Architecture of a digital signal processor

by G. Ungerboeck
D. Maiwald
H.-P. Kaeser
P. R. Chevillat
J. P. Beraud

A digital signal processor (DSP) is described which achieves high processing efficiency by executing concurrently four functions in every processor cycle: instruction prefetching from a dedicated instruction memory and generation of an effective operand, access to a single-port data memory and transfer of a data word over a common data bus, arithmetic/logic-unit (ALU) operation, and multiplication. Instructions have a single format and contain an operand, index control bits, and two independent operation codes called "transfer" code and "compute" code. The first code specifies the transfer of a data word over the common data bus, e.g., from data memory to a local register. The second determines an operation of the ALU on the contents of local registers. A fast free-running multiplier operates in parallel with the ALU and delivers a product in every cycle with a pipeline delay of two cycles. The architecture allows transversal-filter operations to be performed with one multiplication and ALU operation in every cycle. This is accomplished by a novel interleaving technique called ZIP-ing. The efficiency of the processor is demonstrated by programming examples.

Introduction

Digital signal processors (DSPs) are microprocessors with a specialized architecture which allows them to process digitized-waveform signals efficiently. The consistency and compactness of digital circuitry together with the flexibility of programmable devices make DSPs ideally suited for applications in telecommunications, instrumentation, electromechanical control, and elsewhere. Thus, the revolution in electronic system design, started more than a decade ago by the introduction of inexpensive general-purpose microprocessors, is being extended by DSPs into domains where analog circuit implementations have traditionally prevailed [1].

The cost-effective replication of analog signal-processing functions, however, represents only one aspect of this development. The capability of DSPs to store and manipulate signals without loss in precision, and to make data-dependent decisions, allows the realization of signal-processing tasks far more complex than would ever have been possible by analog circuit means.

Advances in VLSI semiconductor technology have made DSPs feasible. Several vendors are offering single-chip DSPs, e.g., those described in [2–5]. A more complete list and comparison of DSPs can be found in [6]. Other DSP architectures have been described in [7–11]. Common to most DSPs is a pipelined parallel architecture and the use of a fast on-chip multiplier to gain the high processing speed and efficiency required for digital signal-processing applications.

This paper describes the architecture and programming of a single-chip IBM DSP developed in Zurich, Switzerland, and La Gaude and Essonnes, France, during the period from 1979 to 1981. The processor was realized in standard bipolar technology with 2.5-μm ground rules on a 25-mm^2 chip, using the macro-stack design technique developed at Essonnes. With suitable instruction and data memories provided externally, it achieves a cycle time of 100 ns. The architecture is similar to that described in [11], but somewhat simplified to keep complexity below 5000 equivalent NAND gates. A novel feature of this single-data-

bus architecture is its ability to perform transversal-filter operations with one multiplication and addition/subtraction in every processor cycle, although only one signal or coefficient value can be transferred per cycle.

The next sections deal with the design considerations and details of the architecture. Finally, programming examples are given to illustrate the operation of the processor.

Design considerations

Digital signal processing requires high levels of arithmetic throughput. The tasks most commonly performed are the accumulation of products for filtering and correlation, execution of fast Fourier transform algorithms, and searching paths in decision trees or finite-state machine diagrams. This requires not only high performance in multiplication and addition/subtraction operations, but also efficient testing and branching. Achieving these functions at high speed with limited circuit complexity was a primary design goal for the DSP described.

A Harvard architecture with separate instruction and data memories permits overlapping of instruction prefetching with instruction execution. This form of pipelining is commonly found in all modern DSPs [2–10]. Consequences in the case of branch instructions and interrupts are explained later.

The size of the instruction and data address spaces depends on the complexity of the application programs envisaged and the amount of directly accessible data required. Choosing 64K instruction and 4K data address spaces avoids the limitations in addressing capability found in many DSPs. The large address spaces also facilitate address decoding when several instruction and/or data memory modules and memory-mapped I/O ports are to be addressed.

The 4K data addressing capability and architectural simplicity desired for the DSP favored an architecture with a single data bus over which data are transferred sequentially between a single-port data memory and the arithmetic/logic elements of the processor. Data memory and data bus are 16 bits wide.

The computing section of the processor is designed around an ALU and a separate fast multiplier. These two elements are now standard in most DSPs [6]. The effect of the limited transfer capability is mitigated by performing transfers over the data bus, ALU operations, and multiplications in parallel. Instructions contain one data address or direct operand, index bits to modify the operand, and two independent "transfer" and "compute" operation codes. During the execution cycle of an instruction, the transfer code, together with the operand and index bits, determines the transfer of a data word over the data bus, e.g., from data memory to a local register in the processor. Simultaneously, the compute code controls the operation of the ALU on the contents of local registers.

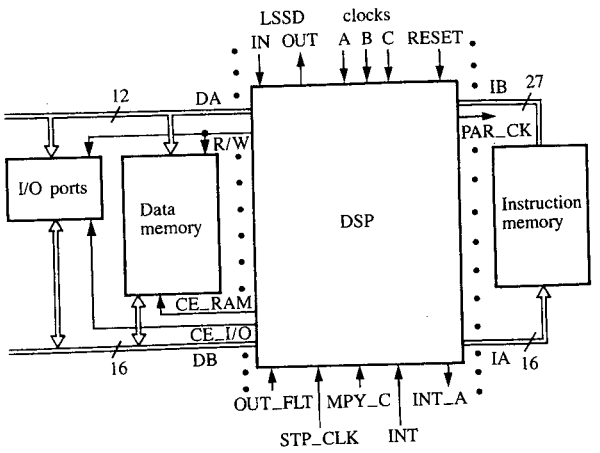


Figure 1

DSP with memories and I/O ports.

The multiplier is not controlled except for optional freezing of the multiplier operation during interrupt mode. The multiplier operates on the contents of two fixed local registers in a free-running mode. In its output register, a new product appears in every cycle and may be involved in an ALU operation during the following cycle. By introducing intermediate pipeline registers in the multiplier, the multiplication rate can be made compatible with the cycle time of the processor without requiring excessively fast circuitry. For the DSP described, one such pipeline register was chosen.

The arrangement described results in a logical pipelining of data transfers, multiplication, and ALU operations, where transfers and ALU operations are under control of transfer and compute codes in successive instructions.

The execution of transversal-filter operations by a single data-bus processor usually requires two transfer cycles for each multiplication. This limitation was overcome by the introduction of two accumulator registers and a programming technique called ZIP-ing. The technique is characterized by transferring signal and coefficient values alternately to the two inputs of the multiplier, and accumulating products emerging from the multiplier output alternately in the two accumulator registers; it is further explained by programming example 2, presented later in the paper. Other DSPs achieve the same processing efficiency only by feeding the multiplier concurrently with two data words from a dual-output-port memory [2, 5]. Since this requires two data addresses, addressing capability is then usually very limited.

Arithmetic operations are performed in fixed-point two's complement arithmetic. The multiplier forms the product of

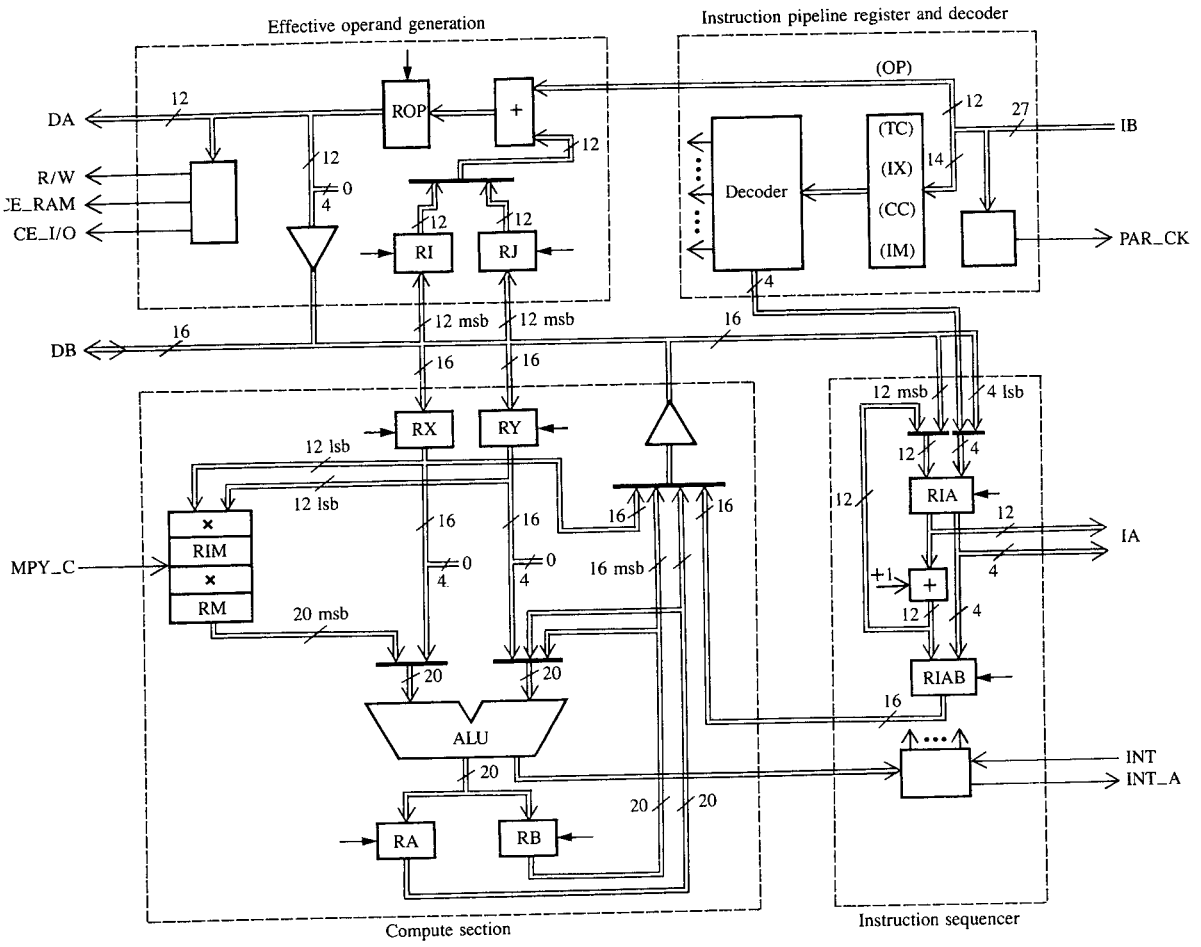


Figure 2

Block diagram of the DSP.

two 12-bit values. The ALU and two accumulator registers are 20 bits wide. By preloading accumulators with appropriate constants, rounding to 12 or 16 most significant bits is accomplished when results are stored in the 16-bit data memory. The arithmetic precision achieved is sufficient for the applications envisaged.

Architecture

Figure 1 gives a general schematic of the DSP with attached memories and I/O ports. The instruction address (IA) and the instruction bus (IB) are 16 and 27 bits wide, respectively. The data address (DA) and data bus (DB) are 12 and 16 bits wide, respectively. The interface line R/W indicates the direction of transfers over the data bus, and the lines CE_RAM and CE_I/O enable data memory and I/O ports.

The DSP is driven by two nonoverlapping clocks B and C operating at the processor cycle rate. Clock A and the lines designated IN and OUT LSSD are used only for chip testing. The interface line STP_CLK allows stopping of the processor, e.g., to extend the cycle time during access to slower memory or I/O ports. Activation of the line OUT_FLT puts the output drivers of the DSP into high-impedance state. The other interface lines are explained later.

The internal structure of the DSP is shown in **Figure 2**. Four functional entities can be identified:

- Instruction pipeline register and decoder,
- Instruction sequencer,
- Effective-operand generation, and
- Compute section.

The abbreviations msb and lsb denote most- and least-significant bits, respectively. The interface line PAR_CK indicates the parity of read instructions. The instruction format is explained in **Figure 3**. The sizes of the internal

working registers and their bit alignment relative to the 20-bit ALV are shown in **Figure 4.**

• *Instruction sequencing in normal mode*

The 16-bit register RIA holds the current instruction address. Its four lsb are page bits. Activation of the interface line RESET sets RIA to zero (hexadecimal X'000-0') and forces the DSP out of interrupt mode. When RESET is released, instruction execution starts at address zero with page bits set to zero. A processor cycle starts when a new value is loaded into RIA. While a new instruction is being fetched, the previously fetched instruction is being executed.

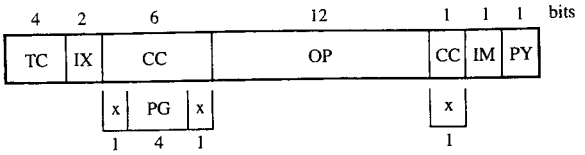
For sequential instruction sequencing, the 12 msb of RIA are incremented by one, and the four page bits remain constant. Unconditional and executed conditional branch instructions write the 12 msb from the data bus into the 12 msb of RIA. Only unconditional branch instructions also load new page bits into RIA. The new page bits come from the instruction decoder when an unconditional direct branch (TC = BRD) is executed, or from the four lsb of the data bus when an indirect branch (TC = BRM) is performed. The 16-bit register RIAB receives the incremented instruction address and last page bits.

In connection with branch instructions, the two-phase pipelining of instruction fetching and execution has the effect that the instruction following a branch instruction is still fetched and unconditionally executed. In almost all cases, this instruction can be used to accomplish a useful program function, e.g., store a result or load a register, before the branch becomes effective. Interesting deviations from this rule occur in the sequential programming of more than one branch instruction (see programming example 4, presented later in the paper).

• *Entering and leaving interrupt mode*

The processor has a one-level interrupt capability. An interrupt is requested by the line INT. Acceptance is acknowledged by the line INT_A and results in RIA being set to X'002-0'. Thus, execution of the interrupt program starts at address two, with page bits set to zero. While the first instruction of the interrupt program is being fetched, the previously fetched instruction of the interrupted program is still being executed. RIAB receives the address of the instruction to be fetched next in the interrupted program. Its contents remain frozen during interrupt mode. Execution of an unconditional indirect branch instruction using data address zero (BRM 0) loads the contents of RIAB into RIA. Interrupt mode is terminated, and instruction fetching and execution return to the interrupted program. Line INT_A is reset and the DSP is ready to accept another interrupt.

Application programs which use the interrupt capability should begin with an unconditional branch instruction at address X'000-0' to skip over the interrupt program starting at address X'002-0'.



TC – transfer code
CC – compute code
IM – interrupt mask bit
PG – page bits, for unconditional long direct branch (TC = BRD), no CC in this case, x = don't care
IX – index control
OP – operand
PY – odd parity bit

Figure 3

27-bit instruction format.

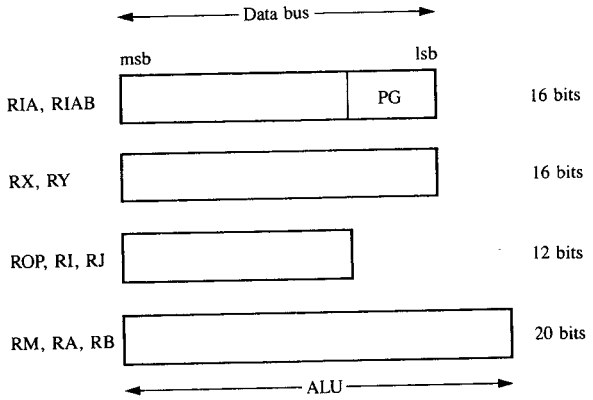


Figure 4

Internal working registers and alignment.

Interrupt acceptance can be disabled by setting the interrupt-mask (IM) bit provided in the instruction format. The IM bit becomes effective after fetching of the next instruction. It must be set in every instruction in the main program which may be followed by a branch instruction, because if an interrupt were accepted after fetching of a branch instruction, execution of the branch instruction would disturb the correct instruction-address sequencing.

The interface line MPY_C determines whether during interrupt mode the multiplier continues to operate or its registers RIM and RM remain frozen. In this case, the four lsb of RA and RB which can only stem from products are also frozen and made to appear as zeros during interrupt mode. Most applications do not need the multiplier in interrupt mode. However, if one wants to use the multiplier during interrupt mode, interrupt acceptance must be disabled in the main program during instruction sequences

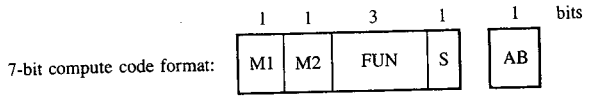


Figure 5 Format of the compute code.

Table 1 Transfer codes.

TC	*Mnemonic*	*Function*
0000	LMX	Load ⟨DM(ROP)⟩ into RX
0001	LMY	Load ⟨DM(ROP)⟩ into RY
0010	LMI/LMJ	Load ⟨DM(ROP)⟩ into RI or RJ *
0011	BRM	Branch unconditionally to ⟨DM(ROP)⟩
0100	LDX	Load ⟨ROP⟩–0 into RX
0101	LDY	Load ⟨ROP⟩–0 into RY
0110	LDI/LDJ	Load ⟨ROP⟩–0 into RI or RJ *
0111	LDYI/LDYJ	Load ⟨ROP⟩–0 into RY and RI or RJ *
1000	SAM	Store 16 msb of RA into DM(ROP)
1001	SXM	Store RX into DM(ROP)
1010	SBM	Store 16 msb of RB into DM(ROP)
1011	SIAM	Store RIAB into DM(ROP)
1100	BRZD	Branch if ALUOUT zero to ⟨ROP⟩
1101	BRAD	Branch if ALUOUT non-zero to ⟨ROP⟩
1110	BRND	Branch if ALUOUT negative to ⟨ROP⟩
1111	BRD	Branch unconditionally to ⟨ROP⟩–PG

⟨...⟩ — contents of ...
DM(ROP) — data memory location addressed by ⟨ROP⟩
* — RI or RJ, depending on second bit of IX
⟨ROP⟩–0 — ⟨ROP⟩ with four zero lsb appended
ALUOUT — 16 msb of ALU output
⟨ROP⟩–PG — ⟨ROP⟩ with four page bits from instruction decoder appended

which use the multiplier. For this reason a dedicated IM bit was provided in the instruction format.

• *Generation of the effective operand*
Instructions contain a 12-bit operand (OP) and a two-bit index control field (IX). Under control of IX, OP either remains unchanged (IX = '0x') or is incremented by the contents of the 12-bit index register RI (IX = '10') or RJ (IX = '11'), using a fast adder. The resulting effective operand is still generated during the fetch cycle of an instruction and stored in the 12-bit register ROP. During the subsequent execution cycle, the contents of ROP either serve as a data address or are gated to the data bus as a direct operand with four zero lsb appended.

• *Data transfers over the data bus*
During the execution cycle of an instruction, its decoded four-bit transfer code (TC) causes one of four types of transfers over the data bus:

Table 2 Compute codes.

M1	*M2*	*FUN*							
		000	*001*	*010*	*011*	*100*	*101*	*110*	*111*
0	0	Y	Y–X	X–Y	X+Y	X|Y	X&Y	X.Y	RDH
0	1	X	C–X	X–C	X+C	X|C	X&C	X.C	CLR
1	0	M	Y–M	M–Y	M+Y	M|Y	M&Y	M.Y	RDL
1	1	ONE	C–M	M–C	M+C	M|C	M&C	M.C	SLC

- Read from data memory (or an input port, or RIAB) and load RX, RY, RI, or RJ.
- Load RX, RY, RI, or RJ by direct operand.
- Branch direct or indirect (load RIA by direct operand or value from data memory).
- Store contents of RX, RA, RB, or RIAB into data memory (or an output port, or an internal register).

Table 1 summarizes the transfer codes.

• *Data memory, I/O port, and internal-register addressing*
Accesses to data addresses above 31 activate the line CE_RAM. Accesses to data addresses 1–31 enable the line CE_I/O. In addition, the value on the data bus is placed in the register RX or RY by a write operation to data address 1 or 2, respectively, and RI or RJ is loaded by the 12 msb of the data bus by a write operation to data address 4, where the concurrent IX field determines RI or RJ. Data address 0 represents a special case. The contents of RIAB are placed on the data bus by a read operation from data address 0.

• *Arithmetic/logic unit and multiplier*
The ALU and multiplier operate in parallel and independently of the data transfers occurring during the same cycle.

In every machine cycle, the 12 × 12-bit multiplier accepts the 12 msb from registers RX and RY. The pipeline register RIM partitions and distributes the circuit delay of the parallel multiplier logic over two cycles. Thus, a 23-bit product truncated to 20 msb appears in the 20-bit output register RM of the multiplier in every processor cycle with a pipeline delay of two cycles. During the subsequent cycle, the product may be involved in an ALU operation, as specified by the compute code.

During the execution phase of an instruction, the seven-bit compute code (CC) connects registers to the inputs of the 20-bit ALU, selects the ALU function, and controls whether the result is to be stored in the 20-bit accumulator register RA or RB. Thus, compared to the 16-bit width of data bus and memory, products from the multiplier output register RM can be accumulated with four more lsb. Besides providing higher precision, this also permits more flexibility for scaling. For input to the ALU, the contents of 16-bit

registers RX and RY are extended by four zero lsb. The format of the compute code is shown in **Figure 5**.

The functions of M1, M2, and FUN are summarized in **Table 2**. Bit S, if set, causes the ALU output to be stored in RA or RB, depending on the value of the accumulator control bit AB. In Table 2, e.g., mnemonic X+Y indicates that the sum of the contents of RX and RY appears at the ALU output. The symbol | denotes the logic OR, & the logic AND, and . the logic EX-OR function. The letter C stands for A or B, depending on the value of bit AB. Besides six arithmetic and logic functions, the ALU can flush the contents of the registers RX, RY, and RM to the ALU output and produce 20-bit constants frequently needed:

```
CLR . .  X'00000'   ONE . .  X'00100'
RDH . .  X'00080'   RDL . .  X'00008'
```

Constants RDH and RDL ("round high," "round low") are usually employed to initialize the accumulators such that sums of products are rounded to 12 or 16 msb when the 16 msb of RA or RB are stored in data memory. The special code SLC (C = A or B) left-shifts the contents of RA or RB by two bits. For conditional branching, the compute code establishes the ALU output to be tested. Hence, the function of conditional branching is accomplished during a single processor cycle.

Programming

Two assembler notations have been developed, one reflecting directly the architecture and instruction format described in the preceding section, and another matching more closely the assembler notation of conventional microprocessors without parallel operation codes. For the programming examples presented in this section, the first notation is used where assembler instructions take the form

```
[label]    tc      oper[+I|J]    [cc[,SA|SB]]    [* comment]

e.g.,

#LAB1      LMX     VAL+27+J      X+A,SA          * example.
```

Here, *tc* and *cc* are mnemonics taken from Tables 1 and 2, and *oper* is a mixed expression of symbolic and constant values. Indexation by RI or RJ is indicated by appending +I or +J; SA or SB denotes that the ALU output is to be stored in RA or RB. If the compute code is omitted from an assembler instruction, a default code without SA or SB is generated.

A program development system has been created which comprises assembler, simulator, and the necessary hardware and software tools for interactive loading and testing of DSP application programs under the control of an IBM PC or Series 1 computer.

- *Programming example 1*

Move V1 to U1, and compute U2 = V1+V2 and U3 = V1−V2:

```
LMX  V1          * Load V1 into RX
LMY  V2          * Load V2 into RY
SXM  U1   X+Y,SA * Store RX into U1, X+Y into RA
SAM  U2   X−Y,SB * Store RA into U2, X−Y into RB
SBM  U3          * Store RB into U3
```

- *Programming example 2*

This example demonstrates the ZIP technique in its simplest form. Two consecutive output samples T_{n-1} and T_n of a transversal filter with real-valued signals S_n, S_{n-1}, S_{n-2}, $\cdots$ and coefficients C_0, C_1, $\cdots$, C_{N-1} are computed:

$$T_{n-1} = S_{n-1} * C_0 + S_{n-2} * C_1 + S_{n-3} * C_2 + \cdots S_{n-N} * C_{N-1},$$

$$T_n = S_n * C_0 + S_{n-1} * C_1 + S_{n-2} * C_2 + \cdots S_{n-N+1} * C_{N-1}.$$

If values S_n, C_0, S_{n-1}, C_1, S_{n-2}, C_2, $\cdots$ are loaded alternately into RX and RY, products belonging to T_{n-1} and T_n appear alternately at the multiplier output and can be accumulated in RB and RA. Thus, asymptotically for large filter length N, one product is obtained and accumulated in every processor cycle.

The following instruction sequence demonstrates this for a filter of length $N = 19$. Index register RI is set and signal values are stored such that S+I = S_n, S+1+I = S_{n-1}, S+2+I = S_{n-2}, etc. Coefficients are provided in the form of direct operands:

```
LMX  S+I              *Load S+I into RX
LDY  C0               *Load C0 into RY (direct operand)
LMX  S+1+I  RDH,SA    *Set round constant into RA
LDY  C1     RDH,SB    *Set round constant into RB
LMX  S+2+I  M+A,SA    *Accumulate S+I   ×C0 in RA
LDY  C2     M+B,SB    *Accumulate S+1+I×C0 in RB
LMX  S+3+I  M+A,SA    *Accumulate S+1+I×C1 in RA
LDY  C3     M+B,SB    *Accumulate S+2+I×C1 in RB
...  ...    ......
...  ...    ......
LDY  C17    M+B,SB    *Accumulate S+16+I×C15 in RB
LMX  S+18+I M+A,SA    *Accumulate S+16+I×C16 in RA
LDY  C18    M+B,SB    *Accumulate S+17+I×C16 in RB
LMX  S+19+I M+A,SA    *Accumulate S+17+I×C17 in RA
---  ---    M+B,SB    *Accumulate S+18+I×C17 in RB
---  ---    M+A,SA    *Accumulate S+18+I×C18 in RA
SAM  T+I    M+B,SB    *Store RA into T+I
SBM  T+1+1            *Store RB into T+1+I
```

Two transfer cycles indicated by the symbols --- --- remain available for use by the next task. Coefficients could also be read from data memory by replacing LDY Ci with LMY Ci. Cyclic addressing schemes devised either in hardware or in software can be employed to avoid shifting of

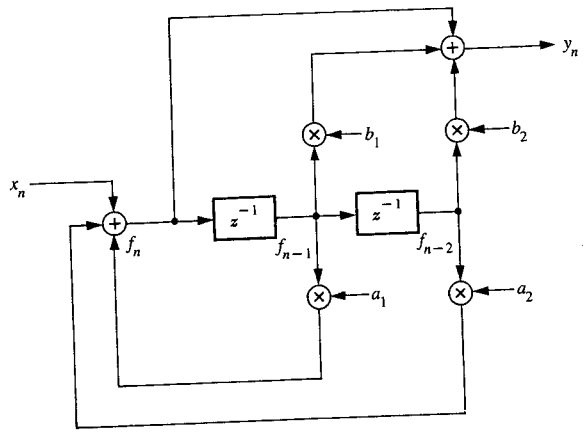


Figure 6

Second-order recursive-filter section.

signal values in data memory. Only eight instructions must be added to the above code to realize a particular software scheme which then performs the entire filter function at 1.23 cycles per multiplication.

• *Programming example 3*

The function of a second-order recursive-filter section depicted in **Figure 6** is performed. Input sample x_n and internal signals f_{n-1} and f_{n-2} are used to compute output sample y_n and new values for f_{n-1} and f_{n-2}:

```
LMY  FN_2
LDX  B2
LDX  A2    RDH,SA   *Set round constant into RA
LMX  FN_1  RDH,SB   *Set round constant into RB
LDY  B1    M+A,SA   *RA ← B2×FN−2
LDY  A1    M+B,SB   *RB ← A2×FN−2
SXM  FN_2           *FN−2 ← FN−1 (shift)
LMX  XN    M+A,SA   *RA ← B2×FN−2+B1×FN−1
---  ---   X+A,SA   *RA ← XN+B2×FN−2+B1×FN−1
SAM  YN    M+B,SB   *YN ← RA (output)
---  ---   X+B,SB   *RB ← XN+A2×FN−2+A1×FN−1
SBM  FN_1           *FN−1 ← RB (new)
```

Two transfer cycles remain available for use by the next task which may, e.g., perform another filter section. Thus, asymptotically each filter section requires ten processor cycles. Assuming a cycle time of 100 ns, 125 filter sections can be executed at an 8-kHz sampling rate.

• *Programming example 4*

This example illustrates the technique of an "instruction table." The variable CODE is translated from straight binary to Gray code. For CODE = 2 (12 msb), instructions are fetched and executed in the sequence (1), (2), (3), (4):

```
      LMI  CODE            *Load CODE into RI.
      ---  ---             *The 12 msb of CODE may
      BRD  #TAB+I          *contain values 0–15.     (1)
      BRD  #CON                                      (2)
*
#TAB  LDX  B'000000000000' *Gray code table
      LDX  B'000000000001'
      LDX  B'000000000011'                           (3)
      ...  ...
      ...  ...
      LDX  B'000000001110'
      LDX  B'000000001010'
      LDX  B'000000001011'
      LDX  B'000000001001'
      LDX  B'000000001000'
*
#CON  SXM  CODE            *Store CODE               (4)
```

• *Programming example 5*

This example illustrates the use of conditional branch instructions. The maximum value among VAL(1), VAL(2), ..., VAL(N) and the corresponding index are determined:

```
      LDYI  N              *Load N into RY and RI
      BRD   #L2            *Branch to instruction #L2
      LMX   VAL+I          *Load VAL(N) into RX
*
#L1   BRND  #L3   X−A      *Branch to #L3 if RX < RA
#L2   LDYI  −1+I  Y,SB     *Update RY/RI, save old RY/RI
      SBM   RJ    X,SA     *Store RB into RJ, RX into RA
#L3   BRAD  #L1   Y        *Branch to #L1 while RY > 0
      LMX   VAL+I          *Load next value
```

The maximum value is now in RA and the corresponding index in RJ.

Conclusion

The architecture of a single-chip digital signal processor has been described; the DSP can operate efficiently with a single data-bus structure. In many cases, it achieves typical processing tasks with just the number of cycles needed to read data from the single-port data memory and store results. The novel ZIP technique allows transversal-filter operations to be performed asymptotically with one multiplication in every cycle. All major components of the processor are then in concurrent use. The architecture was purposely kept very simple to allow a particular high-speed implementation with state-of-the-art technology available when the processor was designed.

References

1. J. G. Posa, "Signal Processors Spur Changes," *Electronics* **53**, 100–101 (February 14, 1980).
2. T. Nishitani, R. Maruta, Y. Kawakami, and H. Goto, "A Single-Chip Digital Signal Processor for Telecommunications Applications," *IEEE J. Solid-State Circuits* **SC-16**, 372–376 (1981); (describes the NEC 7720).
3. E. R. Caudel and R. K. Hester, "A Chip Set for Audio Frequency Digital Signal Processing," *Proceedings, IEEE International Conference on Acoustics, Speech, and Signal Processing* (IEEE Catalog No. CH1746-7/82), 1982, pp. 1065–1068 (describes the TI TMS32010).
4. S. S. Magar, E. R. Caudel, and A. W. Leigh, "A Microcomputer with Digital Signal Processing Capability," *IEEE International Solid-State Circuits Conference, Digest of Technical Papers* **25**, 32–33 and 284–285 (1982); (describes the TI TMS32010).
5. M. Kikuchi, T. Inaba, Y. Kubono, H. Hambe, and T. Ikesawa, "A 23 K Gate CMOS DSP with 100 ns Multiplication," *IEEE International Solid-State Circuits Conference, Digest of Technical Papers* **26**, 128–129 (1983); (describes the Fujitsu MB8764).
6. R. E. Owen, "VLSI Architectures for Digital Signal Processing," *VLSI Design* **5**, 20–28 (1984).
7. W. E. Nicholson, R. W. Blasco, and K. R. Reddy, "The S2811 Signal Processing Peripheral," *Proceedings of WESCON* **25/3**, 1–12 (1978); (describes the AMI S2811).
8. M. Townsend, M. E. Hoff, Jr., and R. E. Holm, "An NMOS Microprocessor for Analog Signal Processing," *IEEE J. Solid-State Circuits* **SC-15**, 33–38 (1980); (describes the Intel 2920).
9. J. R. Boddie, G. T. Daryanani, I. I. Eldumiati, R. N. Gadenz, J. S. Thompson, and S. M. Walters, "Digital Signal Processor: Architecture and Performance," *Bell Syst. Tech. J.* **60**, 1449–1462 (1981); (describes the Bell DSP).
10. Fred Mintzer and Abraham Peled, "A Microprocessor for Signal Processing, the RSP," *IBM J. Res. Develop.* **26**, 413–423 (1982); (describes the IBM RSP).
11. G. Ungerboeck, D. Maiwald, H. P. Kaeser, and P. R. Chevillat, "The SP16 Signal Processor," *Proceedings, IEEE International Conference on Acoustics, Speech, and Signal Processing* (IEEE Catalog No. CH1945-5/84), 1984, pp. 16.2.1–16.2.4.

Received December 30, 1983; revised October 5, 1984

Jean Paul Beraud *IBM France, B.P. 58, 91102 Corbeil-Essonnes, Cedex, France.* Mr. Beraud is currently a logic-design engineer at the Essonnes laboratory. He joined IBM in La Gaude, France, in 1966 and worked with modem and digital audio groups in the Advanced Technology Department. Since 1979 he has worked in the field of signal processing. In 1982, he joined the Component Development Laboratory in Essonnes to work on chip design. He received his Engineer Diploma from the Conservatoire National des Arts et Métiers de Nice, France, in 1971. Mr. Beraud received an IBM Outstanding Innovation Award in 1982 for the development of a real-time signal-processor chip.

Pierre R. Chevillat *IBM Research Division, Säumerstrasse 4, 8803 Rüschlikon, Switzerland.* Dr. Chevillat has been with the IBM Zurich Research laboratory since 1976. He has contributed to various projects in data transmission and signal processing, specifically the conception and prototyping of high-speed data modems for leased telephone lines and the development of IBM's advanced signal processors. He holds several patents in the areas of data communication and signal-processor architecture, and has received an IBM Outstanding Contribution Award for his work in data transmission. Dr. Chevillat received the Dipl. El-Ing. ETH degree from the Swiss Federal Institute of Technology, Zurich, Switzerland, in 1972, and the M.S.E.E. and Ph.D. degrees from the Illinois Institute of Technology in 1973 and 1976, respectively. At the Illinois Institute of Technology he was a National Science Foundation research assistant and the recipient of a Hasler Foundation Fellowship. In 1970, he was a member of the development staff at ITT Creed Ltd., England. Dr. Chevillat is a member of the Institute of Electrical and Electronics Engineers and Sigma Xi.

Hans-Peter Kaeser *Deceased.* Mr. Kaeser was a Research staff member at the IBM Zurich Research laboratory in Rüschlikon, Switzerland. He received the B.S. degree in electrical engineering from the Engineering College in Biel, Switzerland, in 1964, and joined IBM in 1966, after two years with Brown Boveri. He worked on projects dealing with speech processing, in-house communication, and data transmission. He joined Rüschlikon's signal-processing group in 1978. Mr. Kaeser designed and implemented prototypes of three generations of signal processors and participated in the realization of high-speed data modems using these processors. He received a First-Level Invention Achievement Award, a Research Division Award, and an IBM Outstanding Contribution Award for his work on signal processors. Mr. Kaeser died in June 1984.

Dietrich Maiwald *IBM Research Division, Säumerstrasse 4, 8803 Rüschlikon, Switzerland.* Dr. Maiwald is a Research staff member in the signal-processing group at the Zurich laboratory. In 1966, he received the Dr.-Ing. degree in communications engineering from the Technical University in Stuttgart, Germany. From 1963 to 1967 he was employed as a scientific assistant at the Institute for Communications Engineering in Stuttgart. In 1967, he joined the IBM Zurich Research laboratory, where his research work has included digital processing of speech signals for bandwidth reduction and transmission. In 1970, he was assigned temporarily to the technical planning staff of the Research Division in Yorktown Heights, New York. Since 1971 he has contributed to various research projects in the areas of digital switching, data transmission, and magnetic recording. Since 1979 he has participated in the design and benchmarking of digital signal processors and established the program development systems for these processors. He is currently applying programmable signal processors to realize advanced signal-processing concepts in high-speed voiceband data modems. Dr. Maiwald has received two Research Division Awards, a First-Level Invention Achievement Award, and an IBM Outstanding Contribution Award for his work in signal-processor architecture. He is a member of the Institute of Electrical and Electronics Engineers and the German Society for Communications Engineering (NTG).

Gottfried Ungerboeck *IBM Research Division, Säumerstrasse 4, 8803 Rüschlikon, Switzerland.* Dr. Ungerboeck is an IBM Fellow at the IBM Zurich Research laboratory in Rüschlikon, Switzerland. He received the Dipl. Ing. degree in telecommunications from the Technical University in Vienna, Austria, in 1964, and the Ph.D. degree from the Swiss Federal Institute of Technology in Zurich, Switzerland, in 1970. He first worked with the Wiener Schwachstromwerke in Vienna, and later with IBM Austria as a systems engineer. In 1967, he joined the IBM Zurich Research laboratory. He made original contributions to the theory of data transmission and the architecture of signal-processors. His invention of trellis-coded modulation is gaining wide application in digital transmission systems. As manager of the signal-processing group in Rüschlikon since 1978, he has pursued exploratory projects in architecting and prototyping IBM signal processors, developing advanced modems, and designing a new channel technique for digital magnetic recording. In 1973 he was on sabbatical leave at the IBM laboratory in La Gaude, France. Dr. Ungerboeck has received two IBM Outstanding Innovation Awards and two Invention Achievement Awards. For five years he served as an Associate Editor of the *IEEE Transactions on Communications*. He was recently elected a Fellow of the Institute of Electrical and Electronics Engineers.