Advanced Computer Architecture

—

Part II: Embedded Computing
Digital Signal Processors

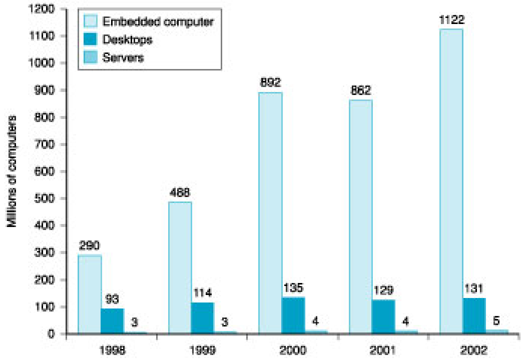
Paolo.Ienne@epfl.ch

EPFL – I&C – LAP

Embedded Systems

- ❑ Application-Specific:
  - ❖ Application fixed in advance
  - ❖ Not or very moderately programmable by the user
- ❑ Reactive:
  - ❖ Reacts on events coming from the environment
  - ❖ Has real time constraints
- ❑ Efficient:
  - ❖ Cost-reduction must profit from specialisation
  - ❖ Low power, small size,...

Embedded Processors

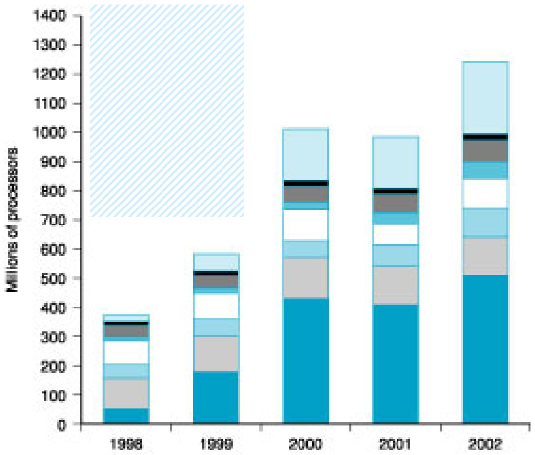


Source: Hennessy & Patterson, © MK 2005

- ❑ Until recently, embedded processors were almost always simple lowest-cost devices (8-bit microcontrollers, etc.)

But it is changing!...

Processor Sales Per Architecture



Source: Hennessy & Patterson, © MK 2005

High-end Embedded Processors

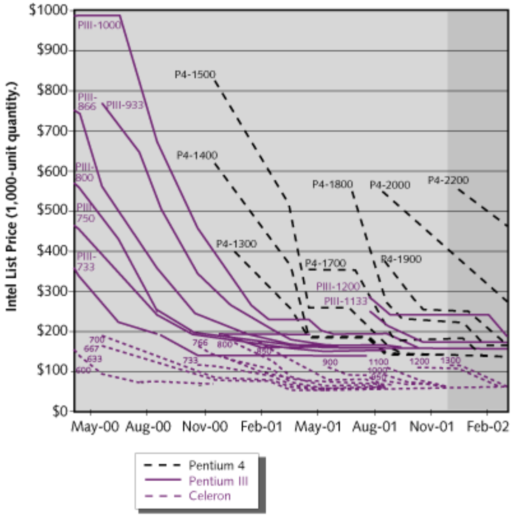
- ❑ Networking, wireless communication, printer and disk controllers, DVDs and video, digital photography, medical devices…
- ❑ Computing power ever growing
  - ❖ Sometimes products with 5-10 Digital Signal Processors on a single die (e.g., xDSL, VoIP)
  - ❖ Multimedia, cryptographic capabilities, adaptive signal processing, etc.

Specificities of Embedded Processors

- ❑ Cost used to be the only concern; now **performance/cost is at premium** and still not performance alone as in PCs (Intel model); performance is often a constraint
- ❑ **Binary compatibility** is less of an issue for embedded systems
- ❑ Systems-on-Chip make **processor volume irrelevant** (moderate motivation toward single processor for all products)

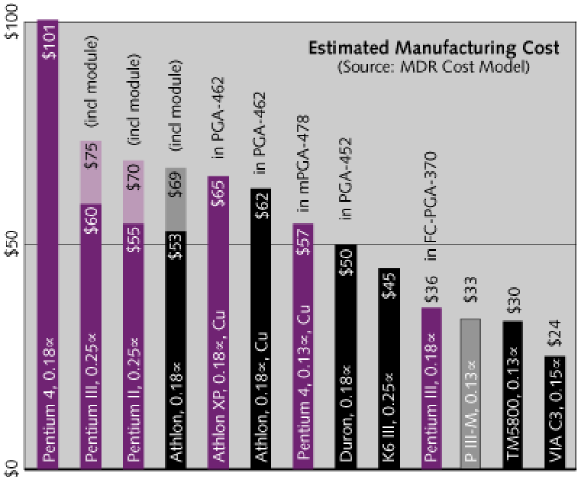
General Purpose Processors Cost and Pricing Policy

Intel List Prices vs. Time



Source: Microprocessor Report, © MPR 2002

General Purpose Processors Costs and Pricing Policy



Source: Microprocessor Report, © MPR 2002

Cost and Performance

- **Performance is a design constraint**
  - General Purpose: new high-end processor must be tangibly faster than best in class
  - Embedded: *adequately programmed*, must just satisfy some minimal performance constraints
- **Cost is the optimisation criteria**
  - General Purpose: must be within some typical range (e.g., 50-120 USD) → profit margin can be as high as some factor (2-3x)
  - Embedded: must be **minimal** → economic margin on the whole product can be as low as a few percent points

Cost Is Not Just the Processor...

- Processors have tangible induced costs
- Some could require:
  - Larger memories
  - More expensive memories (e.g., dual-port)
  - Caches (I and/or D)
  - Peripherals and accelerators
  - Faster clock rate
  - ...
- System cost can be extremely influenced by the architecture of the processor

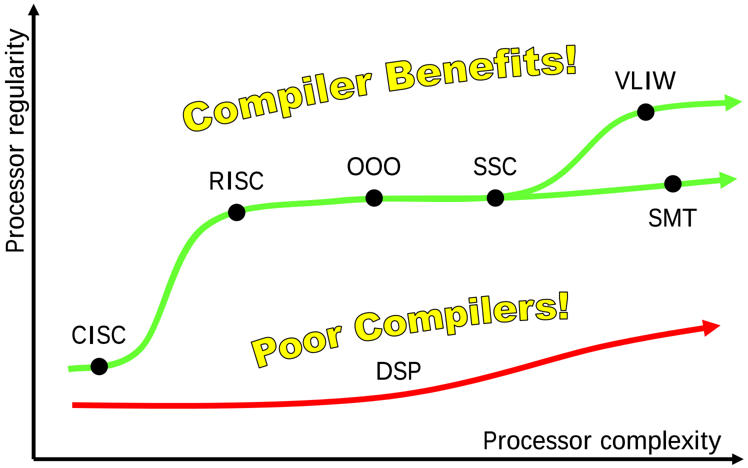
Types of Embedded Processors

- Microcontrollers
  - Relatively slow, microprogrammed, CISC processors
  - Typically derivates of old or very old general purpose processor families (68k, 8051, 6502, etc.)
- RISC Processors
  - Pipelined, relatively simple RISCs, often with special architectural features for the embedded market
  - Typical representatives: ARM7 and ARM9
- Digital Signal Processors
  - Special family of processors with peculiar architectures for arithmetic intensive, signal processing applications
  - Typical representatives: TI C620, DSP56k, etc.
- Multimedia Processors...

Completely Different Benchmarks

- General Purpose (Word, Powerpoint, gcc,...)
  - SPEC → Commercial
    - Scientific computing
    - Regular and irregular typical user applications...
- DSPs (IIR, FIR, FFT,...)
  - DSPstone (Aachen Uni) → Academic
  - EEMBC (pronounced "embassy") → Commercial
    - IIR, FIR, IDCT, FFT, IFFT, PWM,...
    - Matrix arithmetic, bit manipulation, table lookup, interpolation,...
    - Jpeg, RGB-to-CYMK, RGB-to-YIQ, Bezier curves, rotations,...
    - Viterbi, autocorrelation, convolutional encoders,...

Trends in Computing?



Pressure on the Compilers

- ❑ Performance
  - ❖ Squeeze out every possible MIPS of performance from irregular architectures
- ❑ Code Size
  - ❖ Memory is a key cost factor in embedded systems, much more than in general purpose systems
- ❑ Power Consumption
  - ❖ Important metric in embedded systems, hardly of any relevance in general purpose computing (i.e., not even considered by compilers)

Typical Features of DSPs

- ❑ Arithmetic and Datapath
  - ❖ Fixed-point arithmetic support
  - ❖ MAC = multiply-accumulate instruction
  - ❖ Special registers, not directly accessible
- ❑ Memory Architecture
  - ❖ Harvard architecture
  - ❖ Multiple data memories
- ❑ Addressing Modes
  - ❖ Special address generators
  - ❖ Bit-reversed addressing
  - ❖ Circular buffers
- ❑ Optimised Control
  - ❖ Zero-overhead loops
- ❑ Special-purpose peripherals...

DSP Arithmetic: Fixed-Point Vs. Floating-Point

- ❑ Typical example of embedded processor economics: **much more** complexity in designing the algorithm (NRE cost) and in programming to get **much less** complexity in the hardware (mfg. cost)
- ❑ Floating-point DSP **~2-4x** cost of Fixed-point DSP and **much slower**...
- ❑ Very poor support in automatic tools yet ➔ decisions taken by algorithm analysis, simulation, and compliance tests (e.g., accumulated error over a test set below some value)

Fixed Point

- ❑ In principle, if one adds a fractional point in a fixed position, hardware for integers works just as well and there are no additional ISA needs

$$\begin{array}{rccccccl}
 & 0 & 1 & 0 & 0 & 1_2 & \to & 9_{10} \\
+ & 0 & 0 & 0 & 1 & 1_2 & \to & 3_{10} \\
\hline
 & 0 & 1 & 1 & 0 & 0_2 & \to & 12_{10} \\
 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & &
\end{array}
\qquad
\begin{array}{rccccccl}
 & 0 & 1. & 0 & 0 & 1_2 & \to & 1.125_{10} \\
+ & 0 & 0. & 0 & 1 & 1_2 & \to & 0.375_{10} \\
\hline
 & 0 & 1. & 1 & 0 & 0_2 & \to & 1.500_{10} \\
 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} & &
\end{array}$$

- ❑ It's just a matter of representation! (I.e, implicit constant multiplicative coefficient)

Fixed Point Multiplication

- ❑ Multiplication typically introduces the need of arithmetic rescaling with shifts to the right (multiplicative constant cannot be implicit anymore) ➔ **Choice of accuracy** depending on how many bits one can keep...

$$\begin{array}{rccccccccccl}
 & & & & & & 0. & 1 & 0 & 1 & 0_2 & \to 0.625_{10} \\
\times & & & & & & 0. & 0 & 1 & 1 & 0_2 & \to 0.375_{10} \\
\hline
 & 0. & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0_2 & & \to 0.234375_{10} \\
 & \to & 0. & 0 & 0 & 1 & 1 & 1 & 1 & 0_2 & & \to 0.234375_{10} \\
 & \to & \to & 0. & 0 & 0 & 1 & 1 & 1 & 1_2 & & \to 0.234375_{10} \\
 & \to & \to & \to & 0. & 0 & 0 & 1 & 1 & 1_2 & & \to 0.21875_{10} \\
 & \to & \to & \to & \to & 0. & 0 & 0 & 1 & 1_2 & & \to 0.1875_{10}
\end{array}$$

Different Approximation Choices

- ❑ **Truncate**: Discard bits → Large bias
  - ❖ 00.011 ➔ 00 and 01.011 ➔ 01
  - ❖ 00.100 ➔ 00 and 01.100 ➔ 01
  - ❖ 00.101 ➔ 00 and 01.101 ➔ 01
- ❑ **Round**: <.5 round down, >=.5 round up → Small bias
  - ❖ 00.011 ➔ 00 and 01.011 ➔ 01
  - ❖ 00.100 ➔ 01 and 01.100 ➔ 10
  - ❖ 00.101 ➔ 01 and 01.101 ➔ 10
- ❑ **Convergent Round**: <.5 round down, >.5 round up, =.5 round to nearest even → No bias
  - ❖ 00.011 ➔ 00 and 01.011 ➔ 01
  - ❖ 00.100 ➔ 00 and 01.100 ➔ 10
  - ❖ 00.101 ➔ 01 and 01.101 ➔ 10

Fixed-Point Programming Example

```
/* an excerpt from adpcm.c */
/* adpcm_coder, mediabench */

/* Step 2 - Divide and clamp */
** This code *approximately* computes:
**    delta = diff*4/step;
**    vpdiff = (delta+0.5)*step/4;
** but in shift step bits are dropped. The net result of this is
** that even if you have fast mul/div hardware you cannot put it
** into good use since the fixup would be too expensive.
*/
delta = 0; vpdiff = (step >> 3);

if ( diff >= step ) { delta = 4; diff -= step; vpdiff += step; }
step >>= 1;
if ( diff >= step  ) { delta |= 2; diff -= step; vpdiff += step; }
step >>= 1;
if ( diff >= step ) { delta |= 1; vpdiff += step; }
```

Fixed-Point Programming Example

```
/* an excerpt from adpcm.c */
/* adpcm_coder, mediabench */

int index, delta;
…
index += indexTable[delta];
if (index < 0) index = 0;
if (index > 88) index = 88;
```

- ❑ Other classic DSP-type of operation: **accumulation with saturation**
- ❑ Also, as in previous example, **post multiplication shift**

DSP Arithmetic Needs

- ❑ Rather than having full floating-point support (expensive and slow), one wants in a DSP some **simple and fast** ad-hoc operations:
  - ❖ MUL + ADD in a **single cycle** (MAC)
  - ❖ **Accumulation** register after MAC (precision?)
  - ❖ **Approximation** mechanisms
- ❑ Nonuniform precision in the whole architecture (e.g., 24bit x 24bit + 56bit)

Slower Clock Speed but 1-Cycle Multiply-Accumulate Instruction

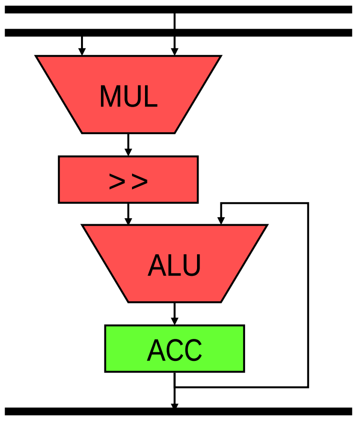
- ❑ MAC operations tend to dominate DSP code (maybe 50% of critical code) ➔ highly optimised MAC instruction

RISC:

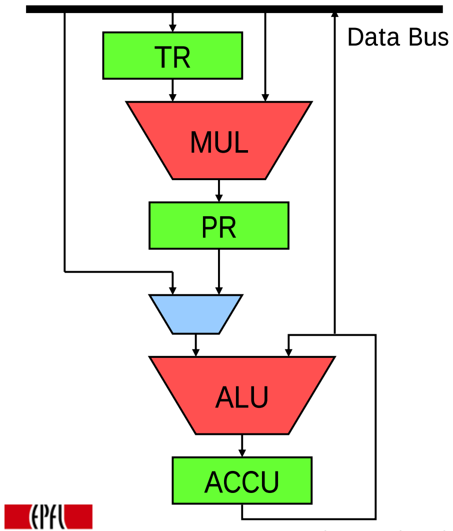
- ❑ Typ. 2 cycles: MUL
- ❑ 1 cycle: ADD
- ❑ 1 cycle: SHR
- ❑ Some more cycles: Saturation, Rounding, etc.

DSP:

- ❑ 1 cycle: "rich" MAC



Example of Pipelined MAC Datapath

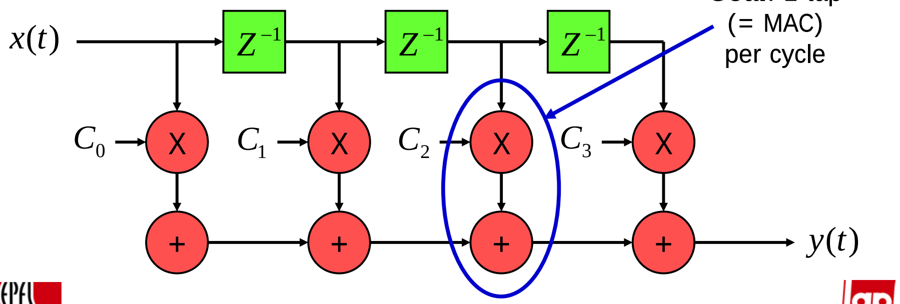


- ❑ Chained operations:
  - ❖ "Pipelined" MAC
- ❑ Many special registers:
  - ❖ Dedicated pipelining
  - ❖ Reduced pressure on general-purpose register file
  - ❖ Shorter instruction length (implicit operand addressing)
- ❑ Architecturally visible pipeline!

Classic FIR Example

- Convolution:

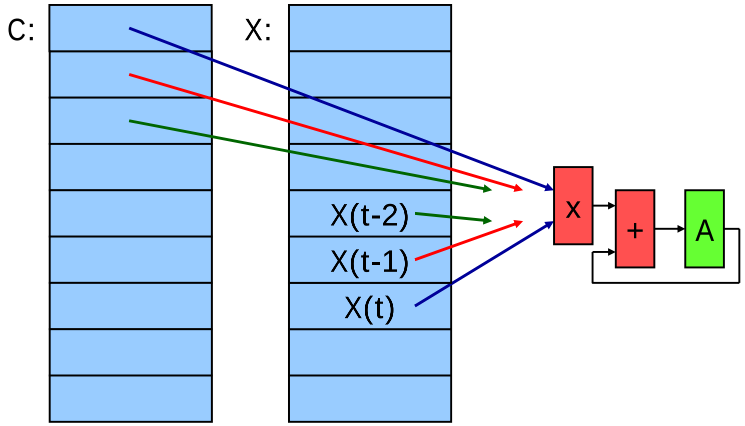
$$y(z)=\sum_{i=0}^{N-1} C_i \cdot x(z-i)$$



Goal: 1 tap (= MAC) per cycle

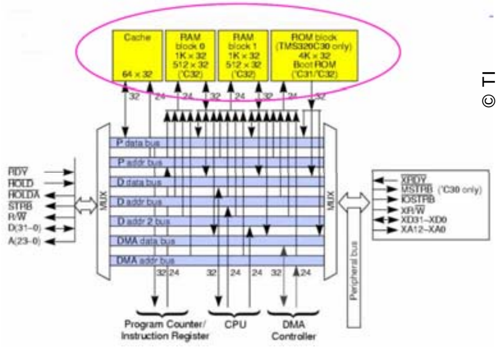
Memory Bandwidth

- The MAC instruction/unit is not enough...



Multiple Memory Ports

- Harvard architecture:
  - Separate instruction memory
  - I-Memory at times accessible as another D-Memory (e.g., TI C2000) to spare memory ports
- Multiple data memories:
  - X-Memory
  - Y-Memory
  - Sometimes more...
- Multiple buses



© TI

RISC vs. DSP Organisation

RISC:

- Von Neumann (Harvard but hidden from the user)
- ~1 access/cycle
- Heavily relies on caches to achieve performance
- Complex blend of on-chip SRAM/SRAM/DRAM

DSP:

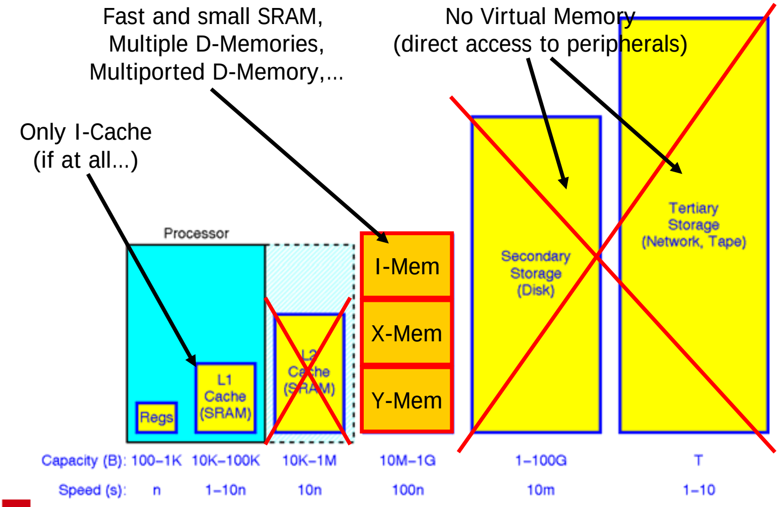
- Harvard (architecturally visible)
- 1-4 memory accesses per cycle
- No caches
- SRAM

Caches and DSPs

- Importance of **real-time constraints**: no data caches...
- Sometimes caches on the instruction memory, but **determinism** is key in DSPs:
  - Caches under programmer control to “lock-in” some critical instructions
  - Turn caches into fast program memory

Once again, one is not after **highest performance** but just the **guaranteed minimal performance** one needs

DSP vs. General Purpose Memory Systems



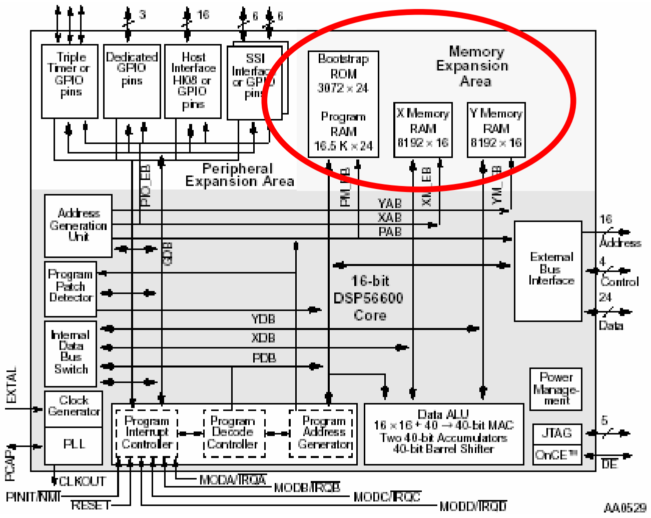
Fast and small SRAM, Multiple D-Memories, Multiported D-Memory,...

No Virtual Memory (direct access to peripherals)

Only I-Cache (if at all...)

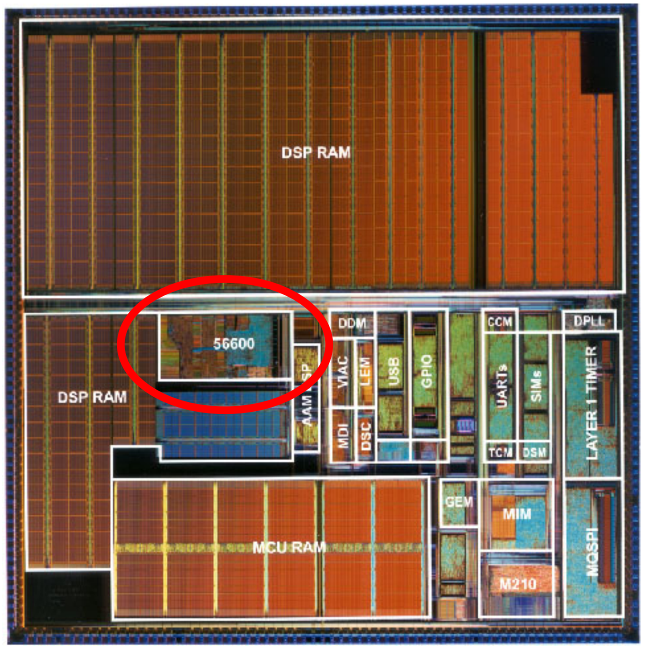
Processor
Regs
L1 Cache (SRAM)
L2 Cache (SRAM)
I-Mem
X-Mem
Y-Mem
Secondary Storage (Disk)
Tertiary Storage (Network, Tape)
Capacity (B): 100–1K 10K–100K 10K–1M 10M–1G 1–100G T
Speed (s): n 1–10n 10n 100n 10m 1–10

Example Motorola DSP56600



© Motorola

Baseband Chip



Courtesy of Motorola, © Motorola 2000

Addressing Modes

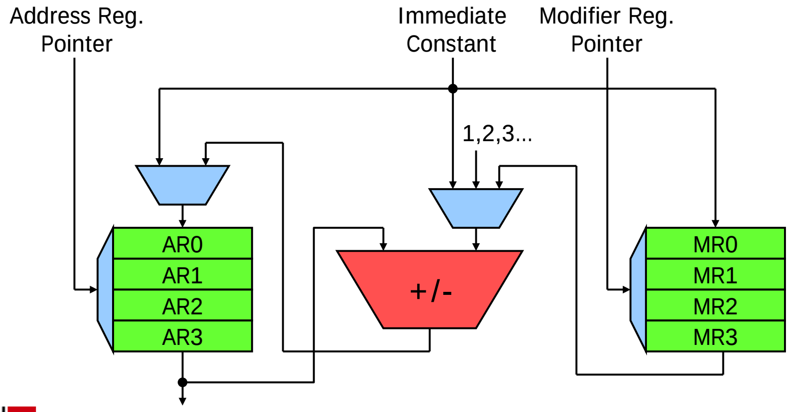
- ❑ To keep MAC busy all the time, with new data from memory, one needs to generate memory addresses
- ❑ Forget about Load/Store architectures
- ❑ Complex addressing is now fully welcome if
  - ❖ Allows automatic next address calculation
  - ❖ Does not require usage of the datapath (MAC is busy…)

Explicit parallelism/pipelining

```
   MPYF3 *AR0++%, *AR1++%, R0
|| ADDF3 R0, R2, R2
```

Address Generation Units

- ❑ Dedicated simple datapaths to generate meaningful sequences of addresses—usually 2-4 per DSP

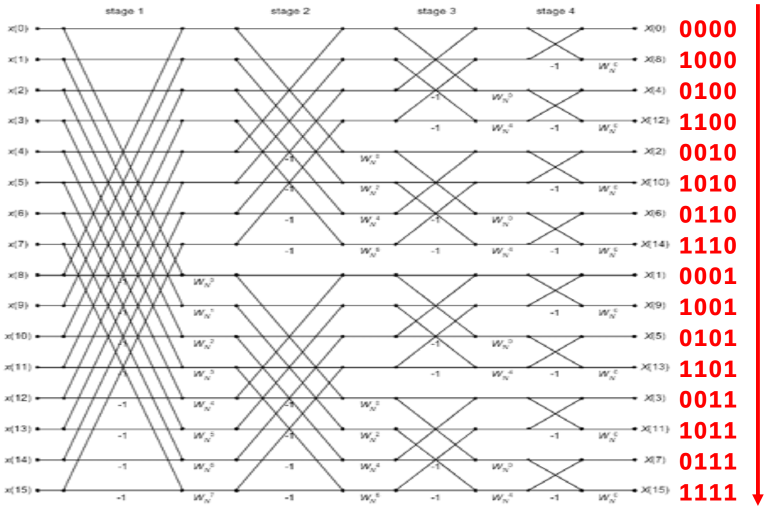


Typical Addressing Modes

AR can be loaded with:

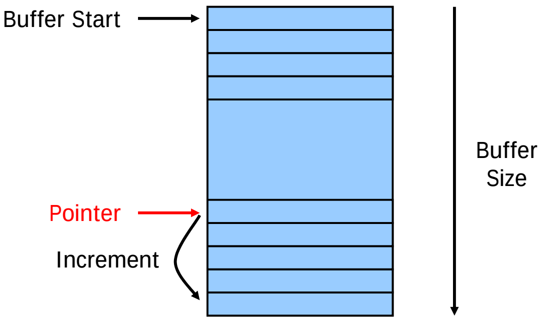
- ❑ **Immediate load**: constant from the instruction field loaded into the pointed AR
- ❑ **Immediate modify**: constant from the instruction field added to the pointed AR
- ❑ **Autoincrement**: small constant (typ. 1 and/or 2) added to the pointed AR
- ❑ **Automodify**: value of the pointed MR added to the pointed AR
- ❑ **Bit Reversing**: value of the pointed AR bit-reversed and loaded into the pointed AR
- ❑ **Modulo/Circular**: autoincrement/automodify with modulo
- ❑ Also decrement/subtract
- ❑ Sometimes pre- and/or post-modification

Radix-2 FFT



Circular Buffers

- DSPs deal with continuous I/O flows, often organised in circular buffers



- All DSPs generate “modulo” or “circular” addresses

Remove Control Bottlenecks

- Remember typical goal: FIR with MAC busy 100% of the time...
- DSP code made essentially of tight loops, often with a statically determined number of iterations (coefficients of a filter, etc.)
- How can one make the branches “cost nothing”?
  - Repeat instructions
  - Zero-overhead loops

Repeat/Loop Instructions

- For loops made of a single instruction:

```
          RPTS   N-1          ; repeat next
          MPYF3  *AR0++%, *AR1++%, R0
     ||   ADDF3  R0, R2, R2
```

- Zero-overhead Loop instruction:
  - Configures the Program Control Unit to generate the appropriate next address depending on a condition (e.g., autodecrement of an AR)

DSP World Is Slowly Changing

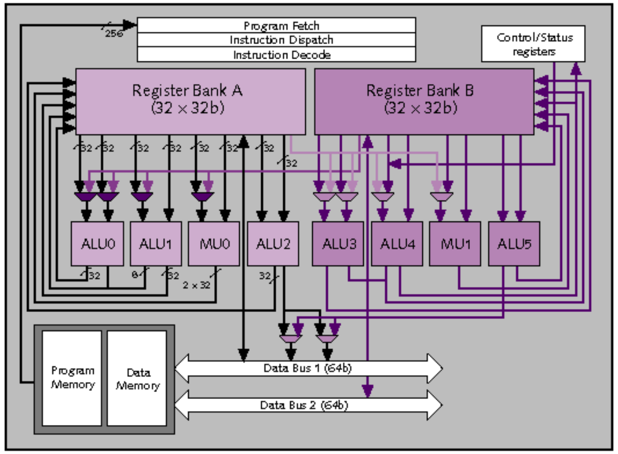
- Need of a fast development turnaround

  ➔Compilers!

- In a sense DSPs have already the main features of VLIWs: explicit parallelism, static scheduling, no “dynamic” low predictability behaviour...

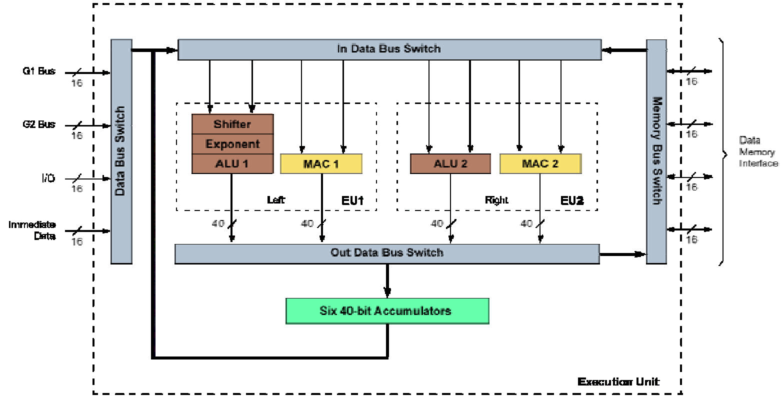
  ➔Convergence?

TI TMC320C64x



Source: Microprocessor Report, © MPR 2000

Infineon Carmel



Sort of VLIW but not all possible instructions are available: only 2048 via Configurable Long Instruction Words with compact coding

© Infineon 2000

Direct Carmel Translation for G.723.1 DC Filter

zero-overhead loop

```
repeat(Frame) block
  {
    a4 = *r0++ * *r1;
    a5 = (unsigned)a0l * *r1;
    a5 = (a5 >> 16) + a0h * *r1--;
    a0 = a4 + a5;
    *r4++ = round(a0);
  }
```

5 cycles

© Infineon 2000

Optimal Carmel Code for G.723.1 DC Filter

1 CLIW™ instruction

```
repeat(Frame) block
  {
    cliw dcf1(r0++)
    {
          a4  = *ma1 * ff1
       || a0  = a4 + a5
       || a5  = (unsigned)a0l * ff2
       || a1h = a0h;
    }
    cliw dcf2(r0, r4++)
    {
          a4  -= *ma1 * ff1
       || *ma2 = round(a0)
       || a5   = (a5 >> 16) + a1h * ff2;
    }
  }
```

1 CLIW™ instruction

2 cycles

© Infineon 2000

Summary

- DSPs are very different from general-purpose computers
  - Dedicated to embedded applications
  - Cost and power consumption come into the picture (and cost is fundamental)
- Relatively narrow variety of applications
  - More application specialisation possible
- Development cost (programming) relatively irrelevant when compared to per-unit cost
  - The most awkward and hard-to-program solutions are ok if they bring enough savings
  - Compilers? Useful for 90% of the code, but the rest...