SIGPL Summer School 2004

임베디드 프로세서 구조와 프로그래밍

서울대학교 전기컴퓨터공학부
소프트웨어 최적화 및 재구성 (SO&R) 연구실
백윤흥

2004년 8월 11일

Topics

- Basic ideas and notions of embedded applications
  - digital signal processing
  - (media processing, network processing)
- Embedded processors
  - off-the shelf DSPs, DSP core, ASIP
  - data path, registers, memory, instruction sets, pipelining, VLIW…
  - (media processors, network processors)
- Outstanding features of embedded code
  - numeric representations
  - ALU operations
  - data access patterns

Embedded S/W development issues

- Embedded systems are now hot…getting even hotter!!
  - telecommunications, multimedia, and more…
  - More and more vendors, even now Intel, produce processors.
  - GPPs like Pentiums and PowerPC are not effective for embedded applications like digital signal processing and network processing.
- S/W and F/W development with assembly for embedded processors is extremely difficult and too much costly.
- As embedded processors become more sophisticated, the amount of legacy code grows too large to maintain it effectively.
- High-level languages (esp. C) are good alternatives to assembly.
  - excellent portability, low cost for development, etc…

High-level languages for embedded S/W?

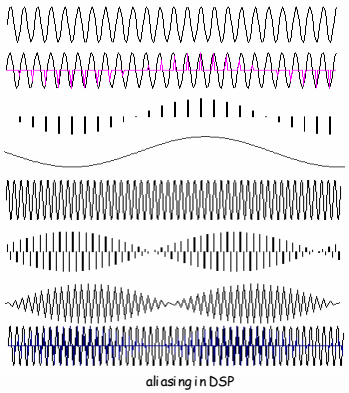
- No serious embedded programmer uses high-level languages.
- Why?
  - Terrible performance of machine code generated
- Who are responsible?
  - Embedded processors are not compiler-friendly.
  - Compilers are not smart enough to generate optimal machine code compatible with hand-written code in terms of performance.

Embedded processing ≈ Signal processing

- No interface to human being
- Processing elements in an embedded system communicate each other through signals.
- Most embedded systems are designed to process signals.

Signal processing in embedded systems

- Signal
- Mathematical representations
- Analog signal processing
- Digital signal processing
- DSP operations
- FIR filters
- IIR filters
- FFT
- Programming examples

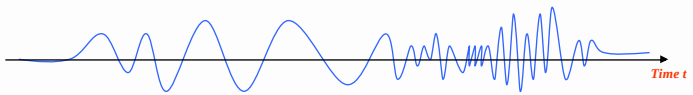


Signal?

- Definition in the American Heritage(r) Dictionary

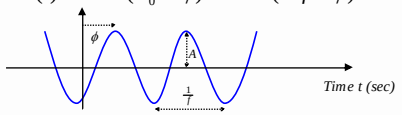
  An impulse or a fluctuating electric quantity, such as voltage, current, or electric field strength, whose variations represent coded information

- patterns of variations in time that represent or encode information
- carry information (e.g., audio, video) through an electronic circuit to be used in measuring or probing other physical systems
- The pattern of variations forms a time waveform.



Contiguous-time (analog) signals

- Mathematical representations of sinusoidal signals
  - cosine (or equivalently, sine) signals
  - the most basic signals in the theory of signal processing
  - a signal: $x(t) = A\cos(\omega_0 t + \phi) = A\cos(2\pi f t + \phi)$

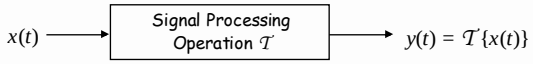


- Mathematical representations of complex exponential signals
  - the form: $\bar{x}(t) = Ae^{j(\omega_0 t+\phi)}$ ➔ *more convenient to analyze and handle signals mathematically*
  - Extraction of a sinusoidal signal after processing is done

$$\mathrm{Re}\{\bar{x}(t)\} = \mathrm{Re}\{Ae^{j(\omega_0 t+\phi)}\} = \mathrm{Re}\{A\cos(\omega_0 t+\phi) + jA\sin(\omega_0 t+\phi)\} = x(t)$$

Signal Processing…

- analyzes and modifies the information conveyed in signals
- speech synthesis/recognition, audio amplification, noise reduction, high-speed modems w/ error correction, …
- indispensable for embedded system
- Analog signal processing
  - Signals from the real world are analog.
  - Such natural signals can be processed directly using analog electronic devices such audio amplifiers (*w/ resistances, conductors,…*)
  - Generally too expensive or often even impossible to process signals using analog electronics.



Digital Signal Processing

- Signals can be processed by digital devices instead.
- Simplicity, cost-effectiveness
  - Tasks that would be difficult or even impossible can be accomplished at much lower cost.
  - The size of digital components is small & consistent unlike analog counterparts whose sizes vary with their values.
- Versatility
  - A digital device like a programmable DSP processor can perform other tasks by simply reprogramming it. (no physical changes)
- Predictability, repeatability
  - considerably less insensitive to environment like temperature and to component tolerances.
  - easily duplicated and ported to other H/W, while having exact known responses that do not vary.

Digital Signal for DSP

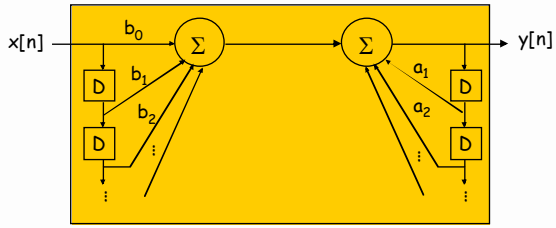
- A digitized form of an analog signal
  - a series of discrete numbers representing a sequence of the samples of the analog signal
  - generated by sampling the analog signal at intervals of T seconds.
  - held in memory and processed by a DSP processor.
- A digital signal: $x[n] = x(nT) = [x(0), x(T), x(2T), \ldots]$

Aliasing distortion *Sampling frequency/2 = nyquist frequency*

- Conversion bet'n analog and digital signals in a DSP system

DSP operation $\mathcal{T}$

- A typical DSP linear transfer function $\mathcal{T}$



$$y[n] = \sum_{q=0}^{Q-1} b_q x[n-q] + \sum_{p=1}^{P-1} a_p y[n-p]$$

- The time delay D implemented with a latch or register gives a delay of a unit sample period T.

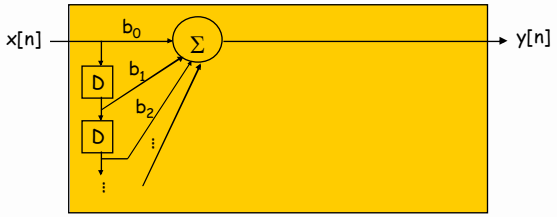
Common functions in DSP

- FIR filters
- IIR filters
- z-transform
- Fourier transform

FIR filters

- For FIR(finite impulse response) filters, each output signal y[n] is the sum of a finite number of weighted samples of the input signal sequence x[n].

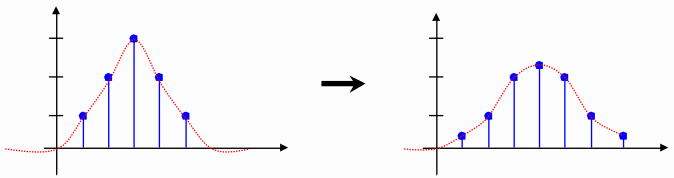
$$y[n]=\sum_{q=0}^{Q-1} b_q x[n-q]$$



The running average filter

- A FIR filter that computes a running (moving) average of several consecutive numbers of the input sequence, and produce a new sequence of the average values.

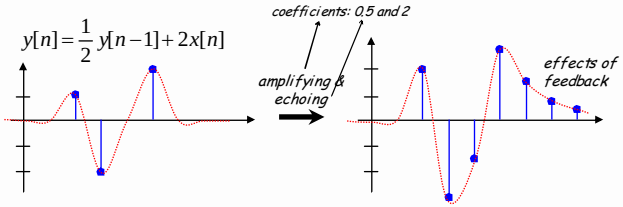
ex) a difference equation for 3-point averaging method $y[n]=\sum_{q=0}^{2}\frac{x[n-q]}{3}=\frac{1}{3}(x[n]+x[n-1]+x[n-2])$



Make the output signal smoother than the input signal

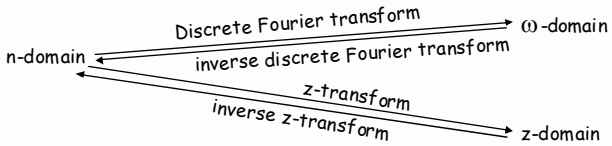
IIR filters

- A difference equation: $y[n]=\sum_{q=0}^{Q-1} b_q x[n-q]+\sum_{p=1}^{P-1} a_p y[n-p]$
- IIR filters involves previously computed values of the output signal as well as values of the 'recent' input signal in the computation of the present output.
- important to model 'resonance' such as would occur in a speech synthesizer.
- Ex: $y[n]=\frac{1}{2}y[n-1]+2x[n]$



Domains of signal representation

- A difficult analysis and process in the time domain is often easier in different domains.
- three domains
  - time domain (n-domain)
  - frequency domain (ω_0-domain)
  - z-domain
- Transformation between domains



Fourier transform

- The frequency domain is useful to analyze sound.
- the Discrete Fourier Transform of a signal x[n]:

$$X(\omega_0) = \sum_{n=0}^{N-1} x[n] e^{-j\omega_0 n} \quad \text{or} \quad X(k) = \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N} \qquad O(n^2)$$

- FFT (Fast FT)

$$X(k) = \sum_{n=0}^{N/2-1} x_{ev}[n] W_{N/2}^{nk} + W_{N/2}^{k} \sum_{n=0}^{N/2-1} x_{od}[n] W_{N/2}^{nk} \qquad O(n\log_2 n)$$

where $W^{nk} = e^{-j2\pi kn/N}$

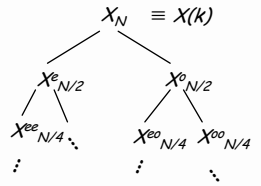
Examples of DSP programming

- FFT, IFFT
  - An extremely important algorithm in DSP that is used to convert DSP problems between the time and the frequency domains.
  - FFT (time complexity = O(NlogN)) is much faster than DFT (= $O(N^2)$) in that FFT reuses the existing results computed for a node in the computations for other nodes.
- Audio equalization
  - A music processing technique that filters a music signal such that the frequency contents of the signal is adjusted to improve the input sound as the audience desire, or remove noise that may be in a frequency band different from the desired signal.
  - 16 ~ 24 bits to represent each sample with 40 ~ 50 kHz sampling rate

FFT

- taking N signal samples (x[0],…,x[N-1]) in the time domain to produce N samples (X[0],…,X[N-1]) in the frequency domain
- A divide-and-conquer method

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi nk}{N}} = \sum_{n=0}^{N-1} x[n] W^{nk} \\ &= \sum_{n=0}^{N/2-1} x[2n] W_N^{2nk} + \sum_{n=0}^{N/2-1} x[2n+1] W_N^{(2n+1)k} \\ &= \sum_{n=0}^{N/2-1} x^{ev}[n] W_{N/2}^{nk} + W_{N/2}^{k} \sum_{n=0}^{N/2-1} x^{od}[n] W_{N/2}^{nk} \\ &= X_{N/2}^{ev}(k) + W_{N/2}^{k} X_{N/2}^{od}(k) \end{aligned}$$

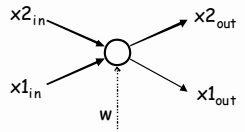


The Cooley-Tukey FFT algorithm

- Iterative, in-place FFT
- the most efficient, thus common, FFT algorithm used in practice
- Each butterfly is visited only once
  - → no redundant computations
- An iterative method is used
  - → no extra time for procedure calls or space for subarrays for the input
- The storage is performed in-place
  - → no extra storage for the output

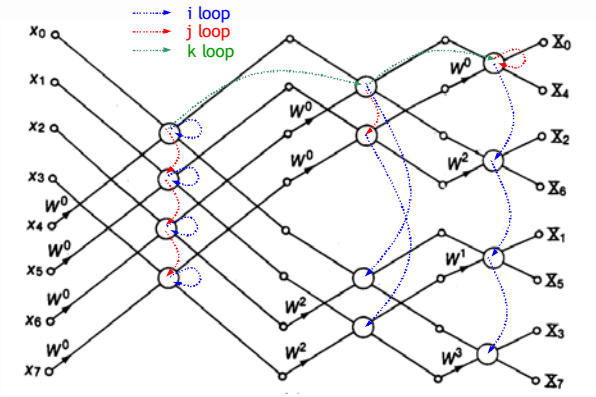
The Cooley-Tukey FFT algorithm

```
fft(int n, float* x) {
  … /* declarations */
  …
  for (k = 0; n > 0; k++, n/=2) {
    …
    for (j = 0; j < n; j++) {
      for (i = j; i < n; i += 2*n) {
        x1 = x + i;    /* lower input leg of a butterfly */
        x2 = x1 + n;   /* upper input leg of a butterfly */
        tmp = *x1 + *x2;
        *x2 = w * (x1 – x2); /* upper output leg */
        *x1 = tmp;           /* lower output leg */
      }
      …
    }
    …
  }
}
```



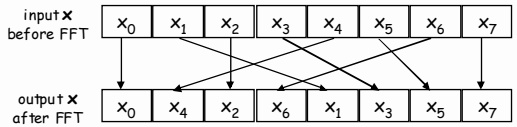
At every iteration k, the contents of the array x are completely updated.

A trajectory of computations in FFT



Bit-reversing

- A problem with the in-place FFT algorithm?
  - → The order of the values stored in the output array is scrambled.



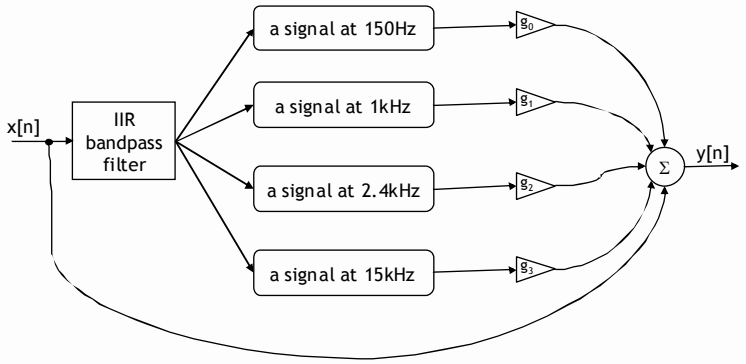
- How to find index *j* of the array **x** such that $x_j = FFT(x_i)$?
  - → Luckily, there is a simple function that maps bet'n input & output. That is, j is a bit-reversed form of i (if $i = b_1b_2...b_n$ in a binary notation, then $j = b_n...b_2b_1$).

input print

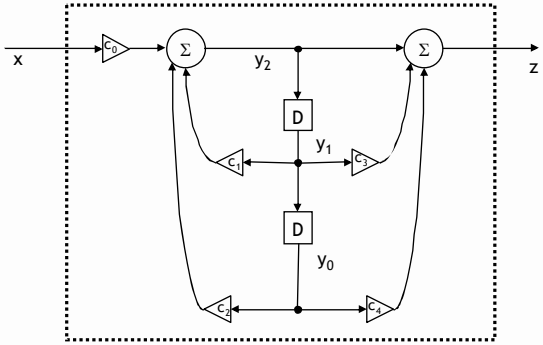
```
float *x;
…
for (i=0; i<n; i++)
  printf("%f%",x[i]);
fft(n,x);
for (i=0; i<n; i++) {
  j=bit_reverse(i);
  printf("%f%",x[j]);
}
```

output print

A 4-band equalizer



IIR bandpass filter



Code for the bandpass filter

```
float iir_bpf(float x, float* c, float* yold) {
  float z;
  float y2 = x * c[0];
  float y1 = yold[1];
  float y0 = yold[0];
  y2 = y2 - y1 * c[1];
  y2 = y2 - y0 * c[2];
  yold[0] = y1;
  yold[1] = y2;
  y2 = y2 + y1 * c[3];
  z = y2 + y0 * c[4];
  return(z);
}
```

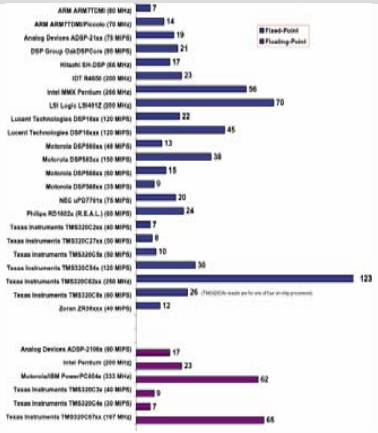

Code for the 4-band equalizer

```
void equalizer() {
  int i;
  float y, x;
  static float coeff[4][5] = {
        /* initialize the coefficients */
  };
  static float yold[4][2] = {
        /* initialize two recent outputs */
  };

  for (;;) {
    y = x = read_input();
    for (i = 0; i < 4; i++)
      y += g[i] * iir_bpf(x,coeff[i],yold[i]);
    print_output(y);
  }
}
```

Embedded processor for signal processing

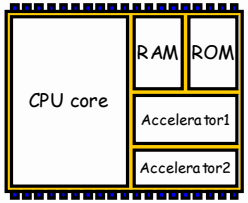
- Overview of embedded systems
- General-purpose vs. DSP processors
- Instruction sets
- Data path
- Memory



performance comparison of commercial microprocessors

DSP systems

- DSP algorithm
  - a series of mathematical operations that are applied to process a sequence of digital signals sampled from the real world, and to respond to the input appropriately.
  - filtering, FFT, noise cancellation, spectral processing, multirate signal processing, audio/speech encoding/decoding, …
- Embedded (real time?) DSP system
  - the embedded system (?) in which DSP algorithms are performed.
  - can be implemented by desktop computers
  - modems, printers, digital A/Vs (cd/dvd/mp3 players), cellular phones, cars, radar systems, flight navigation systems, house appliances (microwave ovens, refrigerators, TVs…)



When DSP systems are implemented

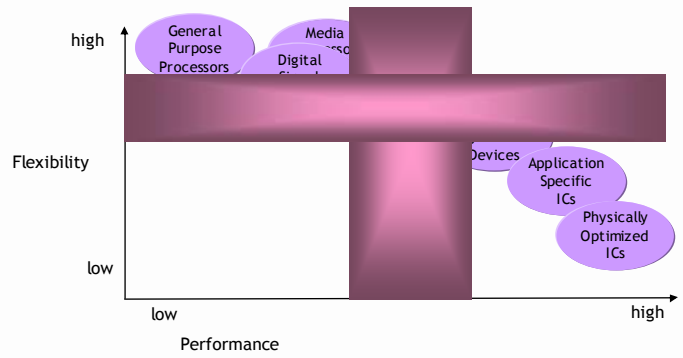
… they are tailored to run DSP algorithms efficiently by meeting the demands from the algorithms:

- Unique data access patterns
  - streams of bulky data that require high data bandwidth
  - low locality of data reference, but some program locality
- Heavy number crunching
- Real-time constraints
- Constraints on power consumption and size
- Often strict cost requirements
- Attention to subtle numeric effects (in fixed-point implementations)
- Specialized peripherals or I/O interfaces

Implementing an embedded DSP system

- Fixed-function solutions
  - custom integrated circuits or FPGAs
  - the smallest size and fastest running
  - prohibitively high initial development costs for each system
  - cost-effective for high-volume products
- Programmable microprocessors
  - off-the-shelf DSP processors or general-purpose processors
  - easy changes, upgrades or fixes of product functionalities
  - cost-effective and less risky for low-volume products
  - GPPs are usually costlier and less energy efficient (also often slower) than DSP processors, which are optimized specifically for DSP algorithms.

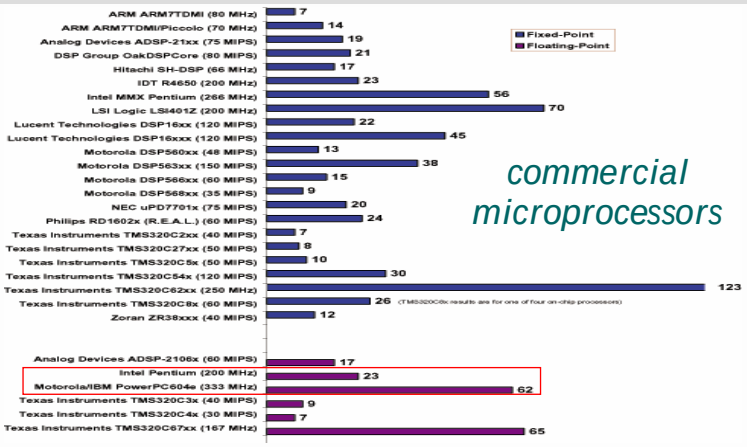
Implementing an embedded DSP system



Are GPPs a solution for DSP?

- Maybe, yes…(becu'z GP means …'can do anything'…)
  - Diverse, versatile SW development environment of desktops
  - Ok for some PC-based applications (telephony, videoconferencing, music play, divx movie play…)
- Maybe not…
  - Few H/W features implemented having DSP in mind
  - High cost/performance ratio (2 ~ 5 times faster but 10 ~ 20 times more expensive than mid-range fixed-point DSP processors)
  - Often too big to fit into some embedded systems
  - Difficult to predict execution time
    - complex hardware (data caches, dynamic branch prediction, and instruction scheduling, …)
    - lack of powerful profiling tools
    - a drawback in DSP applications with real time constraints

BDTi benchmark scores



Signal processing with GPPs

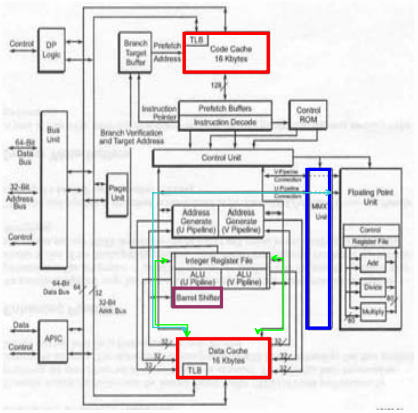
- GPPs target desktop/mobile computing and embedded systems like automotive control and communications.
- GPPs lack hardware features intended for DSP. But…
- Many GPPs achieve good execution time performance on (esp., floating-point) DSP applications through …
  - much higher clock frequency (usually above 1 GHz)
  - a few special instructions (like **mac**) or addressing modes that are extended from the original GPPs (e.g., Pentium MMX)
  - sophisticated memory hierarchy including data caches, and
  - other excellent H/W supports (deep pipelining, advanced branch prediction, ILP exploitation with multi-issue architecture)
- Recently, more and more GPPs include DSP features
  (e.g., Anti-Vec for PowerPC G4 and MMX, SSE for Pentium)
- Two examples of GPPs: PowerPC 604, Intel MMX

Pentium MMX

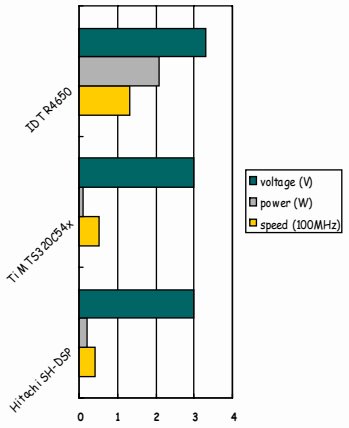
- 32-bit, two-issue superscalar CISC processor formally introduced in 1997 (at an initial price of $550 in quantity 1000)
- Generally CISC processors have less fancy H/W features for DSP than RISC counter parts (e.g., lower clock rates, less deeper pipelines, a less number of issues for superscalar, …).
  - → Intel overcomes the limitations in the traditional CISC way (*that is, by introducing more H/W instructions!)*
- Intel added to her the original 80x86 architecture …
  - 57 new MMX instructions intended for DSP ISA, and a *SIMD*-style MMX data path for fast vector operations, which are common in DSP (multimedia, communications, …) applications.
  - → A similar approach for PowerPC G4: *Anti-Vec* (graphic instructions in a dedicated pipeline running at speeds of 0.5~1GHz)
- Embedded Pentium MMX outperforms most DSP processors.

The Pentium MMX data path

- Two 64-bit ALU(integer) pipelines, a floating-point pipeline and an MMX pipeline
- A barrel shifter (useful to many DSP applications)
- Two 16 KB on-chip caches for instructions & data each with TLBs
- The data cache allows two loads/stores of integers or a single load/store of floating-point operand in a cycle



Energy efficiency



- Energy efficiency is important for many DSP systems.
- DSPs or other embedded processors are designed with energy efficiency taken into account.
- In general, their power consumption rates are much lower than GPPs.

What constitute a DSP processor?

- Strictly speaking, …
  → microprocessor that can operate on digitally represented signals
- In practice, however, …
  → microprocessor that is specifically designed to perform DSP
- Special features to characterize a DSP processor
  - multiple-access memory architectures → e.g. Harvard architecture
  - special circuitry to rapidly perform repetitive, numerically intensive calculations → e.g., MAC, vector processing units, …
  - special memory addressing modes → e.g., bit reversed mode, …
  - no data cache as a low cost solution for DSP w/ low data locality
  - special program control features → e.g, zero-overhead loop, …
  - specialized on-chip peripherals or I/O interfaces w/ other system components like A/D converters or host computers

Important factors in DSP

- Several important factors that affect the selection of a processor for the DSP system
  1. Performance (execution time)
  2. Cost
  3. Energy efficiency
  4. Real-time suitability
  5. DSP application development time and cost
- The order of importance of each factor varies depending on the requirements from the system.
  → To meet a great variety of the requirements from DSP systems, the industry introduced numerous types of processors with minor variations *(ex:* Lucent DSP16xx*)*
- DSP processors can be classified largely into two classes.
  → fixed-point and floating-point processors

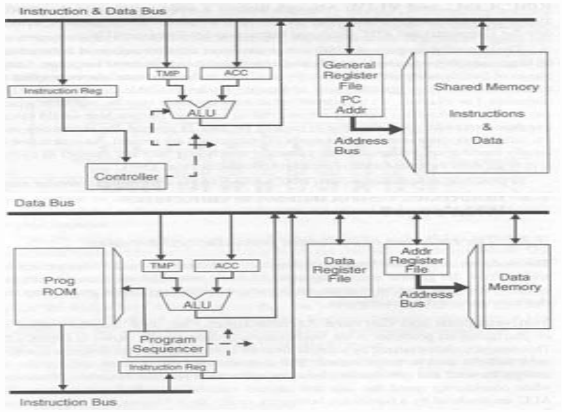
Architectural features of DSP processors

- Architectural features common to virtually all DSP processors
  - Harvard architecture coupled with some RISC properties
  - Address generators
  - On-chip addressable memory
- Features unique to most fixed-point processors
  - no data caches (some have instruction caches)
  - a rich variety of word-lengths (typically 8, 16, 20 and 24 bits)
  - hardware support for floating-point operations, called *block floating-point operations*

Two memory architecture designs

The Princeton architecture design

The Harvard architecture design



Impact of Harvard architecture on ALU ops

- The operations requiring many operands from memory (e.g., MAC) benefit from the Harvard architecture.
- Some GPPs (PowerPC, PA-RISC & MIPS) support limited MAC operations based on the Princeton architecture.
  - Their sustained throughput is two cycles per MAC at maximum.
  - DSP processors can achieve a sustained throughput of one cycle per MAC.

Ex) PowerPC code

```
    MTSPR CTR,R0
L:  LFD F1,0(R2)
    LFD F2,4(R2)
    FMADD F5,F5,F1,F2
    ADDIC R2,R2,4
    BDNZ CTR
```

3 cycles per MAC

Lucent StarCore code

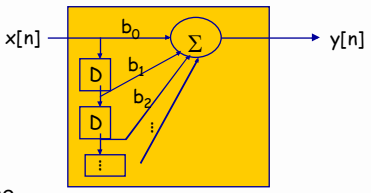
```
    …
L:  mac d0,d1,d2 move.f (r0)+,d0 move.f (r1)+,d1
    …
```

1 cycle per MAC

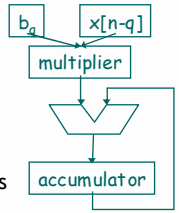
FIR filters on the Harvard design

- The memory operations during each cycle of the FIR filtering
  - a MAC instruction fetch
  - a read for b_q
  - a read for x[n-q]
  - a write to shift x[n-q] along the delay line

  (Can this write be saved if a circular buffer is used?)



$$y[n] = \sum_{q=0}^{Q-1} b_q x[n-q]$$



- Three memory accesses are needed per tap.
  - three cycles per tap on Princeton
  - 1.5 cycles per tap on Harvard w/ two memory banks (1 cycle per tap on Harvard w/ three banks)

Common instruction sets for DSP processor

- DSP algorithms mold the ISA of a DSP processor.
- all the memory addressing modes typically supported by GPPs + special modes useful to important DSP algorithms
- parallel moves (multiple loads/stores in a cycle) for high bandwidth data access
- special ALU operations for DSP
- H/W looping constructs for loop-based DSP algorithms.
- other special instructions (like vectors)
- many high-level language features in the assembly code

Memory addressing

- Common to all types of modern microprocessors
  - register (direct)
  - immediate
  - register deferred (or indirect)
  - displacement
  - indexed
  - absolute (or memory direct)
  - memory indirect
  - auto-increment/decrement
- Additionally common to DSP processors
  - short immediate
  - short memory direct
  - implied
  - bit-reversed
  - circular

Implied and bit-reversed addressing

- Implied addressing
  - Special registers (multipliers, accumulators, …) dedicated to functional units are implicitly addressed by the instruction.
  - e.g., add P (A ← P + A) A: accumulator

    mpy (P ← X * Y) P: product, X/Y: multiplier registers
- Bit-reversed addressing
  - specialized for some DSP processors that are designed to efficiently run the FFT algorithm.
  - The output of the address (or index) register is bit-reversed and applied to the memory address bus.
  - e.g., **`BITREV(I,n)`**: **`I`** ← **`I`** + **`n`** *where* **`I`** *is an index register and* **`n`** *is a 32-bit number*

    an ADSP210XX instruction

Circular (modulo) addressing

- A circular data buffer
  - an important data structure (= a set of memory locations + an index pointer that steps thru them) to many DSP algorithms that handle continuous, long data streams like FIR/IIR filters
  - basic operations to manage the buffer
    1. updates the index pointer by adding the value,
    2. check if the pointer falls outside the buffer
    3. if yes, then it is adjusted back to the start of the buffer
- The circular addressing helps the user to manage a circular buffer efficiently and fast in hardware.

  ex) Analog Devices

  initialization L ← the buffer length; I, B← the base address

  buffer operation I ← B + (I + M - B) % L (*modulo address arithmetic*; *step size*)

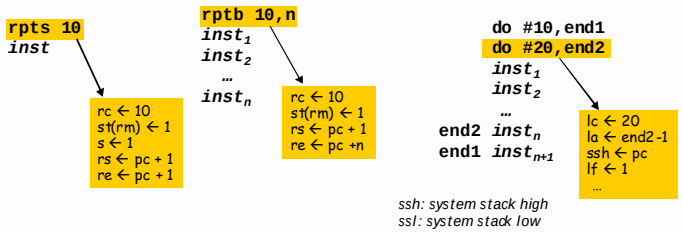
ALU operations

- Most ALU instructions in DSP processors are commonly found in other types of microprocessors except a few exceptions.
- Specialized arithmetic operations
  - dual add/subtract, MAC
  - square: **`SQR R0,R1`** (= **`MULT R0,R0,R1`**)
  - vector: **`ADDV V2,V1,V0`**
- Shift operations
  - Most processors provide instruction to shift a word by 1 ~ 2 bits.
  - Many DSP algorithms requires shifting by any number of bits.
  - DSP processors often features a *barrel shifter* for this.

H/W control flow instructions

- Multiply-nested small loops are quite common in DSP algorithms. → The loop overhead is significant!
- Hardware loops, called zero-overhead loops, are provided by virtually all DSP processors.

  ex) TMS320C3x/4x Motorola DSP5600x



Memory operations

- The majority of DSP processors support *parallel moves* (multiple memory accesses per instruction cycle) in H/W.
- Thus, they provide instructions that read/write multiple memory locations.
  - no load-store architecture style

    ex) MPY (R0),(R4) : P ← (R0) * (R4)

    "*implied + register indirect addressing*"
  - load-store architecture style

    ex) MPY X0,Y0 LD (R0),X0 LD (R4),Y0
- Some processors allow multiple instructions to access memory simultaneously.

  ex) MOV (R0),X0 MOV (R4),Y0 : (R0)→X0, (R4)→Y0

  "*parallel moves*"

Address generation units

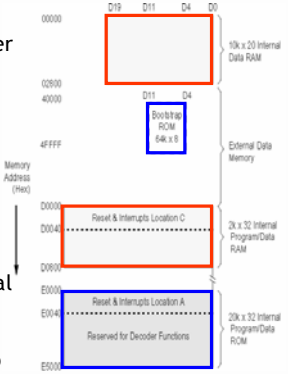
- dedicated to calculate data addresses
  c.f.) a program sequencer for instruction address calculation
- Why address generation units?
  - The Harvard architecture provides parallel moves.
  - Memory throughput is compatible with ALU speed
  - The AGU relieves the burden of address calculation from the ALU
- Address registers
  - attached to the AGU (typically 2 ~ 20 registers)
  - store addresses used for fast register-indirect / indexed(with index registers) / circular(with modulo registers) addressing
  - often also used as data registers for the ALU (e.g., Zoran)

On-chip memory units

- RAM
  Two RAMs are common. One is exclusively for data. To allow multiple data accesses per cycle while saving cost, the other is often shared by program and data
- ROM
  mainly for bootstrapping code (loading operational code & communicating with the host) or kernel code for DSP
- Cache
  To utilize the locality of program on external memory, small (16~32 words) instruction caches are commonly provided. Virtually all fixed-point DSPs have no data caches due to complex coherence H/W.

Typical size for GPPs is 256 or larger…

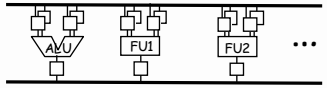
code = 32 bit, data = 20 bit



Memory map of ZR38601

Instruction encoding

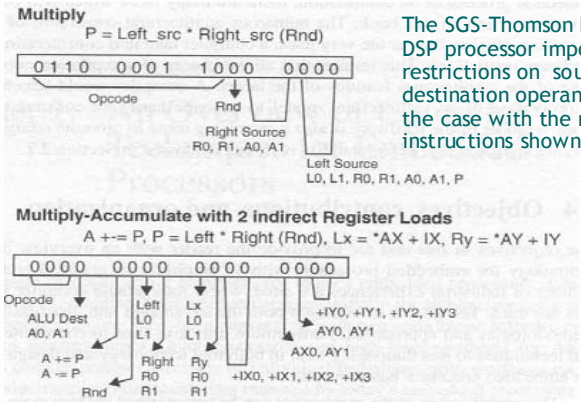
- DSP processors employ radical, irregular architectures specialized for each DSP application domain.
  → Irregular data path w/ small, heterogeneous, distributed registers



  → Non-orthogonal ISAs are common in DSP processor design.
- They suffer tight encoding constraints.
  - severe restrictions on code size
  - maximum parallelism & pipelining exploitation for performance
  - complex application-specific instructions for performance
  - What do these imply?
    → RISC-style fixed address form + CISC-style R-M/R+M ISA
      → opcodes of different widths, variable numbers of operands

Non-orthogonal instruction encoding

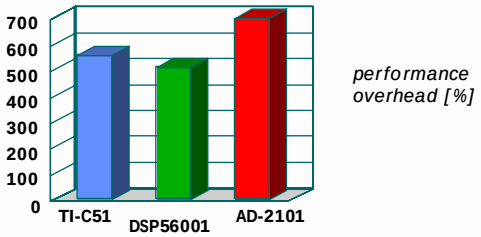
typically found in DSP processors



The SGS-Thomson D950 DSP processor imposes restrictions on source or destination operands as is the case with the multiply instructions shown here.

Code quality of C compilers on DSPs

- DSPStone benchmarking [ISS]
- Due to such complexities in DSP processors, compiled code f or DSPs often shows very high overhead (speed, code size, memory consumption) versus hand-written assembly code



- Manual postpass optimization of hot spots still required

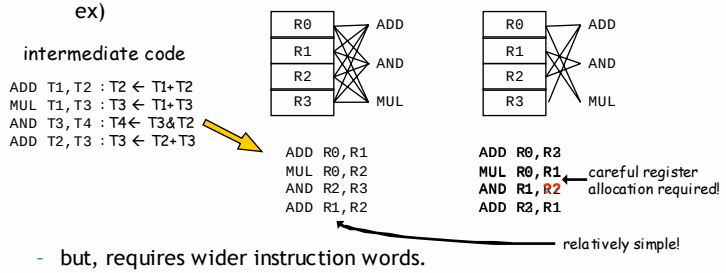
What the compiler prefers for the features of ISAs are …

- Low-level expressiveness of instructions …
  - gives more room for code optimizations.
  - Recall the examples of code optimizations for the load-store ISA and the register+memory ISA.
- Easy predictability of performance …
  - is possible by simple, straightforward hardware, and
  - allows the compiler to have simple trade-offs among alternative instructions, which helps improving code quality and meeting real time constraints.
- Orthogonality, regularity
  - Orthogonal ISAs support all addressing modes which apply to all instructions that transfer data.
  - Simpler optimization techniques will do with orthogonal ISAs

Impact of orthogonality of instruction set

- An orthogonal instruction set …
  - simplifies code generation,

ex)



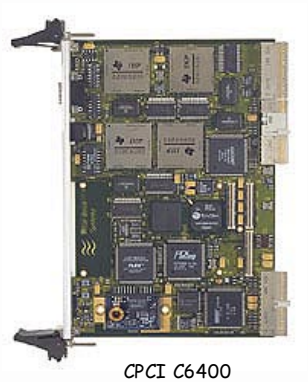
  - but, requires wider instruction words.
    → wider bus, memory widths → increased system costs

Soft-ways to enhance performance

- Traditional ways to improve the performance of applications were time-consuming and expensive.
  - *hard*-way: build faster, powerful *hard*ware.
  - manually optimize machine code
- As compiler technology has advanced, compilers provide cheap, efficient solutions to code optimization.
  - Why to choose hard-ways while there are soft-ways?
  - *soft*-way: little hardware enhancement, and mainly exploiting already-existing hardware features
  - little support from the programmers: saving time and providing more *performance-portability* of existing programs
- Simple, regular ISAs (like load-store ISAs) facilitate compiler optimizations.

Programming on embedded systems

- Components of a DSP program
- Data types
- Fixed vs. floating point data formats
- Operations
- Assembly languages
- Imperative languages
- Object-oriented languages
- Dataflow languages



CPCI C6400

Components of a DSP program

- Representations
  - data types to represent signal samples and other numbers
    - integers, real numbers, complex numbers
    - scalars, arrays (for sequences of data values)
  - numeric data formats for signal processing/computation
    - fixed-point
    - floating-point
- Operations to be performed on the data
  - Arithmetic
  - Logical and relational
  - Program control flow
  - Calls to mathematical library functions

ALU operations

- Some logical operations are often used to substitute for expensive arithmetic operations.
  - shift operations << and >> for multiplication/division operations
  - bit-wise logical operations |, & and ~ for error correction and decision processing (coupled with compare instructions) and bit manipulation (bit set/reset/toggle).
- C/C++ support a variety of low-level, bit-wise logical and relation operations which are directly implemented in a DSP processor.

Control flow statements

- Reactive control (choosing proper DSP algorithms responding to external input or interrupts) is an important functionality required by DSP applications.
- Programming constructs provided in high-level languages for reactive control
  - if-then-else, goto, do/for/while loops
  - Are these sufficient for DSP in terms of performance?
- DSP processors have many special instructions for efficient program control.
- Several extension to existing languages have been proposed (mostly by the vendors) to help programmers to exploit such instructions in their high-level programs.

Numeric C

- Extensions to ANSI C proposed by NCEG
- The **`Iter`** operator for looping

e.g., $y[n] = \sum_{j=0}^{m-1} b[n, j]x[n] + z[j]$ ➔

```
iter i=n, j=m
y[i] = 0.0
for (j) {
   y[i] = y[i]+b[i][j]*x[i]
   y[i] = y[i]+z[j]
}
```

- The **`sum`** operator
  - calculate the sum of values of the argument computed from values within a loop → very common in DSP algorithms
  - e.g. 1, …

```
for (j) {
   y[i] = sum(b[i][j]*x[i])
   y[i] = y[i]+z[j]
}
```

  - e.g. 2, multiplication of 10x20 and 20x30 matrices

```
iter i=10, j=30, k=20
c[i,j] = sum(a[i][k]*b[k][j])
```


Extensions of Numeric C for control flow

- Rationales for the extensions
  - Typical DSP programs contain multiply-nested loops. vector operators
  - A high-level description of a loop increases the chance for the compiler to translate the loop into more efficient looping instructions implemented in H/W
    - zero-overhead loops
    - SIMD functional units
    - hardware conditional instructions

The languages of choice for DSP

- Assembly
- Imperative languages (e.g., C)
- Object-oriented languages (e.g., C++, Ada95)
- Applicative languages rooted in dataflow principles (e.g., Silage, DFL, SDF, Id, Sisal, Lucid, Lustre)

Assembly languages

- guarantee to produce the most efficient code
  - THE…E… most important factor in DSP programming with strict performance and real-time constraints
  - So, up to now, have been the most popular in DSP programming
- directly support all data formats specified by H/W.
- expensive to maintain
- becoming more difficult to read and write as DSP algorithms and state-of-the-art DSP architectures get more complex and sophisticated
- legacy code problem
  - not portable to other architectures
  - hardly reusable or extensible on the change of design requirements

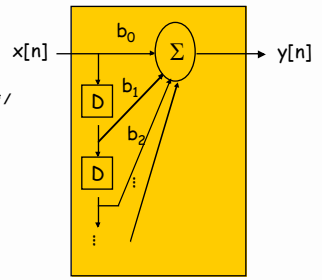
ANSI C

- imperative programming
  - *relatively* straightforward to generate target code for existing DSP processors, which are Von-Neumann or its variations based
- most widely recognized and used
  - flexible enough to describe any known DSP algorithms
  - having most nice features of high-level languages such as readability, portability and extensibility
  - yet, semantically close to assembly languages with a variety of low-level H/W operations
  - freely available compilers (e.g., GNU C and LCC)
- A great volume of legacy code has been standardized in C or C-like languages.
- limited data type support (e.g., complex numbers, word-length)

FIR filter in C

```
float x[Q],y;
float b[Q] = { 1.5, …
             /* initialization
               of coefficients */
               … };
main() {
  for (;;) {  /* 0 < n < ∞ */
     y = 0;   /* implicit time step n */
     x[Q-1] = receive_input();
     for (q = 0; q < Q; i++) {
        x[q] = x[q+1];
        y = y + x[q] * b[q];
     }
     send_output(y);
  }
}
```

$$y[n] = \sum_{q=0}^{Q-1} b_q x[n-q]$$



Data types in C

- C supports all numeric types needed in a DSP program except the type for *complex numbers*.
- Complex numbers are ubiquitous in DSP algorithms.
  → complex exponential signal $\overline{x}(t) = Ae^{j(\omega_0 t+\phi)}$
- How then to represent complex numbers in C?
  - No magic. Use the **`struct`** construct.

```
ex) typedef struct {
        float real;
        float imag;
    } complex;
```

  - Operations can be defined in the form of functions or macros.

```
#define cmult(x,y) (Z.real = x.real*y.real-x.imag*y.imag, \
                    Z.imag = x.real*y.imag+x.imag*y.real, \
                    Z)
```


Limitations of complex type in C

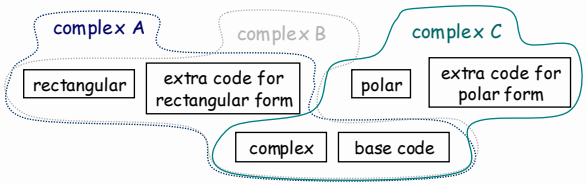
- Incomplete data abstraction
  - The complex type in C is not a true abstract data type. (object + operation)
  - No encapsulation of the details of representations for the complex type is guaranteed with **`struct`**s and macros.
  - No operations are incorporated in the definition of the data type
- Any change in the representation of complex may affect the definitions of its operations
- Awkward to use them
  → w = cmult(x,cmult(y,z)) ? … compiler error!

Numeric C for the **complex type**

- Numeric C extends ANSI C for complex
  - Built-in 6 integer and 3 floating-point complex types
    → **TYPE complex**
    where **TYPE** = short int, int, long int, float, double, long double
    ex: **complex int m = 3 + 5i; complex float n = 3.2 + .12i;**
  - All but logical and relational operations are defined
  - Coercion to complex from all other numeric types
  - Additional functions
    - **TYPE creal/cimag(complex)**
    - **complex conj(complex)**
- It is still C, offering all of the nice advantages of ANSI C for DSP, yet more efficient due to other extensions of control flow statements like the **iter** operator.
- An alternative solution? … object-oriented languages!

Object-oriented programming

- An object-oriented language like C++/Ada95 allows each object of the same type to have the autonomy to select representations suitable for its purpose.
- ex)



- Subtypes polar & rectangle are included in type complex. Any variables declared as complex can have access to both subtypes and operations. → *type inheritance*

Data abstraction in C++

- An abstract data type **complex** w/ type inheritance

```
class complex {
   public:
            virtual float real() {}
            virtual float imag() {}
            virtual float magni() {}
            virtual float angle() {}
            …
            virtual operator+ …
}
class rcomplex : public complex {
      float r,i;
   public:
      float real() { return r; }
          …
}
```

dynamic binding

```
class pcomplex : public complex {
      float m,a;
   public:
      float real() { return m*cos(a); }
          …
} …
complex *c1 = rcomplex(…);
complex *c2 = pcomplex(…);
complex *c3 = pcomplex(…);
 …
 … c1->angle() …
 … *c2 + *c3 …
 …
```

- Disadvantages to DSP applications?
  - extra pointers for type redirection (bad for embedded systems)
  - extra time for level of indirection to invoke code (maybe critical to real-time applications commonly found in DSP)

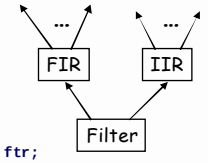
Implementations of filters in C++

taking a sequence of input samples produce a sequence of the output

a template for the data type (int,float,complex) of signal samples

```
template <class Type>
class Filter {
  protected:
    Type* b, x;
    …
  public:
    Filter () {…}
    virtual Type* filter(Type* xlist) {…}
    …
};
class FIR: public Filter<Type> {
  public:
    FIR (…) {…}
    virtual Type* filter(…) {…}
    …
};
class IIR: public Filter<Type> {
   …
};
```

derived types of FIR



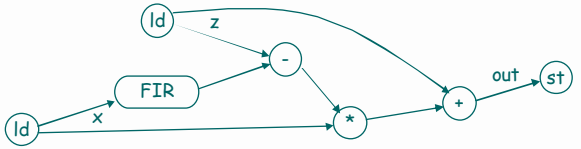

```
main() {
   Filter* ftr;
   Type* x, y;
   …
   switch (filtertype) {
     case FIR_TYPE: ftr = new FIR(…);
                    …
     case IIR_TYPE: ftr = new IIR(…);
                    …
   }
   y = ftr->filter(x) …
}
```

Dataflow approach for DSP programming

- of applicative nature
  - behavior specification of reactive systems (produce the output reacting to the, usually unbounded, input stream)
  - a natural paradigm to describe the reactive system like DSP
- a programming paradigm the least familiar (thus possibly difficult) to average programmers
- Computation can be described in a graphical form, called the *dataflow graph G = (N,E)*
  - E = a set of (semi-infinite streams of) values, called *tokens*
  - N = a set of *actors,* objects each of which receives input tokens (if any), performs operations on the tokens, and *fires* the token at the next actors waiting for it.

Computation in a dataflow model

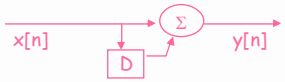
Ex) out = (z - FIR(x)) * x + z;



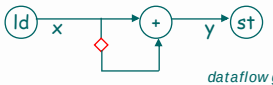
- A dataflow machine has a unique structure fundamentally different from traditional Von-Neumann machines.
- Typically, it consists of a network of actors that exchange tokens with each other and memory where tokens are temporarily stored.
- The computation automatically starts by providing the initial input tokens to the network.
- *Theoretically*, this type of machines is suitable to model parallel & distributed systems and reactive systems.

Dataflow languages for DSP

- provide textual notations to describe DSP algorithms in dataflow style
- a single assignment form → an arbitrary order of statements
- flexible data type support (dynamic type binding!)
- usually crafted for specific domains of DSP, so not as versatile as imperative languages (e.g., limited to describe program control flow)
- provide a construct for *time delay* required in DSP.



Flow graph structure for an example of DSP



dataflow graph for the example

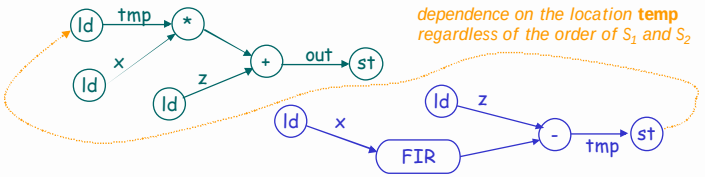
Silage

- A dataflow language specifically designed for signal processing
- Represents dataflow graphs in a textual form

ex) S_1: out = tmp * x + z;

S_2: tmp = z - FIR(x);

Two actors are connected thru the data dependence on the location **temp** *regardless of the order of S_1 and S_2*



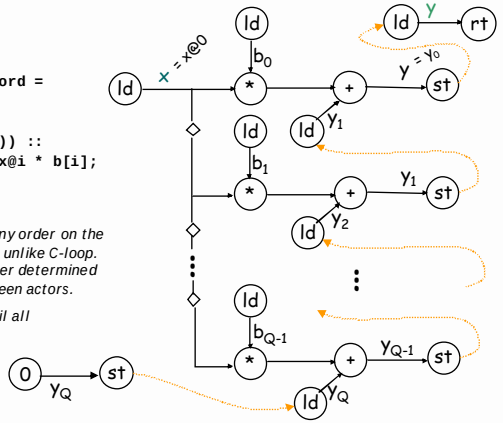
- The symbol @ is reserved to represent the time delay

FIR filter in Silage

```
#define word fix<48>
b = [word(1.5), … ];
func main(x: word): word =
begin
  y[upb(b)+1] = 0;
  (i: lwb(b) .. upb(b)) ::
     y[i] = y[i+1] + x@i * b[i];
  return = y[lwb(b)];
end;
```

The loop does not impose any order on the execution of each iteration unlike C-loop. The execution order is rather determined by data dependencies between actors.

The actors are waiting until all their input tokens arrive.



Conclusion

- Embedded systems are subject to strict performance and cost constraints.
- Embedded processors are often designed very irregularly to meet these constraints.
- Various programming language paradigms have been applied to programming embedded systems.
- Imperative programming languages dominate embedded system programming.
  - Strong C-affinity/loyality of EE engineers
  - Poor optimization
  - Limited versatility in non-imperative programming languages
- So, major efforts go into..
  - Leveling up assembly languages
  - Further extending C to other applications: SystemC
  - improving C compiler optimizations