Floating-Point Unit in standard cell design with 116 bit wide dataflow

Guenter Gerwig and Michael Kroener
IBM Deutschland Entwicklung GmbH
S390 Development
71032 Boeblingen, Germany
ggerwig@de.ibm.com mkroener@de.ibm.com

Abstract

The floating-point unit of a S/390 CMOS microprocessor is described. It contains a 116 bit fraction dataflow for addition and subtraction and a 64 bit-wide multiplier. Besides the register array, there are no other dataflow macros used; it is fully designed with standard cell books and is placed flat with a timing driven placement algorithm. This design method allows more 'irregular' structures than usually found in custom designs.
An overview of the floating-point unit is given and some interesting design items are shown: a 120 bit-wide true-complement adder with precounting of leading zero digits, a signed multiplier with bit-optimized Wallace tree, intensive forwarding in source equal target cases and the checking method.

1. Introduction

This paper describes the floating-point unit of a high performance microprocessor, where the microprocessor is optimized for commercial workloads. There are two slightly different design points described herein.
The first is the floating-point unit for the IBM Mainframe G3 (CMOS Generation 3) which supports only IBM hex floating point formats.

Technology	CMOS 5X ($L_{eff} = .25\mu$ m)
Cycle time	5.9 ns
Chip size	14.5 * 14.5 mm
System with	12 PU's
Transistor Count	7.2 Million Transistors
	(abot 1 Million for FLPT)
Latchcount	6600 Latches for FLPT

The second is a newer floating-point design which supports binary floating-point formats according to ANSI/IEEE Std 754-1985, including an extended format of 128bits with 1 sign bit, 15 exponent bits and 112 fraction bits as well. This design was not continued due to the selection of an internal design alternative (no floating point related reasons). Figure 1 shows the the supported operand formats.

2. Dataflow Overview

Figure 2 shows the main dataflow. At the top of the figure there is the FLPT register array with 16 registers of 64 databits (plus 8 parity bits) each. For instructions with extended operands, register pairs (0 and 2, 4 and 6, ..) are addressed. In one cycle, two register pairs of 128 bits can be read and one register pair can be written. The FLPT register array has a write-through capability.
The following dataflow is separated in the main add-sub path of 116/120 bit width and an extra multiplier path with 64 bit input width.
In the add-sub path, the alignment can be done for the FA operand as well as for the FB operand and therefore, no switching is necessary. Before the digit alignment, a conversion from binary to hex format is done. In the case of an effective subtract, the operand into FC is inverted and the following true-complement adder delivers directly the correct result, no recomplementing is necessary afterwards. This adder includes the precounting of leading zero digits and generates the Zero Digit Count ZDC. The ZDC is needed as shift amount for the normalizer and also for exponent correction.
After the normalizer, rounding is done for binary instructions, including format conversion from hex to binary format. In the next cycle, the result in the FF register can be written to the FLPT register array. A write-through into FA, FB, MA, MB, EA and EB is possible.
In the multiplier path, we have two input registers of 64 and 65 bits. The multiplier uses a radix 4 Booth decoding, storing intermediate partial sum bits and partial carry bits in the PS and PC registers. An extra adder in the multiplication path generates then the raw multiplier result.
The FO register (Floating-point Output register) is normally used for store instructions to write data to the L1 cache or to the general purpose registers. A store aligner (STALI) is

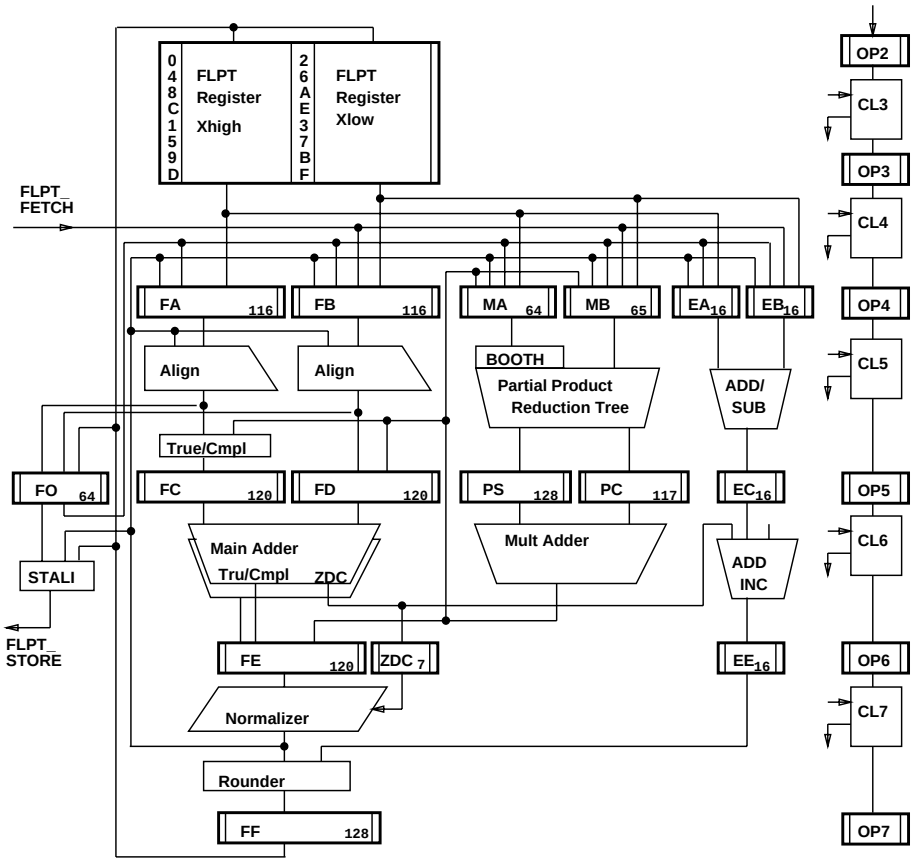

Figure 2. Main Dataflow of Floating-Point Unit

available when the operand is not at a double word boundary in the memory.
The exponent path with EA, EB, EC and EE registers are of 16 bit width to allow the required calculation of the 15 bit exponent plus an extra bit to be able to handle intermediate overflow and underflow conditions.
On the right column of Figure 2, the principal control flow is shown. A 16 bit opcode and 16 bit register fields are available in each pipeline stage. The outputs of the control logic are saved in latches and are activated in the next cycle.

3. Main Adder

The main adder is built as a 120 bit wide true-complement adder with precounting of leading zero digits.

3.1 True-Complement Implementation

The true-complement implementation prevents recomplementation cases. When there is an effective subtract, we do not know which operand is bigger. Therefore, the result of the mantissa subtraction could be a negative number (two's complement). However, we always need a positive mantissa in the floating point format, so we would need to recomplement the result. This would complicate the design and normally require an extra cycle.
The output of the adders before the final selection are:

$$Add\_out_{true} = FC + FD + Carry\_in \quad (1)$$
$$Add\_out_{cmpl} = \overline{FC} + \overline{FD} + Carry\_in \quad (2)$$

When $Operand\_A$ is in the FA register and $Operand\_B$ is in the FB register and an effective subtraction has to be executed (depends on instruction and signs), then $Operand\_A$ from FA is inverted before the FC register and $Operand\_B$ is moved to the FD register. Alignment is done for the smaller operand and the $Carry\_in$ is set to '1'.
When $Operand\_A \leq Operand\_B$ (indicated by $Carry\_out_{true}$ getting active), then Add_out_{true} is selected, otherwise Add_out_{cmpl} is taken. The following equations show that this leads to the correct result. The conversion uses the fact that
$-X$ is equal to $\overline{X} + 1$ for two's complement numbers. FC is loaded with $\overline{Operand\_A}$ and FD is loaded with

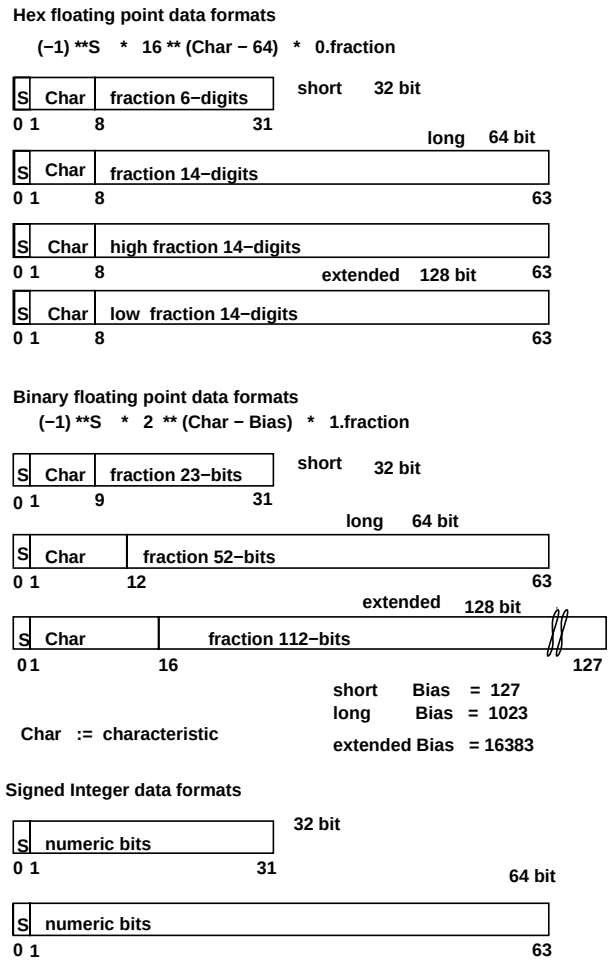

Figure 1. Supported operand formats

$Operand\_B$.

$$Add\_out_{true} = FC + FD + Carry\_in \quad (3)$$

$$Add\_out_{true} = \overline{Operand\_A} + Operand\_B + 1 \quad (4)$$

$$Add\_out_{true} = Operand\_B - Operand\_A \quad (5)$$

$$Add\_out_{cmpl} = \overline{FC} + \overline{FD} + Carry\_in \quad (6)$$

$$Add\_out_{cmpl} = \overline{\overline{Operand\_A} + \overline{Operand\_B}} + 1 \quad (7)$$

$$Add\_out_{cmpl} = Operand\_A - Operand\_B \quad (8)$$

We can see that the true adder delivers $Operand\_B$ - $Operand\_A$ and the complement adder delivers $Operand\_A$ - $Operand\_B$.

3.2 Precounting

To normalize the floating-point result, the zero digit count (ZDC) of the leading zeros is needed. With that the normalizer is driven and the exponent is decremented by the ZDC amount. To get rid of this potential timing critical path, precounting is done within the main adder. Precounting satisfies the timing requirements and delivers a exact zero digit count. The alternative of a leading zero anticipator (LZA) could be somewhat faster and require somewhat less hardware, but would be imprecise by one digit.

Figure 3 shows the principal function of the precounting adder on a 64bit example. Since the dataflow is organized for hex format, zero digits with four bits are counted, but the principle is also working for binary digits. In addition a masking for the range of counting is done, but not shown on the example for simplicity reasons. Naming Definitions for figure 3:

Digit	:=	Group of four bits
A	:=	Mantissa A (64bit)
B	:=	Mantissa B (64bit)
Cin	:=	Carry Input
G	:=	Generate
P	:=	Propagate
GD	:=	Generate for Digit
PD	:=	Propagate for Digit
GG	:=	Generate for 16bit (4digits)
PP	:=	Propagate for 16bit (4digits)
FZ	:=	Fraction Zero signal for 16bit (4digits)
CS	:=	Carry select signal for 16bit block
SUM	:=	Result of addition
ZDC	:=	Zero Digit Count
x	:=	related to case 'A + B + 0'
y	:=	related to case 'A + B + 1'
Cout	:=	Carry out
r	:=	value of digit unequal zero

In each 16 bit block of figure 3 the SUM, zero digit count ZDC and the fraction zero signal FZ are built twice.
In the first case it is assumed that no carry is coming into that block.

SUMx = A + B + 0

In the second case it is assumed that a carry is coming into that block.

SUMy = A + B + 1

The internal details of an ADD16 block are not shown, but a hierarchical structure can be used where the ADD16 block looks similar to the figure for the 64-bit adder, but having four subblocks of ADD4.
In parallel, the carry select signals CS0-3 are built in an extra propagate-generate network (block PROP-GEN) according to the following equations:

$$g_i = a_i b_i \quad (9)$$

$$p_i = a_i + b_i \quad (10)$$

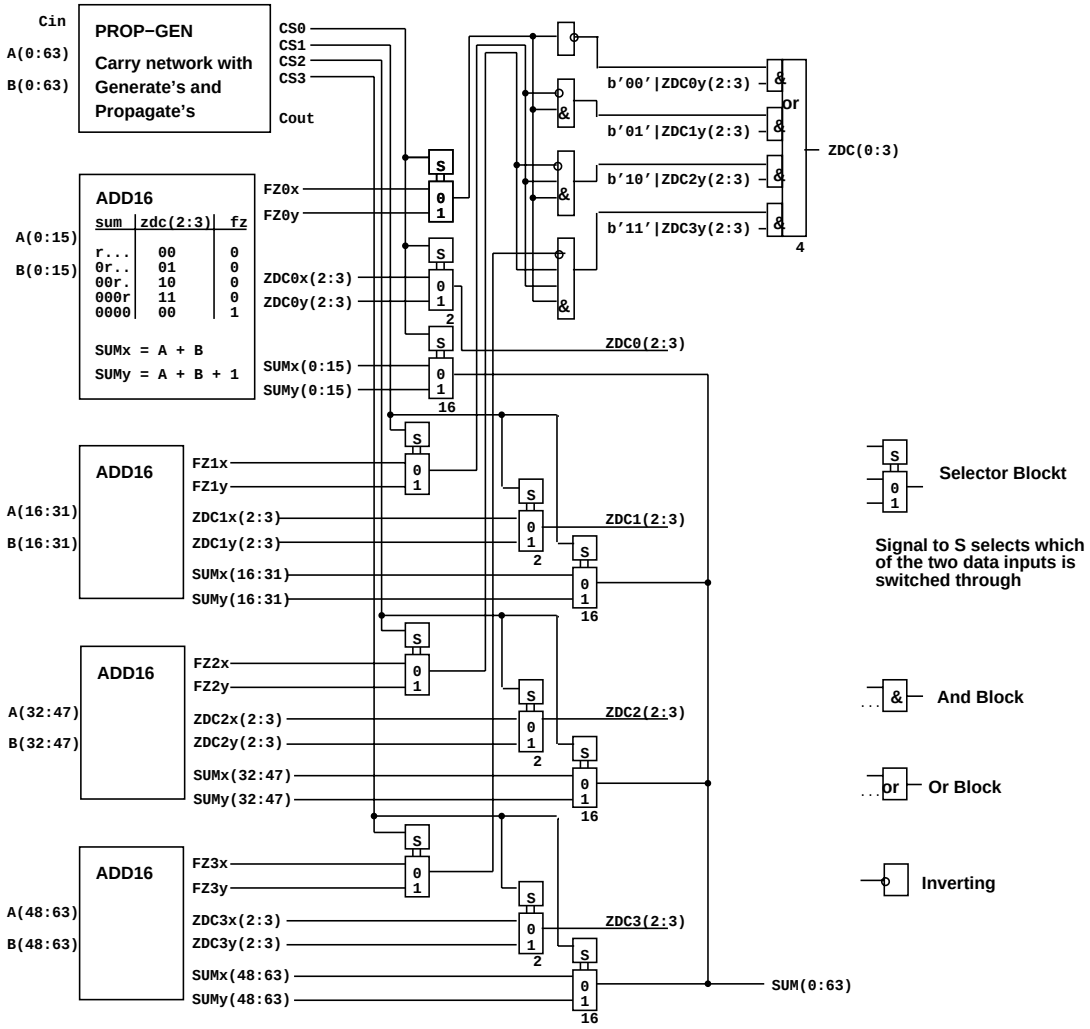

Figure 3. Adder with precounting (64 bit example)

$$
\begin{aligned}
gd_j &= g_n + p_n g_{n+1} \\
&\quad + p_n p_{n+1} g_{n+2} \\
&\quad + p_n p_{n+1} p_{n+2} g_{n+3} && (11)\\
pd_j &= p_n p_{n+1} p_{n+2} p_{n+3} && (12)\\
gg_k &= gd_m + pd_m gd_{m+1} \\
&\quad + pd_m pd_{m+1} gd_{m+2} \\
&\quad + pd_m pd_{m+1} pd_{m+2} gd_{m+3} && (13)\\
pp_k &= pd_m pd_{m+1} pd_{m+2} pd_{m+3} && (14)\\
Cout &= gg_0 + pp_0 gg_1 + pp_0 pp_1 gg_2 \\
&\quad + pp_0 pp_1 pp_2 gg_3 + pp_0 pp_1 pp_2 pp_3 Cin && (15)\\
CS0 &= gg_1 + pp_1 gg_2 + pp_1 pp_2 gg_3 \\
&\quad + pp_1 pp_2 pp_3 Cin && (16)
\end{aligned}
$$

$$
\begin{aligned}
CS1 &= gg_2 + pp_2 gg_3 + pp_2 pp_3 Cin && (17)\\
CS2 &= gg_3 + pp_3 Cin && (18)\\
CS3 &= Cin && (19)\\
i &= 0..63 && (20)\\
j &= 0..15 && (21)\\
k &= 0..3 && (22)\\
n &= 4j && (23)\\
m &= 4k && (24)
\end{aligned}
$$

These CS0-3 signals select the final SUM and FZ and the potential ZDC of each ADD16 block. With the FZ signals then the final ZDC is selected as shown in figure 3.

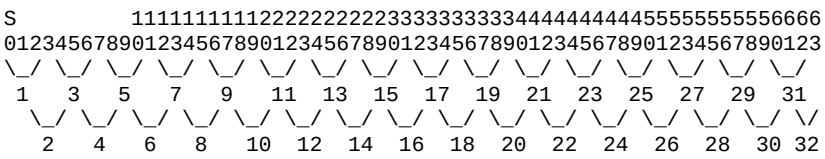

Figure 4. Booth Scanning of the Multiplier operand

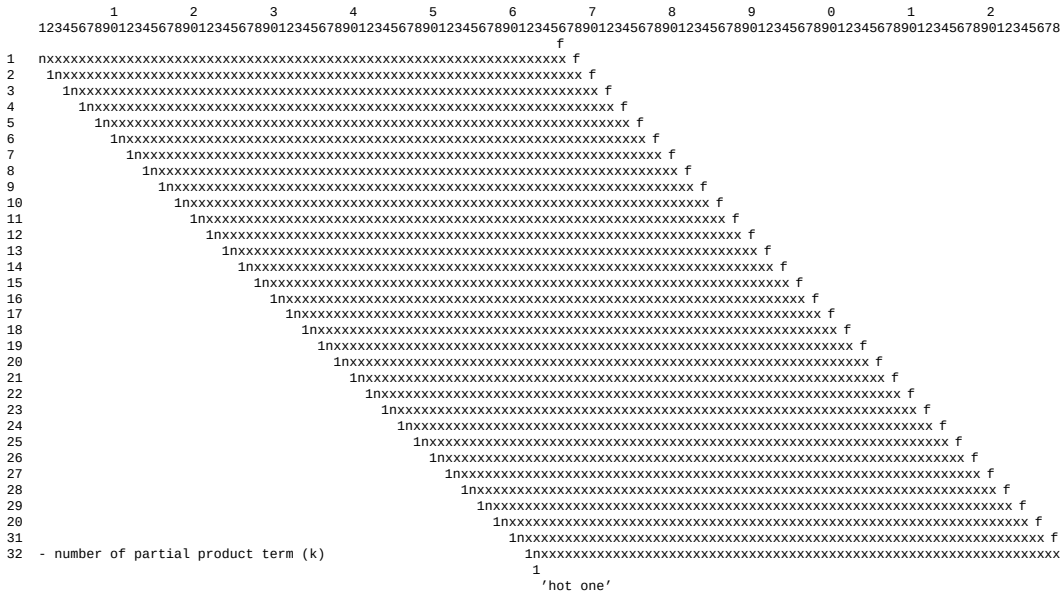

Figure 5. Partial product terms after Booth decoding

4. Multiplier

A signed multiplier with S+63 bits times S+64 bits is implemented completely in standard cell design.
The multiplier operand consists of Y_i bits, where i=0..63 and Y_0 is the sign bit.
The multiplicand operand consists of X_i bits, where i=0..64 and X_0 is the sign bit.
A radix 4 Booth algorithm is used, which creates 32 partial products of 66bits + '1'.
For each scan, there are 3 bits which can be represented by Y_{i-1}, Y_i, and Y_{i+1} from most significant to least significant (see figure 4).
The equations for the select signals of the k-th partial product are as follows.

$$spos_k = \overline{Y_{i-1}} \quad (25)$$

$$s1x2_k = Y_i \oplus Y_{i+1} \quad (26)$$

$$stru_k = (Y_{i-1} + Y_i + Y_{i+1}) \quad \overline{Y_{i-1}\, Y_i\, Y_{i+1}} \quad (27)$$

$$i = 2k - 1 \quad (28)$$

$$k = 1..32 \quad (29)$$

The select signals are additionally shown in Table 1. The $spos_k$ signal is used to invert the k-th partial product. If equal to one it is the positive signal, if equal to zero it is inverted.
The $s1x2_k$ signal is used to selext 1x or 2x. If equal to one the k-th partial product is used directly, if equal to zero it is shifted left 1 bit (multiply by 2).
The $stru_k$ signal is used to gate off zero cases of the k-th partial product. If equal to one the partial product is used, if equal to zero all ones are selected (together with the f-bit this results in a zero term).
This multiplier can handle signed binary integer operands directly. This leads to special handling for the so-called n-bits and f-bits. The n-bit is the sign bit of a partial product. The f-bit is the '+1' correction when the partial product is

used inverted or is zero.

$$n_k = (\overline{Y_{i-1}} \oplus X_0) \quad stru_k \tag{30}$$

$$f_k = Y_{i-1} + \overline{Y_i} \quad \overline{Y_{i+1}} \tag{31}$$

The partial products including n-bits, f-bits and the 'hot

Y_{i-1}	Y_i	Y_{i+1}	spos	s1x2	stru	Selection
0	0	0	0	0	0	0x
0	0	1	0	1	1	+1x
0	1	0	0	1	1	+1x
0	1	1	0	0	1	+2x
1	0	0	1	0	1	-2x
1	0	1	1	1	1	-1x
1	1	0	1	1	1	-1x
1	1	1	1	0	0	0x

Table 1. Determination of Select Signals for Booth terms

one' are shown in figure 5.
The partial product reduction tree is a modified Wallace tree, which is single bit optimized. 3 to 2 counters (full adder books) are used to get 128 sum bits and 117 carry bits. These bits are saved in the PS and PC register. The 128 bit wide product of the multiplication is available after an extra carry propagate adder in the next cycle.
The flow of the partial product bits is described in a table and the netlist is generated directly out of this table with a special synthesis program, which already considers timing aspects. The powering is done with up to five priority levels. The unsymmetrical timing behaviour of the 3 to 2 counters is also considered.
For Booth decoding normal physical books like Nand, Muxes and Selects are used. The partial product reduction tree is built with full adder books (for the 3 to 2 counters).
The timing driven placement and wiring is able to emphasize on the 'few' timing critical paths, while most paths of a multiplier are only medium timing critical.

5. Control Logic

5.1. Interface to Processing Unit

The floating-point control logic gets the opcode related signals during OP1 time. The memory data from the L1 cache arrives at OP3 time on the FLPT_FETCH bus (64 bit plus 8 parity).
The synchronization between the floating-point unit and the processing unit (PU) for multi-cycle instructions or unresolvable source equal target conflicts is done by the signal FLPT_WAIT (stalls the PU). FLPT_BUSY is used to tell the

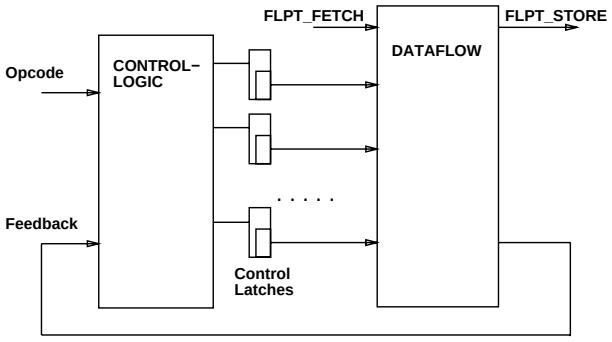

Figure 6. Latched Control Interface

PU, that a FLPT instruction is still in the pipeline and either the condition code will be set or a potential exception could arise. FLPT_STORE_PENDING tells the L1 cache that a late floating-point store is in the pipeline. In the other direction, there are different cancel conditions to stop execution of floating point instructions.

5.2. Pipeline Structure

The floating-point unit uses a strict pipeline control. Every instruction is assigned to a pipeline slot and passes regular OP2, OP3, OP4, OP5, OP6 and OP7. Multi-cycle instructions cover multiple slots of the pipeline. A five bit cycle counter allows the identification the different cycles of a multi-cycle instruction.
The control logic has a latched interface to the dataflow for about 250 different control signals. This prevents timing critical paths within the control logic (see figure 6). There is a feedback of about 30 single signals from the dataflow to influence the further control of an instruction.

5.3. Synthesis

The control logic is completely synthesized out of 'timing chart tables'. Because of the latched interface to the dataflow, the control logic is only medium timing critical, and synthesis is running without big problems.

6. Performance, Latency and Forwarding

There are about 200 hardware implement instructions supported. Most of them are pipelined executed in one cycle except divide, square root, multiply with extended input operands and binary mult-add.
For divide short and long a Goldschmidt algorithm is used with back-multiplication at the end, requiring 16-19 cycles.

Divide extended and square root using a simple restoring subtraction method, gaining 1 bit per cycle.

6.1. Latency

The latency for hex add and subtract instructions is two cycles, and for hex multiplication three cycles. Binary instructions need a latency of four cycles because of rounding. A load has always the latency of one cycle because of the intensive forwarding.

6.2. Forwarding

There is a 'bigger' part in the control logic which checks if there is a dependence between the instructions in the pipeline. First, second and third dependencies are checked and the appropriate forward path is activated, or in the case when forwarding is not possible, a wait cycle is inserted.
The forwarding paths for first dependence are:
- the hold path for the FA, FB ,MA and MB registers
- the path from multiplier to FC and FD registers
- the path from normalizer to FLPT_STORE

The forwarding paths for second dependence are:
- the path from FO register to FA, FB, MA, MB registers
- the path from normalizer to FC and FD registers
- the path from FF register to FLPT_STORE.

The forwarding paths for third dependence are:
- the path from normalizer to FA, FB, MA, MB registers
- the path from FF register to FO register.

Most of the forwarding paths allow mixed operand formats.

6.3. Linpack Benchmark

The most important benchmark for S/390 hex floating-point is the 'Linpack loop', showing here the rolled version:

1.	LD	0,6	Load register from F6 to F0
2.	MD	0,x	Multiply memory operand to F0
3.	AD	0,x	Add memory operand to F0
4.	STD	0,x	Store F0 in memory
5.	BXLE	..	Branch on index back to 1 and increment index register.

Where x is the memory operand (address calculated out of base register + index register + offset).
Since the BXLE is executed in parallel in the PU and the floating-point unit is doing intensive forwarding, only four cycles are needed for one loop path.
This gives a performance of 60 MFlops at 5.9ns for the rolled S/390 Linpack benchmark.

7. Checking

The Add-Sub fraction dataflow is completely checked with parity bits. The floating-point register array and most paths are checked with Byte parity (one parity bit per eight data bits); in some cases Digit parity (one parity bit per four data bits) is used.
The multiplier path is checked with Modulo 3 and Modulo 5 residue calculations.
For the main adder, the true and complement parts are somehow symmetrical and this is used for checking.
Within the control logic, the opcode registers are parity checked. The control signals for a certain multiplexer in the dataflow are mutually exclusive and the checking is done with an additional 'bunch control signal'.
The different checks are separated in recoverable and non-recoverable checks. Non-recoverable checks will immediately stop the execution of the one related PU. The service processor will configure this PU out and will continue with the remaining PU's.

8. Placement and Wiring

The floating point dataflow has been described with an graphical entry tool with the exception of the multiplier, where the netlist was generated with a special synthesis program. The control logic was also synthesized. These netlists were combined together with other PU netlists to a huge flat netlist.
This netlist was then placed and wired flat using timing driven algorithms. For the floating-point unit, only the register array was preplaced. All other books, including the more than 5000 latches with their clocking, were processed automatically. This includes optimizing the power level of each book. For a certain book (such as Nand 2) there were up to 6 different power levels available in the standard cell library. Additional automatic buffer insertion was done.
In figure 7 the PU chip of the G3 hardware is shown and in the lower right corner the floating-point unit can be seen. The books are grouped around the manually placed floating-point register array. In the lower right corner the multiplier found its place. Towards the center a 'cloud' of control logic can be found.
There is no strict separation of the different logic parts, sometimes they got mixed, since it was a fully flat placement.
The floating-point unit was not limiting the cycle and no manual correction to placement and wiring was necessary.

9. Summary

The G3 Floating-point unit was designed in nearly 15 months by two designers, including unit simulation and bring-up support. This indicates that standard cell design is a very effective design method.

Since the cycle time was set by the processing unit, looking for good performance in commercial workloads, the floating-point should not gate the other design in any way. Simplicity and low additional hardware requirement were the main reason for implementing an extended dataflow. The True-Complement Adder with precounting and the signed multiplier made the control easier.
The intensive forwarding is the remaining method to gain a better performance for a machine with a given relative relaxed cycle time (compared to full custom design approach).
The timing driven placement gets in some areas close to the timing results of custom design. Overall we estimate the potential timing advantage of custom design to be in the range of 20

References

[1] "Enterprise Systems Architecture/390 Principles of Operation," International Business Machines Corporation Publication No. SA22-7201-03+,

[2] "IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985," The Institute of Electrical and Electronic Engineers, Inc., New York, Aug. 1985.

[3] G. Doettling, K. J. Getzlaff, B. Leppla. "S390 Parallel Enterprise Server Generation 3: A balanced system and cache structure" *IBM Journal of Research and Development*, 41(4/5):405–27, July/Sept. 1997.

[4] B. Kick, U. Baur, J. Koehl. "Standard-cell-based design methodology for high-performance support chips" *IBM Journal of Research and Development*, 41(4/5):505–14, July/Sept. 1997.

[5] S. Vassiliadis, D. S. Lemon, M. Putrino. S/370 sign-magnitude floating-point adder *IEEE Journal of Solid-State Circuits*, vol.24, no.4, p. 1062–70, Aug. 1989.

[6] S. F. Oberman, H. Al-Twaijry, M. J. Flynn. The SNAP project: design of floating point arithmetic units *Proceedings 13th IEEE Symposium on Computer Arithmetic Asilomar*, p.156–65, July 1997.

[7] E. M. Schwarz, R. M. Averill, L. J. Sigal. A Radix-8 CMOS S/390 Multiplier *Proceedings 13th IEEE Symposium on Computer Arithmetic Asilomar*, p.2–9, July 1997.

[8] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K.Mashiko, T. Sumi. Leading-zero anticipatory logic for high-speed floating point addition *IEEE Journal of Solid-State Circuits*, vol.31, no.8, p. 1157-64, Aug. 1996.

[9] A. D. Booth "A signed multiplication technique," Quarterly J. Mech. Appl. Math., 1951.

[10] C. S. Wallace "A suggestion for a fast multiplier," IEEE Trans. Comput., Nov. 1964.

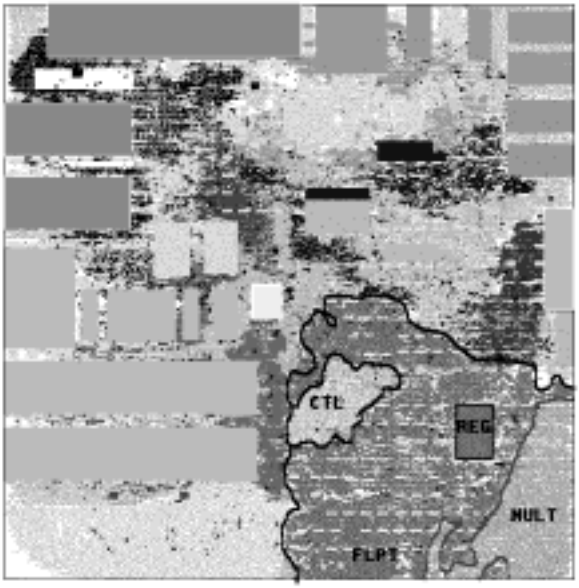

Figure 7. Layout of Microprocessor with Floating-Point-Unit