B | C | D | | | | | | | | | |
| Fetch Instruction | | A | B | C | D | | | | | | | | |
| Decode | | | A | B | C | D | | | | | | | |
| Fetch Displacement | | | | | | | | | | | | | |
| Form Operand Address | | | | | | | | | | | | | |
| Fetch Operand | | | | | | | | | | | | | |
| Execute | | | | A | B | C | D | | | | | | |
| | | | | | | | | | | | | | |
| Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | | | | | |

- Cycle 1,2: Compare with pipelining of the CPU in the last section
- Cycle 3: Address of a third instruction is formed (Super-16: $Z, Z_{\emptyset}, Z_1$-registers; Other: Z-register)
- Cycle 4: Simultaneous instruction execution and formation of instruction address

  Execution of A needs:

  - one microcycle for most instructions because of simultaneous decoding of B
  - more than one microcycle for a few instructions (e.g. I/O instructions). The pipeline stops.

- After the first three cycles, every cycle produces a result.

## Pipelining at the Macrolevel (continued)

### Register-to-Indexed-Storage Pipeline Operation

A, B, C, D are RX instructions

| Action | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Form Instruction Address | A | | B | | | C | | | | | | | |
| Fetch Instruction | | A | | B | | | C | | | | | | |
| Decode | | | A | | | B | | | C | | | | |
| Fetch Displacement | | | A | | | B | | | C | | | | |
| Form Operand Address | | | | A | | | B | | | C | | | |
| Fetch Operand | | | | | A | | | B | | | C | | |
| Execute | | | | | | A | | | B | | | C | |
| | | | | | | | | | | | | | |
| Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |

- Cycle 2: The same actions as in the RR pipeline operation occurs (fetch instruction A, MAR <-- PC + 2)

- Cycle 3: Form instruction address, memory read and decode are performed simultaneously

  The system doesn't "know" whether a displacement or the next instruction is fetched from memory.

  After the decode of instruction A it is determined to be a displacement

- Cycle 4: Simultaneous form instruction address (PCU) and form operand address (ALU). This was impossible in the CPU of the last section.

- Cycle 5: Only one pipeline stage is active: memory read

  The operand is fetched

  Decoder is not enabled yet and therefore the upper part of the pipeline is stopped

Pipelining at the Macrolevel (continued)

Register-to-Indexed-Storage Pipeline Operation (continued)

- Cycle 6: Compare with Cycle 3

  Additional execute A cycle (ALU)

  In this cycle you see again the advantage of two separate devices for the PCU and the ALU.
  You can form instruction address and execute in the same microcycle.

- The pipeline needs 6 cycles from the beginning to produce the first result.

- Every third cycle a result is produced.

## Pipelining at the Macrolevel (continued)

### Branch on Condition RX Pipeline Operation

A = RX Branch Instruction
B = Next RX Instruction if branch is not taken
K = next RX Instruction if branch is taken

| Action | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Form Instruction Address | A | | B | K | B+2<br>K+2 | | | | | | | | |
| Fetch Instruction | | A | | B | K | B+2<br>K+2 | | | | | | | |
| Decode | | | A | | | B<br>K | etc. | | | | | | |
| Fetch Displacement | | | A | | | | B<br>K | | | | | | |
| Form Operand Address | | | | | | | | B<br>K | | | | | |
| Fetch Operand | | | | | | | | | B<br>K | | | | |
| Execute | | | | $A_1$ | $A_2$ | $A_3$ | | | | B<br>K | | | |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | | |
| | | | | | | | | | | | | | |

Cycle 3: Form instruction-address (next instruction if branch is not taken)

Fetch displacement of next instruction

Decoding A detects a branch instruction

Pipelining at the Macrolevel (continued)

Branch on Condition RX Pipeline Operation (continued)

- Cycle 4: Execution of the branch instruction starts

  - Determination whether the condition is true or false is not yet possible.

  - Sequencing of the program is temporarily unknown

  To place the correct instruction in the Z-register when decoding of the next instruction (B or K) is enabled, form the instruction addresses of the two alternative instructions:

  - Form instruction address K (where $K = (X_2) + d$)

  - Instruction address B was formed in the last cycle and B can be fetched in this cycle.

- Cycle 5: Fetch instruction K

  Executing $A_2$ determines whether the condition is true or false

  The cycle is long enough to form the correct instruction address (K+2 or B+2)

- Cycle 6: The last execution cycle $A_3$ enables the decoder.

  Decode either K or B as determined in cycle 5.

- You see: The generation of the two alternative instruction addresses prevents the pipeline from flushing out totally.

# Super-16 Microword Format

16-BIT COMPUTER — MICRO CONTROL WORD BIT DEFINITIONS

| Description | Mnemonic | Bit | Field |
|---|---|---|---|
| ROUTE TO B | $\overline{RTB}$ | 95 | MISC |
| TRANSFER Z TO ZI | (BP) Z → ZI | 92 | MISC |
| Am2910 | $\overline{CCEN}$ | 91 | MISC |
| Am2903 IEU WORD/BYTE | $\overline{WORD}$ | 90 | ALU (13) |
| Am2903 | $\overline{EA}$ | 89 | ALU (13) |
| Am2903 | $\overline{OEY}$ | 88 | ALU (13) |
| Am2903 | $\overline{OEB}$ | 87 | ALU (13) |
| Am2903 | $I_8$ | 86 | ALU (13) |
| Am2903 | $I_7$ | 85 | ALU (13) |
| Am2903 | $I_6$ | 84 | ALU (13) |
| Am2903 | $I_5$ | 83 | ALU (13) |
| Am2903 | $I_4$ | 82 | ALU (13) |
| Am2903 | $I_3$ | 81 | ALU (13) |
| Am2903 | $I_2$ | 80 | ALU (13) |
| Am2903 | $I_1$ | 79 | ALU (13) |
| Am2903 | $I_0$ | 78 | ALU (13) |
| ENABLE TRANSFER REG. | $\overline{ENTREG}$ | 77 | Data Path (13) |
| LOAD TRANSFER REG. | $\overline{LDTREG}$ | 76 | Data Path (13) |
| I-REG EN $\overline{CTR}$ | $\overline{ENCTR}$ | 75 | Data Path (13) |
| I-REG INC/$\overline{DEC}$ | INC | 74 | Data Path (13) |
| PCU TRANS CHIP DISABLE | PCUCD | 73 | Data Path (13) |
| PCU TRANSFER REG. | PCU → Y | 72 | Data Path (13) |
| LOAD MEMORY ADDR. REG. | LDMAR | 71 | Data Path (13) |
| LOAD D-REG. | LDD | 70 | Data Path (13) |
| LOAD ZI INTO I REG. | ZI → I | 69 | Data Path (13) |
| ENABLE Z0 → DA | $\overline{ENZ0}$ | 68 | Data Path (13) |
| ENABLE PSW | $\overline{PSW}$ | 67 | Data Path (13) |
| SHIFT CNT Am2910 ADDR. | $\overline{SHTCNTEN}$ | 66 | Data Path (13) |
| BRANCH INSTR. EN | $\overline{BRIEN}$ | 65 | Data Path (13) |
| Am2901 F → B/$\overline{Q}$ | $PCUI_7$ | 64 | Program Control (11) |
| Am2901 | $PCUI_3$ | 63 | Program Control (11) |
| Am2901 | $PCUI_2$ | 62 | Program Control (11) |
| Am2901 | $PCUI_1$ | 61 | Program Control (11) |
| Am2901 | $PCUI_0$ | 59 | Program Control (11) |
| Am2901 | $PCUA_2$ | 59 | Program Control (11) |
| Am2901 | $PCUA_1$ | 58 | Program Control (11) |
| Am2901 | $PCUA_0$ | 57 | Program Control (11) |
| Am2901 | $PCUB_2$ | 56 | Program Control (11) |
| Am2901 | $PCUB_1$ | 55 | Program Control (11) |
| Am2901 | $PCUB_0$ | 54 | Program Control (11) |
| BUS REQUEST | REQB | 53 | Memory Control (5) |
| MEMORY REQUEST | $\overline{MREQ}$ | 52 | Memory Control (5) |
| HOLD REQUEST | $\overline{HREQ}$ | 51 | Memory Control (5) |
| MEMORY WRITE/$\overline{READ}$ | WRITE | 50 | Memory Control (5) |
| MEMORY WORD/$\overline{BYTE}$ | MWORD | 49 | Memory Control (5) |

## Super-16 Microword Format (continued)

16-BIT COMPUTER — MICRO CONTROL WORD BIT DEFINITIONS

| Field | Bits | Description | Mnemonic |
|---|---|---|---|
| Control Strobes (6) | X X 48 47 46 45 44 43 | EN IMMEDIATE → DA BUS | $\overline{IMMD}$ |
| | | ROM/IREGEN | ROM/$\overline{I}$ |
| | | I/O CONTROL REG. EN | $\overline{IOEN}$ |
| | | Am2914 INTERRUPTS DISABLE | $\overline{INTDIS}$ |
| | | Am2914 $ENI_0$-$ENI_3$ | $\overline{INTRIEN}$ |
| | | Am2904 SHIFT EN | $\overline{SHFTEN}$ |
| Control Bits (8) | 42 41 40 39 38 37 36 35 | GENERAL USE CONTROL BITS | $CNTLB_7$ |
| | | | $CNTLB_6$ |
| | | | $CNTLB_5$ |
| | | | $CNTLB_4$ |
| | | | $CNTLB_3$ |
| | | | $CNTLB_2$ |
| | | | $CNTLB_1$ |
| | | | $CNTLB_0$ |
| Status (9) | X X 34 33 32 31 30 29 28 27 26 | Am2904 OUT EN CONDITIONAL TEST | $\overline{OECT}$ |
| | | Am2904 EN ZERO | $\overline{EZ}$ |
| | | Am2904 EN CARRY | $\overline{EC}$ |
| | | Am2904 EN SIGN | $\overline{ES}$ |
| | | Am2904 EN OVERFLOW | $\overline{EOVR}$ |
| | | Am2904 EN MACHINE STATUS | $\overline{CEM}$ |
| | | Am2904 EN MICRO STATUS | $\overline{CE\mu}$ |
| | | Am2904 $I_{12}$ CARRY OUT CNTL | $I_{12}$ |
| | | Am2904 $I_{11}$ CARRY OUT CNTL | $I_{11}$ |
| Test (6) | 25 24 23 22 21 20 | Am2904 | $TEST_5$ |
| | | Am2904 | $TEST_4$ |
| | | Am2904 | $TEST_3$ |
| | | Am2904 & Am25LS251 | $TEST_2$ |
| | | Am2904 & Am25LS251 | $TEST_1$ |
| | | Am2904 & Am25LS251 | $TEST_0$ |
| Sequences CNTL (4) | 19 18 17 16 | Am2910 $I_3$ | $NAC_3$ |
| | | Am2910 $I_2$ | $NAC_2$ |
| | | Am2910 $I_1$ | $NAC_1$ |
| | | Am2910 $I_0$ | $NAC_0$ |
| Next Micro Addr & Immed (16) | 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | | $M_{15}$ |
| | | | $M_{14}$ |
| | | | $M_{13}$ |
| | | | $M_{12}$ |
| | | | $M_{11}$ |
| | | | $M_{10}$ |
| | | | $M_9$ |
| | | | $M_8$ |
| | | | $M_7$ |
| | | | $M_6$ |
| | | | $M_5$ |
| | | | $M_4$ |
| | | | $M_3$ |
| | | | $M_2$ |
| | | | $M_1$ |
| | | | $M_0$ |

Comparing Am29Ø1, Am292Ø3, Am29116
for General Purpose CPU's

Comparison of Features as General Purpose CPU's

| Operation | Am29Ø1C | Am292Ø3 | Am29116 |
|---|---|---|---|
| 16-bit ADD | 83 ns (max) | ~ 12Ø ns (max) | 1ØØ-12Ø ns (max) |
| 16-bit MULT | ~ 16ØØ ns (max) | ~ 18ØØ ns (max) | ~ 9ØØØ ns (max)<br>~ 3ØØ ns (max with Am29516/7) |
| BCD ADD | ~ 1ØØØ ns (max) | ~ 12Ø ns (max) | ~ 1ØØØ ns (max) |
| Data Word Width | 4-256 Bits | 4-256 Bits | 8 or 16 Bits |
| # Registers | 16 only | 16,32,48,64 | 32 only |
| 8-Bit Rotate | ~ 8ØØ ns (max) | ~ 1ØØØ ns (max) | ~ 1ØØ-12Ø ns (max) |
| 8-Bit Rotate & Merge | ~ 1ØØØ ns (max) | ~ 13ØØ ns (max) | ~ 1ØØ-12Ø ns (max) |
| Devices for 16-bit ALU | 4 x Am29Ø1C<br>1 x Am29Ø2<br>1 x Am29Ø4<br>~ 3 x SSI | 4 x Am292Ø3<br>1 x Am29Ø2<br>1 x Am29Ø4 | 1 x Am29116 |
| Typical power for 16-bit ALU | 5.2 W | 5.6 W | 2.2 W |

## Performance Analysis

| Processor Type | Bit Test | Bit Set |
|---|---|---|
| 8048 | 2 / 5,000ns | 1 / 2,500ns |
| Am9080 | 5 / 4,250ns | 2 / 1,750ns |
| AmZ8000 | 1 / 1,000ns | 1 / 1,000ns |
| Am2901B (4)<br>Am2904<br>Am2902A<br>25LS2538 (2) | 1 / 100ns | 1 / 100ns |
| Am2901B (4)<br>Am2904<br>Am2902A<br>25LS2538 (2)<br>25S10 (8) | 1 / 100ns | 1 / 100ns |
| Am29116 | 1 / 100ns | 1 / 100ns |

Instruction Type: Bit Test, Bit Set

Key: Number of Instructions / Execution Time

Performance Analysis (continued)

Instruction Type

| Processor Type | Rotate by N | Rotate and Merge |
| --- | --- | --- |
| 8Ø48 | 28 / 212,5ØØns | 44 / 252,5ØØns |
| Am9Ø8Ø | 32 / 113,ØØØns | 42 / 138,ØØØns |
| AmZ8ØØØ | 8 / 11,25Øns | 12 / 15,25Øns |
| Am29Ø1B (4)<br>Am29Ø4<br>Am29Ø2A<br>25LS2538 (2) | 9 / 1,Ø25ns | 12 / 1,325ns |
| Am29Ø1B (4)<br>Am29Ø4<br>Am29Ø2A<br>25LS2538 (2)<br>25S1Ø (8) | 1 / 16Øns | 4 / 46Øns |
| Am29116 | 1 / 1ØØns | 1 / 1ØØns |

Key: Number of Instructions / Execution Time

<u>Performance Analysis</u> (continued)

Instruction Type

| Processor Type | 16-bit ADD | 16-bit Multiply |
|---|---|---|
| 8∅48 | 6 / 15,∅∅∅ns | 54 / 81∅,∅∅∅ns |
| Am9∅8∅ | 1 / 2,5∅∅ns | 4∅ / 5∅9,∅∅∅ns |
| AmZ8∅∅∅ | 1 / 1,∅∅∅ns | 1 / 17,5∅∅ns |
| Am29∅1B (4)<br>Am29∅4<br>Am29∅2A<br>25LS2538 (2) | 1 / 115ns | 2 / 2,∅∅∅ns |
| Am29∅1B (4)<br>Am29∅4<br>Am29∅2A<br>25LS2538 (2)<br>25S1∅ (8) | 2 / 115ns | 2 / 2,∅∅∅ns |
| Am29116 | 1 / 1∅∅ns | 12 / 9,6∅∅ns |

Key: Number of Instructions / Execution Time

Performance Analysis (continued)

Are you familiar with the Am25S1Ø?

How does it help speed up shifts and rotates on Am29Ø1 systems?

It is a four-bit high-speed shifter:

- It shifts four bits of data Ø,1,2 or 3 places.

- Several devices can be connected to:

  . perform shifts of Ø,1,2 or 3 places
    on words of any length. (# of 25S1Ø's equals # of bits/4)

  . perform a complete end-around barrel shift.
    (# of 25S1Ø's needed equals # of bits/2)

# CHAPTER 7

Exercises - Part 2

# Exercises - Part 2

1. You have 8 ASCII characters that normally occupy 64 bits of memory. Since each byte has in fact only 7 active bits, it may be useful to pack these bytes into only 56 bits for storage on a disk. In writing these bytes to a disk you wish to pack them into a 56-bit contiguous frame by discarding the parity bit from each byte (i.e. the bit in the most signifigant position). Write the microinstructions for the Am29116 for packing these 64 bits into 56 bits. Assume that the characters are initially in RØ thru R3 and are to be packed into R4 thru R7.

2. Write the code to perform a 16 x 16-bit unsigned integer multiplication. Write your code on the assumption that the operands are already in the RAM or on the assumption that the operands are to be supplied by your code as immediate values.

3. If you have time, try the following exercise:

   The Am29116 can be very effective in arbitrating requests for service from several different sources. Suppose there are eight sources of service requests and the Am29116 has already serviced all requests from sources S7,S6 and S5. Write the code to cause the Am29116 to branch to the service routine associated with the highest priority source of the group S4,S3,S2,S1,SØ while ignoring S7,S6 and S5.

Solutions for Exercises - Part 2

Solutions for Exercises - Part 2

Problem #1: Packing ASCII characters

Example

```
                                                          Even parity
                                                          ↓
 ...      ... 0100 0100   1100 0011   0100 0010   0100 0001
              \___D___/   \___C___/   \___B___/   \___A___/

                      R1                      R0
```

```
SOR, B, MOVE, SORA, R0       ;0000 0000 0100 0001 --> ACC
ROTM, W, 15, MRAI, R0
IMME  H#3F80

    U: 0100 0010 0100 0001
rot U: 1010 0001 0010 0000
    R: 0000 0000 0100 0001

 mask: 0011 1111 1000 0000
-------------------------
  ACC: 0010 0001 0100 0001
          \______/\______/
       B without A without
       parity    parity

ROTM, W, 14, MRAI, R1
IMME H#C000

    U: 0100 0100 1100 0011
rot U: 1101 0001 0011 0000
    R: 0010 0001 0100 0001

 mask: 1100 0000 0000 0000
-------------------------
  ACC: 1110 0001 0100 0001
       \/\______/\______/
        ↑    B        A
        └─ two last bits of C

      etc.
```

Solutions for Exercises - Part 2 (continued)

Microinstructions for Packing ASCII Characters

```
SOR     B, MOVE, SORA, RØ
ROTM    W, 15, MRAI, RØ          IMME   H # 3F8Ø
ROTM    W, 14, MRAI, R1          IMME   H # CØØØ
SOR     W, MOVE, SOAR, R4
ROTR1   W, 14, RTRA, R1
ROTM    W, 13, MRAI, R1          IMME   H # ØFEØ
ROTM    W, 12, MRAI, R2          IMME   H # FØØØ
SOR     W, MOVE, SOAR, R5
ROTR1   W, 12, RTRA, R2
ROTM    W, 11, MRAI, R2          IMME   H # Ø3F8
ROTM    W, 1Ø, MRAI, R3          IMME   H # FCØØ
SOR     W, MOVE, SOAR, R6
ROTR1   W, 1Ø, RTRA, R3
ROTM    W, 9, RAI, R3            IMME   H # ØØFE
SOR     B, MOVE, SOAR, R7
```

## Solutions for Exercises - Part 2 (continued)

Problem #2: Unsigned Integer Multiplication

Some observations...

o Result is a 32-bit value.
  The Am29116 does not provide a double length shift directly.

o Remember that the Am29116 has a single port RAM architecture.

  - try to minimize operations which need two register operands

    i.e. - shift the result, not the multiplicand
           (only the result needs two registers)

         - use the ACC or D-latch for the multiplicand

         - in the double-precision addition use zero as
           as an immediate value

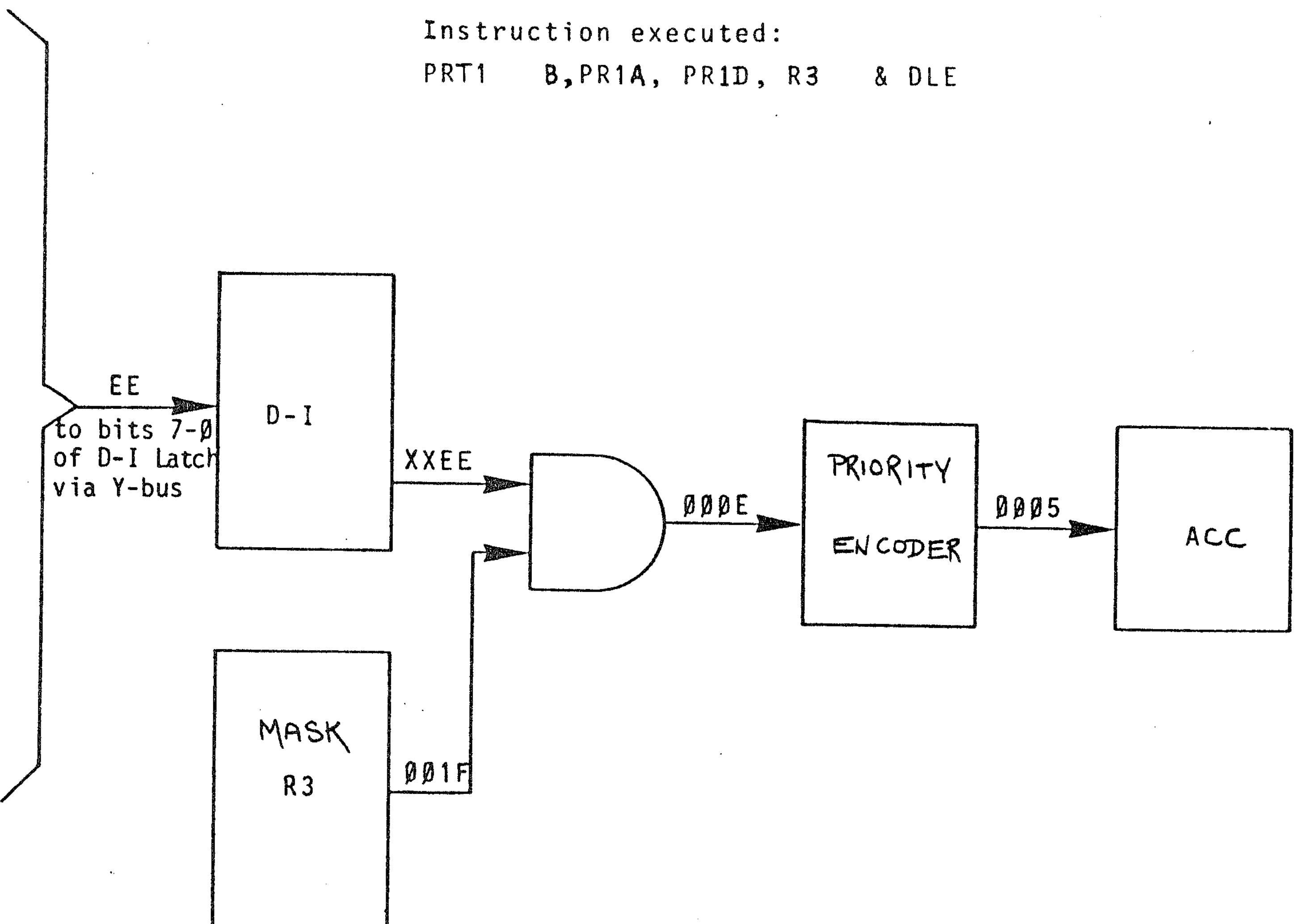
Exercise Solutions (continued)

Unsigned Multiplication (16 x 16 bit)

| Symbolic Address | Branch Address | Sequencer | CC | Am29116 Instruction | $\overline{SRE}$ | Comment |
|---|---|---|---|---|---|---|
| | | cont | | SOR,MOVE,SOIR,RØØ | x | Multiplier --> RØØ |
| | | cont | | Multiplier | x | Immediate value |
| | | cont | | SONR,MOVE,SOI,NRA | x | Multiplicand --> ACC |
| | | cont | | Multiplicand | x | Immediate value |
| | | cont | | SOR,MOVE,SOZR,RØ2 | x | Initialize MSH of product to Ø |
| | 16 | PUSH | PASS | SOR,MOVE,SOZR,RØ3 | x | Initialize LSH of product to Ø<br>Load counter & push loop address |
| LOOP: | | cont | | SHFTR,SHRR,SHUPZ,RØ3 | Ø | Shift LSH of product by one.<br>Load LSB with zero. |
| | | cont | | SHFTR,SHRR,SHUPL,RØ2 | x | Shift MSH of product by one.<br>Load LSB with QLINK. |
| | | cont | | SHFTR,SHRR,SHUPZ,RØØ | Ø | Shift multiplier.<br>Load LSB with zero. |
| | SKIP | CJP | $\overline{CT}$(29116) | TEST,TL | x | Branch to SKIP if MSB was zero. |
| | | cont | | TOR1,TORAR,ADD,RØ3 | Ø | Double precision ADD<br>to accumulate partial product ...<br>LSH: RØ3+ACC --> RØ3 |
| | | cont | | TOR1,TORIR,ADDC,RØ2 | x | MSH: RØ2+Ø+C --> RØ2 |
| | | cont | | Ø | x | Immediate zero |
| SKIP: | | RFCT | | | x | Loop back to "LOOP" |

## #3 Exercise Solution

DRIVE 1 REQUEST H
DRIVE 2 REQUEST H
DRIVE 3 REQUEST H
DRIVE 4 — L
DRIVE 5 REQUEST H
DRIVE 6 REQUEST H
DRIVE 7 REQUEST H
DRIVE 8 — L



# APPENDIX A

## A BRIEF REVIEW OF NUMBER THEORY

A Brief Review of Number Theory

Number theory has application in relation to several products of Advanced Micro Devices, including the Am29500 Family of signal and array-processing devices and the Am9520/Am8065 Burst Error Processor. The theory also is important in cryptography and in random-number generation. What follows is a brief summary, without proofs, of selected number theory results and related notation.

For proofs and much more material see:

J.V. Uspensky and M.A. Heaslet, Elementary Number Theory (McGraw-Hill, New York 1939)

H.J. Nussbaumer, Fast Fourier Transform and Convolution Algorithms (Springer-Verlag, New York 1982)

If 'a' and 'b' are integers, with 'b' positive, the division of 'a' by 'b' is defined by:

$$a=bq+r \quad , \qquad 0 \leq r < b \quad ,$$

where 'q' is called the quotient and 'r' is called the remainder. When 'r' is zero, 'b' and 'q' are factors or divisors of 'a'. To indicate that 'b' is a factor of 'a' we write "b|a", which we can read as "'b' divides 'a'". When 'a' has no divisors other than 1 and 'a', 'a' is a prime. In all other cases, 'a' is composite.

All integers which yield the same remainder when divided by 'b' are said to be congruent modulo 'b'. To say that 'c' is congruent to 'd' modulo 'b' we write

$c \equiv d$ modulo b. (but we will write = for ≡ from here on)

This condition will be true if $b|(c-d)$.

Modulus arithmetic is similar to ordinary arithmetic:

$(a_1 \pm a_2)$ modulo b = (($a_1$ modulo b) $\pm$ ($a_2$ modulo b)) modulo b

$(a_1 \cdot a_2)$ modulo b = ($a_1$ modulo b)$\cdot$($a_2$ modulo b) modulo b

However, division is a little more complicated. If

$na_1 = na_2$ modulo b

we <u>cannot</u> always cancel the n's and conclude

$a_1 = a_2$ modulo b.

What is true is that

$a_1 = a_2$ modulo (b/d),

where d is the greatest common divisor of 'n' and 'b' which we write as

d=(n,b).

Only if d=1 can we cancel the n's. That is, we can only divide by numbers that are "relatively prime" to the modulus.

It proves useful to define a function which designates the number of integers that are smaller than a given integer, 'm', and are relatively prime to 'm'. We call this <u>Euler's totient function</u> and write it as "$\varphi(m)$".

If 'm' is a prime then $\varphi(m) = m-1$.

If 'm' is composite such that

$m = p_1^{a1} \cdot p_2^{a2} \cdot p_3^{a3} \cdot \ldots \cdot p_s^{as}$ for $p_1, p_2, p_3, \ldots, p_s$ all prime,

then

$$\varphi(m) = m \cdot \frac{p_1-1}{p_1} \cdot \frac{p_2-1}{p_2} \cdot \frac{p_3-1}{p_3} \cdot \ldots \cdot \frac{p_s-1}{p_s}$$

We now have enough notation to state Euler's Theorem:

If $(a,m) = 1$ then

$$a^{\varphi(m)} = 1 \text{ modulo } m.$$

Euler's Theorem allows us to perform division with respect to a modulus in a more general way than we discussed above. Suppose we want to solve for 'x' given this linear congruence with one unknown:

$$ax=c \text{ modulo } m$$

Using Euler's Theorem

$$ax=c \cdot a^{\varphi(m)} \text{ modulo } m$$

and provided $(a,m)=1$ we may divide both sides by 'a' so that

$$x=c \cdot a^{\varphi(m)-1} \text{ modulo } m$$

and we can see that $a^{\varphi(m)-1}$ is the reciprocal of 'a' with respect to modulus 'm'.

One of the uses of Euler's Theorem relates to its application to the Chinese Remainder Theorem.

The Chinese Remainder Theorem states that simultaneous linear congruences in one unknown can be solved:

> Let $m_i$ be k positive integers greater than 1 and relatively prime in pairs. The set of linear congruences $x=r_i$ modulo $m_i$ has a unique solution modulo M, where $M=m_1 \cdot m_2 \cdot m_3 \cdot \ldots \cdot m_k$.

The calculation of x from the $r_i$'s and $m_i$'s is called the Chinese remainder reconstruction. It can be shown that

$$x = \left(\frac{M}{M_1}\right)^{\varphi(m_1)} r_1 + \left(\frac{M}{M_2}\right)^{\varphi(m_2)} r_2 + \ldots + \left(\frac{M}{M_k}\right)^{\varphi(m_k)} r_k \quad \text{modulo M.}$$

That is, 'x' is obtained from a linear combination of the remainders:

$$x = A_1 r_1 + A_2 r_2 + \ldots + A_k r_k \quad \text{modulo M}$$

We can note that for x=1, all $r_i$'s also equal 1. Hence we see that the coefficients, $A_i$, also sum to 1:

$$A_1 + A_2 + \ldots + A_k = 1 \quad \text{modulo M}$$

which is a useful check on the correctness of a set of such Chinese remainder reconstruction coefficients which we may have calculated.

It will often be easier to calculate the coefficients of the $r_i$'s by knowing that

$$\left(\frac{M}{m_i}\right)^{\varphi(m_i)} \text{modulo M} = \frac{M}{m_i} \cdot \left[\left(\frac{M}{m_i}\right)^{\varphi(m_i)-1} \text{modulo } m_i\right]$$

where the equivalent expression on the RHS above can be calculated on a machine with a shorter capacity for integers than is required for the LHS.

An Example of Chinese-Remainder Reconstruction

Let us apply the above material to the problem of calculating the location of an error burst detected by an Am952Ø/Am8Ø65 Burst Error Processor using its 32-bit polynomial. In this case, the location in bits will be given modulo 21 and 2Ø47. These moduli have no common factors and hence are relatively prime. We may expect from the Chinese Remainder Theorem that we can calculate the location modulo M=21·2Ø47=42,987.

We will need the totients for the two factors:

21 is composite, thus $\varphi(21) = \varphi(3 \cdot 7) = 21 \cdot \frac{3-1}{3} \cdot \frac{7-1}{7} = 12$

2Ø47 is also composite, thus $\varphi(2047) = \varphi(23 \cdot 89) = 2047 \cdot \frac{23-1}{23} \cdot \frac{89-1}{89} = 1936$

To understand how to proceed to calculate the Chinese remainder reconstruction coefficients, let us begin with the second coefficient:

$$A_2 = \left(\frac{M}{m_2}\right)^{\varphi(m_2)} = \left(\frac{42,987}{2047}\right)^{1936} = 21^{1936} \bmod 42,987.$$

But $21^{1936}$ =

65547269477777817757200969720480807996746850662145197381218624584202721941899507
21423897202387805933094706460527736304494713254745339081373600441413238843120193
32585827514379333859761233117712533509551459443911208580632578963327992769384375
42603036057970228687681011232749791880072864832561980276282272591617574260963299
10793538286544760653100399786955431363161625429428147166799811460139502247280241
48421990224184032422659001952950125880854620435176178676536304017189886727728577
66478762808632533720316541630429665323923628823770688465926358082614916364897819
95299608995721341133829535195688330579297981894772229974859050987993820371499794
81395954754540598720417165654121078033944840123118486698356560103988760885806716
49401708128661849049145563730469483530514180432470512369426690573967032574682856
19505758551358401503702558533907289929937210320299658444316439393747438560146636
68039649829256742652205700088710439478948136143462730131378420168689745030305135
30381506509830279486809578859037463252657776488671886012947491670732595286632623
51238881264892457976606743296108336267285279283503060928469489247666663467542882
42872774392908778211927604044280555600130652888690727539976402131080690579575162
33026218677122044778041620387985106584998777559109963128759282081825133237066491
14360200478860036199638263326158639613593793313283634473525862578760726807117983
83504786413185684889586777243776418266864603084497027156863545962997632912488928
24382666238233440270657440539167330004997768395266387139798524809164723581467953
70122867501116649732338446947707912921938208517168298067784043392255318744545510
48987827341190093854105444095401537664898756640898301015058812731231352290137586
72561948104908089710618342205542107240251659744640805787879248125748563346533914
49216499047922658518221629883204672382218062598860097198121309519710514803739640
47732062609858999542190657569218137188675174633600219492324116932670268519947893
95154782422791252339103131937353661659594851341730694387344197857885617918745707
51567134183215224267935955780888418997411766677983604449691049545327112018360127
56862721010965881665681274008574228137556218007157472347276882310366231153287428
90393414227090838945939699975151036281500004075438705477189761562777940953711229
20344720978570249941195753494856222496335355113341500070914900206483837650114413
19486308794026419003550913064591870027487527693765073802654682863900464126535037
46847030398846937549187409894959831291292882913809459139940065827331105367368931
14522615452826162207116348274817955572941952657616714150555124862263651922390721

No kidding! ... 2,560 digits.

It is inconvenient to deal with such large integers. Thus it is clear that doing the multiplications to get the 1936th power then reducing this large result modulo 42,987 is a difficult route to get $A_2$. Since the power is a relatively small number, we could alternately do one multiplication, reduce this partial result modulo 42,987 and then repeat this process until we have reached the 1936th power. By this route we will never have to produce a partial result that exceeds (42,987-1)x21=9Ø2,7Ø6. This 6-digit integer is within the range of a pocket calculator. However, multiplying 1936-1=1935 times and reducing the result modulo 42,987 each time is still more tedious than necessary.

At this point it is useful to note that we can reach large powers rapidly by repeated squaring. That is, to raise a number 'a' to the power $2^b$ we just square 'a' 'b' times:

$$a^{16} = a^{(2^4)} = (((a^2)^2)^2)^2$$

That is, we can get the 16th power by 4 multiplications rather than by 16-1=15 multiplications.

| | 1Ø | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | Ø |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1936 in binary is | 1 | 1 | 1 | 1 | Ø | Ø | 1 | Ø | Ø | Ø | Ø |

$$\text{Thus} \quad a^{1936} = a^{2^{10}} \cdot a^{2^9} \cdot a^{2^8} \cdot a^{2^7} \cdot a^{2^4}$$

which we can calculate by only (1Ø+9+8+7+4)+4=42 multiplications and by far fewer multiplications if we can use the calculation of the smaller powers as a start towards the larger powers.

All of this can be consolidated in a short BASIC language program that calculates integer powers by converting the power to binary and accumulating the power by the squaring method while building the larger terms from the smaller:

```
2000 'Integer Power Subroutine
2020 'Calculates A=BASE^PWR mod M
2040 'Alters BASE & PWR
2060 '
2080 A=1' initial partial result
2100 IF PWR MOD 2 = 1 THEN A = (A*BASE) MOD M
     ' update partial result if next bit of PWR is one
2120 PWR=INT(PWR/2)' right shift pwr
2140 IF PWR=0 THEN RETURN' done if all bits of PWR have been used
2160 BASE=(BASE*BASE) MOD M' square again
2180 GOTO 2100
```

By the use of this program or other means we obtain

$$A_2 = 4095 \quad **$$

And similarly we get $A_1 = \left(\frac{42,987}{21}\right)^{12}$ modulo 42,987

$= 2047^{12}$ modulo 42,987

$= 38,893$ **

Note that $A_1+A_2 = 42,988 = 1$ modulo 42,987 (as is expected for valid coefficients). Of course, if we were absolutely sure of the value for $A_2$, we could have obtained $A_1$ from $42,988-A_2$.

Thus, if the error location was known to be 16 mod 21 and 1128 mod 2047 then the location is

$$L = (38,893*16+4095*1128) \text{ modulo } 42,987$$

$= \underline{40,021}$ (Checking: $40,021 = 1,905*21+\underline{16}$ & $40,021 = 19*2047+\underline{1128}$ )

** Note: Am9520/Am8065 data sheets show $A_1$ and $A_2$ as above but multiplied by 8 for convenience in using byte counts rather than bit counts in calculating error locations.