Using the AMD 2901/2909 as a Dedicated Floating Point Adder

by

Jonathan W. Mills
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 261

Using the AMD 2901/2909 as a
Dedicated Floating Point Adder

by

Jonathan W. Mills

September, 1988

Using the AMD 2901 / 2909 as a Dedicated Floating Point Adder

Jonathan W. Mills

1. Introduction

Typical solutions to the floating point adder problem assume that the microcode will be run on a homogeneous bit-slice machine: one where all slices execute the same microcode. This results in some long and involved microcode sequences that do not exploit the parallelism that is available to the user of bit-slice products.

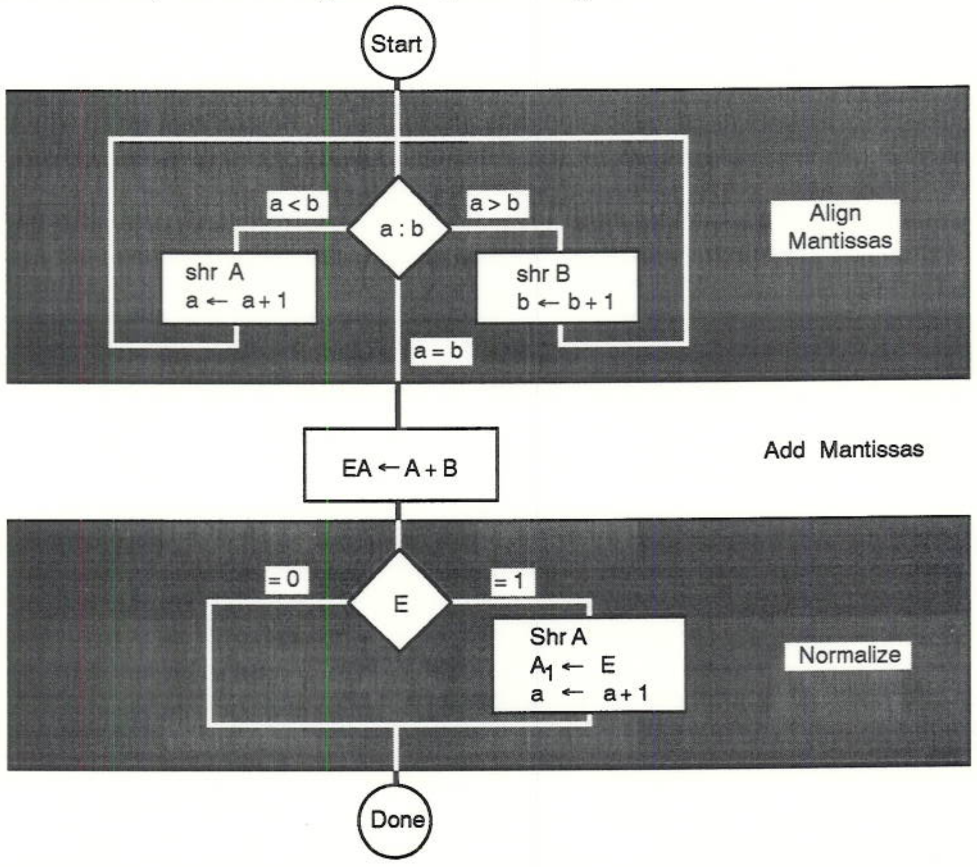
In this solution I shall treat the AMD 2901 as a "piece of logic", and build a floating point adder that depends much more on the interconnection between chips, and upon microcode as "software wire-wrapping" rather than low level assembly language.

2. Simplification of the Algorithm

The first thing to do is to examine the problem statement, and compare it to the algorithm shown in Figure 10-8 of the handout. As it turns out, the restrictions simplify things considerably:

a. Only addition is to be performed,
b. Only 16-bit numbers are implemented, and
c. Addend and augend are greater than zero.

This leaves us with only the following relevant parts of Figure 10-8:



Looking at this algorithm it is apparent that the format of the floating point numbers used can be simplified, especially if we have only 16 bits to apportion. No sign bit is necessary — addend and augend are both > 0. If we allow the fact that a 2901 bit slice is 4 bits wide to influence us, then we can divide the number into a 4 bit exponent and a 12 bit mantissa. The exponent is expressed in excess-8 format to avoid manipulating an exponent sign bit, and allow easy comparisons over the entire exponent range.

3. Key Design Decisions

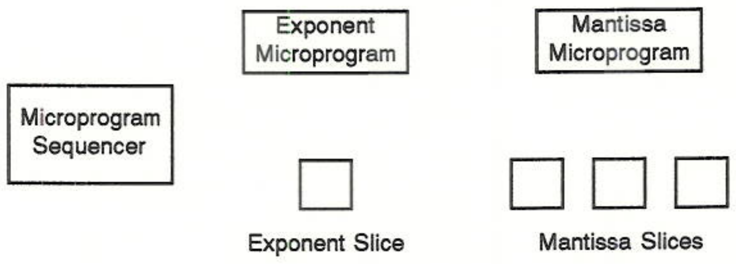
We will make a number of decisions that will influence the design. They are summarized below :

a. This is a dedicated piece of special-purpose hardware,
b. Some extra "glue" logic is acceptable,
c. The device operates correctly only under the conditions shown in paragraph (2),
d. Exponent and mantissa slices are not homogeneous,
e. Parallelism available from the hardware will be exploited.

Decisions (a), (b) and (c) allow us to wire the 2901 bit slices in an arbitrary, application specific way. There is no need to connect the slices to allow the full range of arithmetic and logical operations on general operands. Any "tricks" that we can play to obtain high performance (raw speed) will be acceptable. As will be seen later, we shall play some.

Probably the most interesting decisions are (d) and (e). As a consequence we shall microprogram the exponent slice using one control word, and the mantissa slices with a separate control word. This gives a wider, more horizontal control word, but one that we can still control with a single microprogram sequencer. This organization allows for fine-grained parallelism, and high speed operation.

The overall design of the floating point adder (FP adder) is shown below :



What the figure does not show are the interconnections between the devices. These will be described as we find a need for them in the following sections.

4. Implementing the Algorithm in Hardware

A parallel microprogram for heterogeneously microcoded exponent and mantissa bit slices will overlap processing between microinstructions. For example, after aligning the mantissas it would be efficient to detect the zero condition resulting from equal exponents, and use the flag to trigger adding the mantissas. Thus we could overlap, or "fold", the addition into the microprogram loop that aligns mantissas. To do this we must find some way of making one microinstruction do three different things — for both the exponent and the mantissa !

A "horizontal algorithm" corresponding to the simplified flowchart is shown below :

Step	Exponent Microprogram			Mantissa Microprogram		
1	Compare the exponents a and b.			Do nothing to the mantissas		
2	If $a < b$ then $a \leftarrow a + 1$	If $a > b$ then $b \leftarrow b + 1$	If $a = b$ then do nothing	If $a < b$ then shift A right	If $a > b$ then shift B right	If $a = b$ then A + B
3	If overflow from A + B in step 2, then $a \leftarrow a + 1$			If overflow from A + B in step 2, then shift A right inserting 1 into msb A		

Step 1 must select one of the three actions in step 2, and it must do so in both control words. This suggests that part of each control word's input will be derived from the signals generated by the exponent's 2901 bit slice. The arithmetic result flags are used to detect the conditions named in the microprogram; let's try to use them as control signals.

Let us choose R1 for a and A, and R0 for b and B (we'll see why this choice was made a little later). Now let's program step 1 for the exponent :

A	B	I_{210}	I_{534}	Operation	I_{876}		Dest	Description
		R, S	R op S					
0	1	1	1	▨	1		X	exponent b - exponent a, discard result but set flags
R0	R1	A, B	S - R	R1 - R0	Z	C	OVF	
				a - b				
b	a		tie carry-in high	a = b	1	0		carry out (C) selects value to adjust
				a > b	0	0		
				a < b	0	1		

and the mantissa :

A	B	I_{210}	I_{534}	Operation	I_{876}		Dest	Description
		R, S	R op S					
0	1	3	1	▨	3		B	$B \leftarrow B - 0$
R0	R1	0, B	S - R	R1 - 0	Z	C	OVF	effectively a "no-op" for mantissa
B	A		tie carry-in high					don't confuse 2901 A & B with mantissa A & B

Programming step 2 will be very "tricky". For the exponent we will want to change two fields based on the status of the zero (Z) and carry-out (C) flags:

1. The B-bus will address either R0 or R1, and
2. The destination will either be the B-bus register, or nothing (result discarded).

For the mantissa we will want to condition four fields:

1. The B-bus will address either R0 or R1,
2. The ALU source pair will be either A,B or 0,B,
3. The result will either be right shifted 1 bit before being stored in the B-bus register, or it will be stored in the B-bus register without shifting, and
4. The function will either be A + B or B - 0.

Let's tentatively microcode step 2 for the exponent :

A	B	I_{210}	I_{534}	Operation	I_{876}		Dest	Description
		R, S	R op S					
0	0 / 1	3	0	▨	0 / 2		X / B	increment either exponent a or exponent b, or do nothing
R0	R0 or R1	0, B	R + S tie carry-in high	increment (R0 or R1)	Z	C	OVF	

and the mantissa:

A	B	I_{210}	I_{534}	Operation	I_{876}		Dest	Description
		R, S	R op S					
1	0 / 1	1 / 3	0 / 1	▨	3 / 4			shift right either mantissa A or B, or add mantissas (we will condition add by Z = 1 so that we will always get R0 + R1) 3 = B, 4 = result shifted right 1 bit into B
R1	R0 or R1	A, B or 0, B	R + S or S - R tie carry-in high	R0 + R1 or shift right (R0 or R1)	Z	C	OVF 0 / 1 if add	

Finally, let's microcode step 3. This step will normalize the mantissa if an overflow occurred when the mantissas were added. Thus, we will condition fields in this step based on the <u>mantissa's</u> overflow (which we take from the most significant mantissa carry-out).

For the exponent we will increment exponent a and write it back if the mantissa overflowed:

A	B	I_{210}	I_{534}	Operation	I_{876}		Dest	Description
		R, S	R op S					
0	1	3	0	▨	0 / 2		X / B	either increment exponent a or do nothing
R0 b	R1 a	0, B	R + S tie carry-in high	increment R1	Z	C	OVF	

The mantissa will be shifted right once, inserting a one in the most significant bit if the mantissa addition overflowed :

A	B	I_{210}	I_{534}	Operation	I_{876}		Dest	Description
		R, S	R op S					
1	1	3	1	▨	3 / 4			either pass mantissa A or shift it right 1 bit
R0 B	R1 A	0, B	S - R tie carry-in high	R1 or R1 shifted right 1 bit	Z	C	OVF	3 = B, 4 = result shifted right 1 bit then put into B

The next step — developing "glue" logic — took a couple of hours since an attempt was made to develop a boolean equation for all of the possibilities in the tentative microcode. The key observation here was that what we really want to do is sequence the inputs to certain I-bus and B-bus lines, thus generating the conditional microcode. This sequencing can be done trivially using a dual 1-of-4 multiplexer with selection done using microprogram address lines A_0 and A_1 (because we have only 3 microcode words we can even "splurge" with an extra multiplexer input and use it in the design). Such multiplexers (e.g., 74LS253) are cheap, costing about 25¢, and simple to wire. We will use two in the design.

Let's assume that the Z, C and OVF flags are not internally latched on the 2901, and become stable sometime during each microinstruction cycle. Let's also assume that the state of these flags is stable enough to be latched on the rising edge of each microinstruction clock pulse. We can latch the state of the flags for use in the next microinstruction with D flip-flops — even cheaper than multiplexers! — of which we'll need three. This translates into 2 74LS74's or something similar.
We will need a few inverters and OR gates, but will assume that they are available as spares elsewhere in the design. All in all, we will use no more than 6 chips as "glue".

The circuit is shown on the next page. To be sure that the logic works, let's determine the microcode bits that the circuit generates, and see if they match those needed for the conditional microcode. The table below gives the 2901 and "glue" logic behavior :

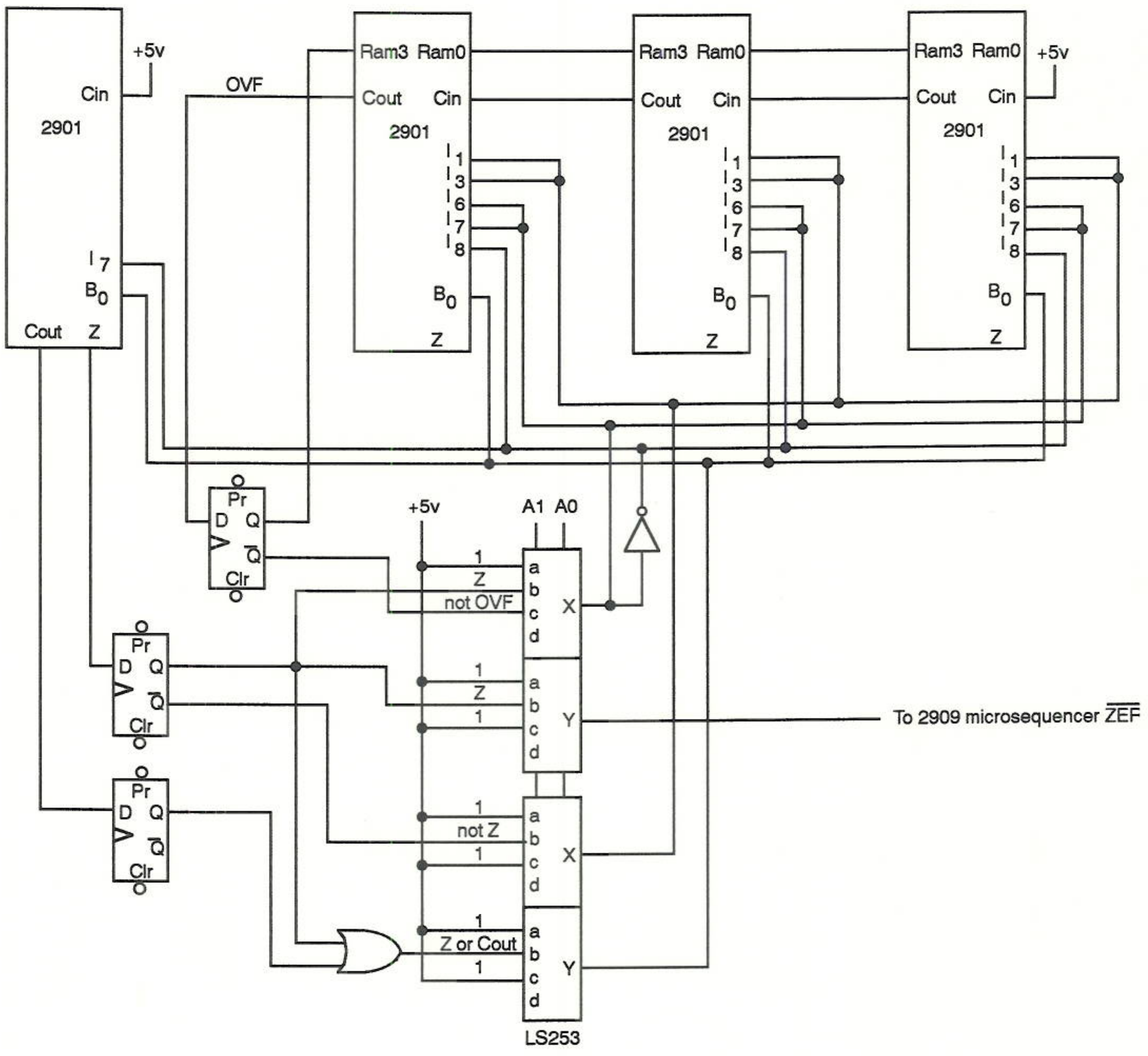
Conditions				Exponent				Mantissa							
Addr	Z	C	OVF	B_0	B	I_7	Dest	B_0	B	I_1	Src	I_3	Fxn	I_{876}	Dest
0	x	x	x	1	R1	0	x	1	R1	1	0, B	1	S - R	3	B
1	0	0	x	0	R0	1	R0	0	R0	1	0, B	1	S - R	4	shr B
1	0	1	x	1	R1	1	R1	1	R1	1	0, B	1	S - R	4	shr B
1	1	0	x	1	R1	0	x	1	R1	0	A, B	0	R + S	3	B
2	x	x	0	1	R1	0	x	1	R1	1	0, B	1	S - R	3	B
2	x	x	1	1	R1	1	R1	1	R1	1	0, B	1	S - R	4	shr B

As can be seen from this table, the 2901 conditional microcode can be generated from the circuit given. Certain bits in each microinstruction are provided by feeding 2901 latched conditions from the previous microinstruction through the multiplexers. This means we need only three microinstructions for our floating point add.

Branching is taken care of by feeding either 1 or the zero flag (Z) into the 2909 microsequencer's NOT ZERO input, and using it to force a transfer back to address 0 — the start of the addition — if the mantissas are not normalized. As soon as the mantissas are equal, and Z = 1, then the 2909 will allow the final microinstruction to execute. Note that we only need one 2909 slice!

Will this solution give us speed? Well, if we assume a random distribution of numbers to be added, then the mean number of alignment shifts will be approximately 4. Thus an average floating point add will take 9 microinstruction cycles. At worst, an add will take 23 cycles, and at best only three cycles will be needed. This is a substantial improvement over the more general solution, and shows how fast a dedicated processor may be. Remember, though, that this design is good for one and only one thing. If you need more flexibility, this is not the way to go ... but isn't it fun to design little hot-rods like this?

Dedicated AMD 2901 Floating Point Adder



2901
+5v
Cin
Cout
Z
I 7
B0
OVF
Ram3 Ram0
Cout Cin
I 1
I 3
I 6
I 7
I 8
Pr
D
Q
$\overline{Q}$
Clr
+5v
A1 A0
1
Z
not OVF
a
b
c
d
X
Y
not Z
Z or Cout
To 2909 microsequencer $\overline{ZEF}$
LS253