Applications of signal processors

ARCHITECTURE OF DIGITAL SIGNAL PROCESSORS

Author: Grzegorz Szwoch

Gdańsk University of Technology, Department of Multimedia Systems

Digital signal processor

Our definition:

digital signal processor (DSP)

is a chipset specialized in optimal processing of digital signal samples, performing repeated operations on each sample.

In this lecture we will answer the question:

which features of a digital signal processor architecture allow for optimal processing of digital signals, better than a CPU?

Main features of DSP architecture

The most important features of DSP architecture, that separate them from general purpose processors, are:

- Harward architecture,
- pipelining,
- circular addressing,
- special instructions (MAC, vectorization, etc.).

DSP components

The most important components of a DSP:

- ALU – arithmetic-logic unit, operations: + – AND OR NOT XOR
- multiplier (*)
- FPU – floating point processing unit
- registers – memory cells holding data on which the processor operates,
- accumulator – a special register which holds intermediate results of the operations,
- address generator
- buses – lines for data exchange between registers and memory.

Performing operations

Example calculations in a program:

```
y = 0.5 * a + 0.3 * b + 0.2 * c
```

A typical sequence of operations on a processor:

- read data from memory (a, b, c, constants), write them to registers,
- execute operations (* * + * +), save intermediate results in the accumulator,
- copy the result to the memory (y).

Accumulator

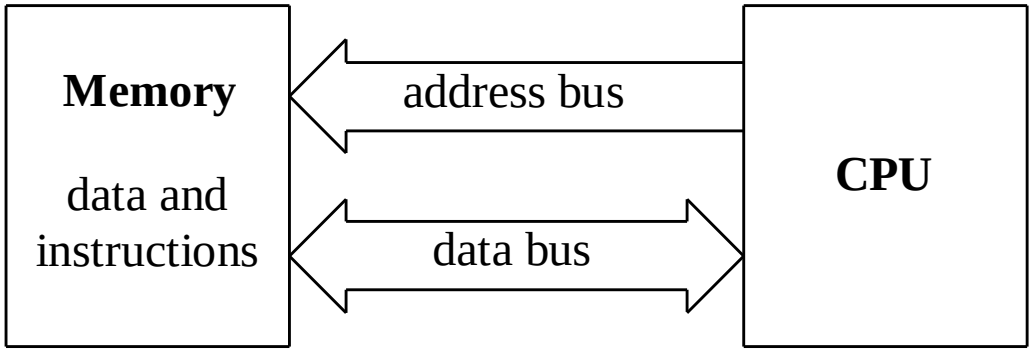
- Accumulator is a special register which holds results of most arithmetic and logic operations.
- On a 16-bit processor, multiplication of two 16-bit numbers yields a 32-bit result, so the accumulator must be at least 32 bits long.
- When the intermediate results are accumulated, the number can exceed 32 bits.
- Therefore, the accumulator has additional “guard bits”.
- On 16-bit processors, the accumulator can be 40-bit long.
- A programmer can write to and read from registers, including the accumulator.

Processor architectures

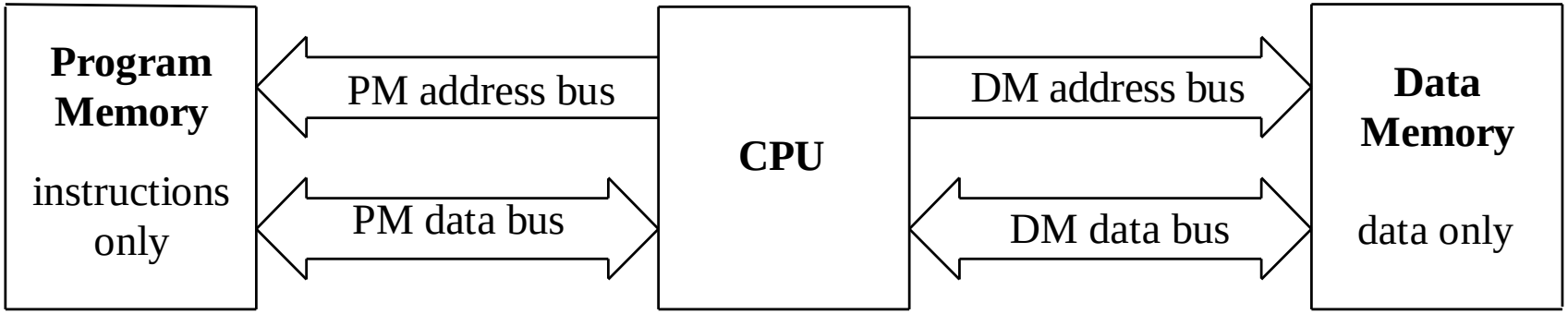
- von Neumann architecture
  - common memory for program and data,
  - used in general-purpose CPUs, e.g. in PC.
- Harward architecture
  - separate memory for program and data,
  - independent access to each memory,
  - used e.g. in DSPs.
- Harward architecture extensions in DSPs:
  - dual memory access (dual data buses),
  - instruction cache,
  - input/output controller.

Processor architectures

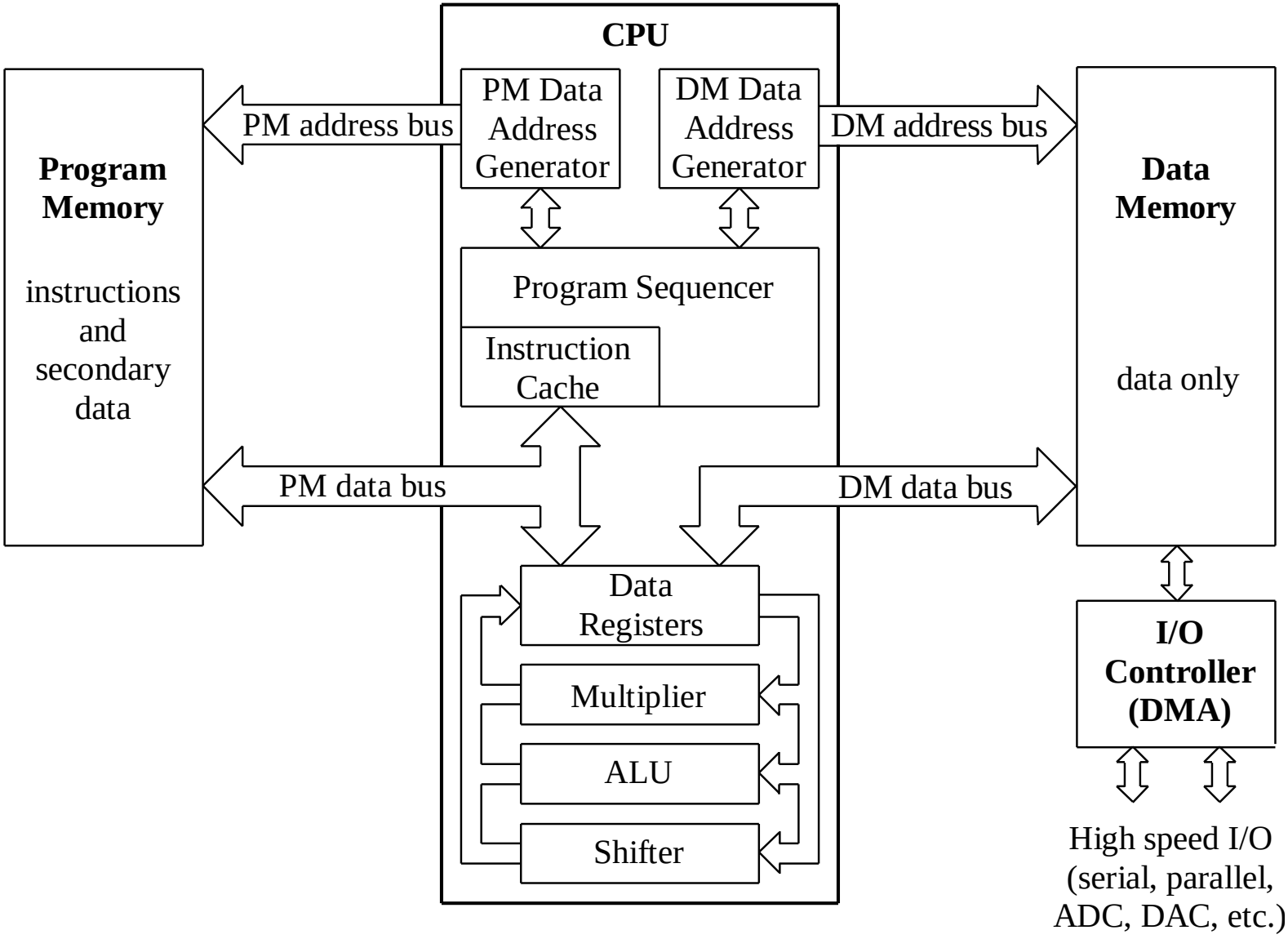
Von Neumann Architecture (*single memory*)



Harvard Architecture (*dual memory*)



Block diagram of a DSP



Processor cycles

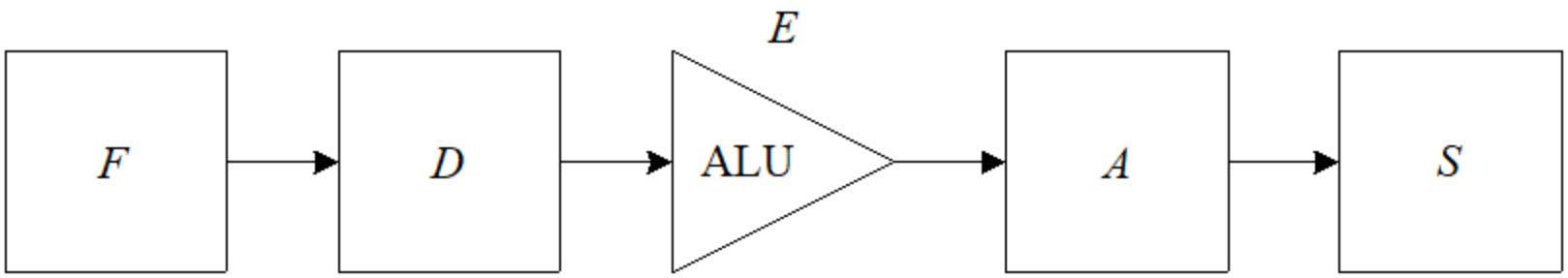
- Processor is controlled by a clock, frequency of which is set by a phase-locked loop (PLL) circuit.
- Each clock pulse starts a processor cycle.
- Each program instruction requires one or more cycles.
- Clock frequency determines the number of cycles available per second for program execution. For example, a 100 MHz clock means 100 million cycles per second.
- For an audio signal sampled with 48 kHz, we have 2083 cycles to process one sample.

Instruction execution

Execution of a single instruction may be divided into stages:

- F (*fetch*) – get the instruction from memory or cache,
- D (*decode*) – decode the instruction
- E (*execute*) – run the instruction
- A (*access*) – open the memory
- S (*store*) – write the result to memory

Often, only F, D, E stages are considered.



Sequential processing

In sequential processing, a new instruction can be started only after the previous one has completed.

Clock cycle	1	2	3	4	5	6	7	8	9	10	
Instr. 1	*F*1	*D*1	*E*1	*A*1	*S*1						
Instr. 2						*F*2	*D*2	*E*2	*A*2	*S*2	
⋮											

Pipelining

Pipelining is performed as follows.

- First stage (F) of the first instruction is performed.
- When the processor proceeds to the second stage (D), the first stage (F) of another instruction is started.
- Instruction stages are performed with overlapping, which speeds up the program execution.
- Pipelining is used on digital signal processors.
- Conflict cases (hazards), such as jump to another instruction in the code, break the pipeline, result in reverting partially performed instructions and restarting the pipeline.

Pipelining

Clock cycle	1	2	3	4	5	6	7	8	9	10	
Instr. 1	*F*1	*D*1	*E*1	*A*1	*S*1						
Instr. 2		*F*2	*D*2	*E*2	*A*2	*S*2					
Instr. 3			*F*3	*D*3	*E*3	*A*3	*S*3				
Instr. 4				*F*4	*D*4	*E*4	*A*4	*S*4			
Instr. 5					*F*5	*D*5	*E*5	*A*5	*S*5		
Instr. 6						*F*6	*D*6	*E*6	*A*6	*S*6	
⋮											

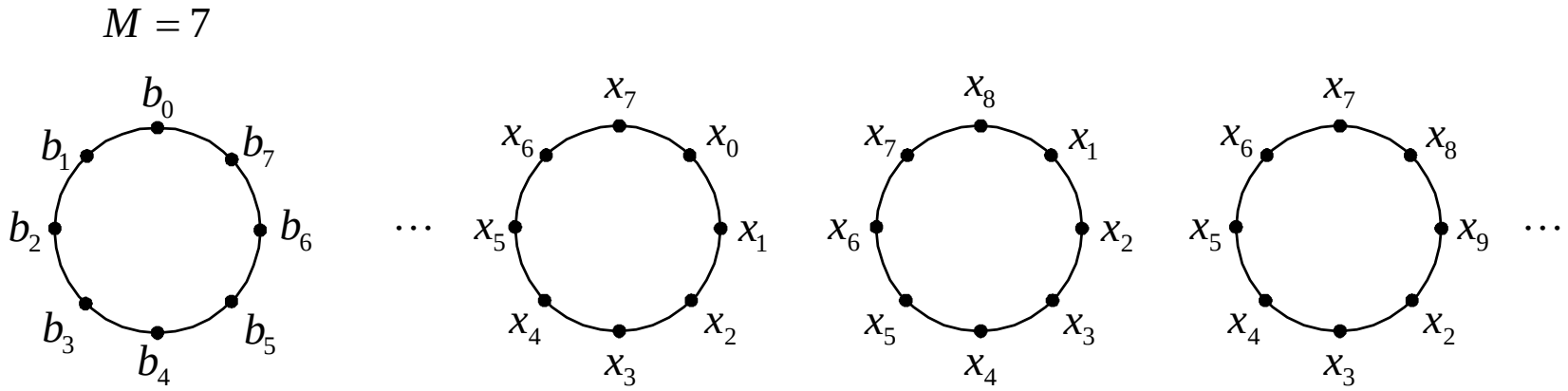
Linear buffer

A common case in digital signal processing (e.g. FFT filter).

- *N* latest samples are processed.
- Samples are stored in a buffer in memory.
- A new sample arrives:
  - the oldest sample is removed,
  - the remaining samples are shifted by one position,
  - a new sample is written at the end of the buffer.
- This is a linear buffer.
- Processor cycles are wasted for moving samples in memory.

Circular buffer

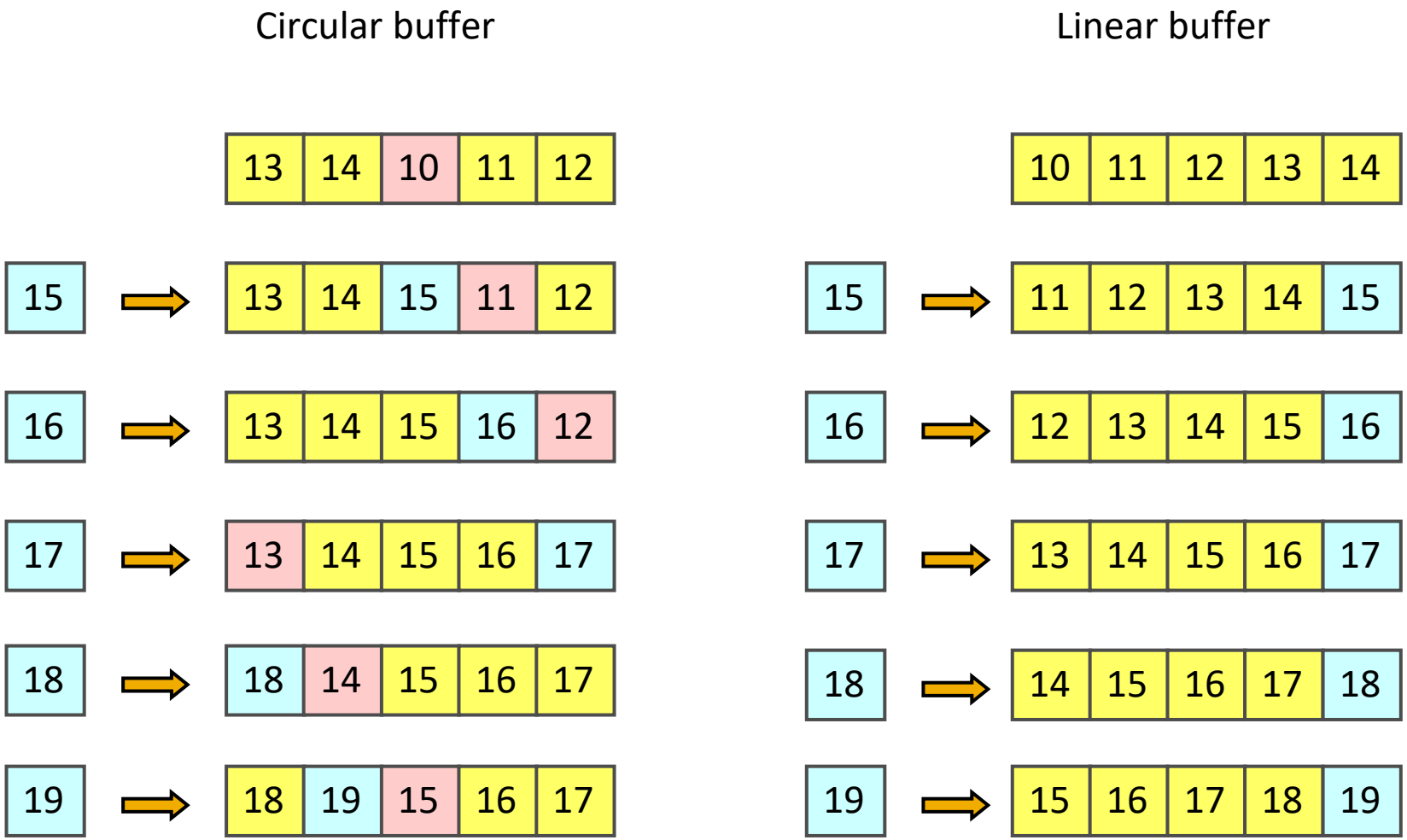
- A circular buffer may be visualized as a ring.
- Pointer (index) indicates the current write position (the oldest sample).
- A new sample is written at the pointer.
- The pointer is advanced to the next position.
- No data is moved, the other samples remain in position.



Circular addressing

- In practice, the circular buffer is implemented as a normal linear buffer and a pointer (index).
- The pointer is moved, it indicates the order in which the samples are processed.
- Once the pointer reaches the buffer end, it wraps to the beginning.
- A linear buffer in memory uses circular addressing.
- Digital signal processors have circular addressing implemented in hardware.

Circular and linear buffer – an illustration



Circular addressing

On a standard CPU, we have to wrap the pointer manually.

```
buffer[index] = new_sample;          // write
// … do the processing
index = index + 1;                   // advance the index
if (index == N)                      // end of buffer
    index = 0;                       // wrap index
```

On DSP, we use circular addressing

- in Assembler – by turning on the circular mode,
- in C – by using a special instruction (*intrinsic*):

```
buffer[index] = new_sample;          // write
// … do the processing
index = _circ_incr(index, 1, N)      // advance the index
                                     // with wrapping
```

MAC

- A common operation is signal processing: multiply numbers and accumulate the results

  $y \leftarrow y + a * x$

- MAC = multiply and accumulate.
- On a standard CPU, we must do multiplication and addition separately.
- Signal processors have MAC implemented in hardware, as a single processor instruction.
- This reduces the number of used cycles and speeds up program execution.
- Many modern DSPs can do two MAC operations at the same time (*dual MAC*).

MAC in practice

Standard CPU:

```
for (i = 0; i < N; i++) {
    result += buffer[index] * coeff[index];
    index = index + 1;
    if (index == N) index = 0;
}
```

DSP with MAC

```
for (i = 0; i < N; i++) {
    result = _smac(result, buffer[index], coeff[index]);
    index = _circ_incr(index, 1, N);
}
```

SIMD (vectorization)

- Another common case: multiplication of two vectors.
- Requires *N* multiplications (*N* = vector length).
- Floating point numbers may be written with single (4 bytes) or double (8 bytes) precision.
- A processor can multiply two 4B or two 8B numbers.
- Two 4B numbers can be packed into one 8B number.
- One 8B multiplication instead of two 4B multiplications.
- The number of multiplications is reduced to *N*/2.
- This is vectorization, or SIMD (*single instruction, multiple data*) – the same operations on different data.
- CPUs also allow vectorization (SSE extensions).

Vectorization - an example

Without vectorization:

```
for (i = 0; i < N; i++) {
    y[i] = a[i] * b[i];
}
```

With vectorization – more code, less operations:

```
for (i = 0; i < N; i+=2) {
    _amem8_f2(&y[i]) =
       _dmpysp(_amem8_f2(&a[i]), _amem8_f2(&b[i]));
}
```

Memory organization

DSP memory is divided physically and logically into several levels. Each successive level has slower access time.

- L1 – cache memory, on DSP
  - for internal use of the processor.
- L2 – DSP internal memory (on chip)
  - can be used by a programmer (program and data),
  - usually small (e.g. 1 MB).
- L3 – external memory (off chip)
  - a separate memory module, e.g. DDR3
  - much slower than L2 but can be bigger (GBs).

ROM memory, often flashable – program and constants.

SARAM and DARAM

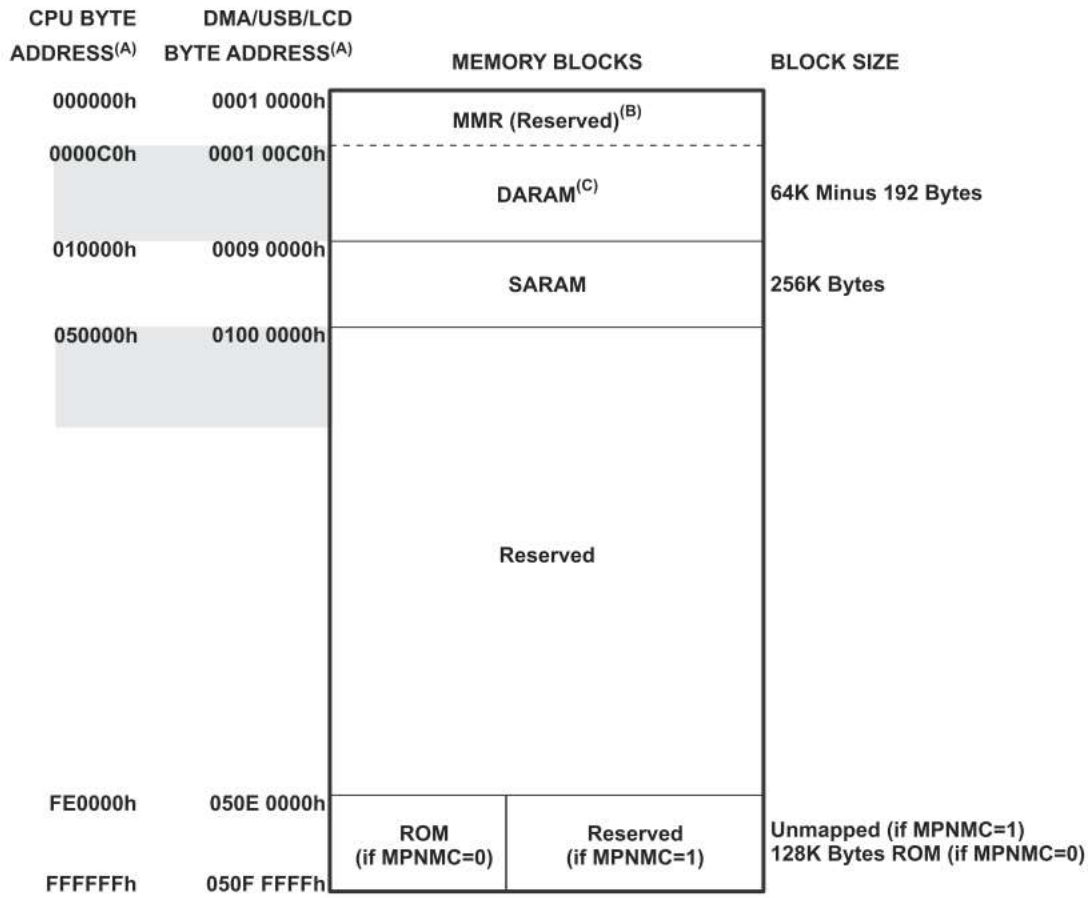
- SARAM (*single access random access memory*)
  – a standard memory, one memory read or write at a time.
- DARAM (*dual access RAM*) – dual data bus, two operations at a time (two writes, two reads or read + write).
- On a DSP: the whole memory may be DARAM, or part of memory may be DARAM and the rest is SARAM.
- Memory is divided into banks (pages) – concurrent access to different banks.
- A programmer must consider which data to put into DARAM and which may be in SARAM.

Memory map

- A memory map is specific for a DSP model, it is determined by the manufacturer.
- Each memory area is assigned to a range of addresses.
- Address is a number which determines the position in memory.
- A memory map assigns the logical addresses to physical memory sections.
- It is required in each DSP program – the compiler must have it to compile a program.

Memory map

An example of a memory map, from C5535 documentation:



Memory map

An example of memory map definition (C5535):

```
MEMORY
{
 PAGE 0:  /* ---- Unified Program/Data Address Space ---- */

  MMR    (RWIX): origin = 0x000000, length = 0x0000c0  /* MMRs */
  DARAM0 (RWIX): origin = 0x0000c0, length = 0x00ff40  /*  64KB - MMRs */
  SARAM0 (RWIX): origin = 0x010000, length = 0x010000  /*  64KB */
  SARAM1 (RWIX): origin = 0x020000, length = 0x020000  /* 128KB */
  SARAM2 (RWIX): origin = 0x040000, length = 0x00FE00  /*  64KB */
  VECS   (RWIX): origin = 0x04FE00, length = 0x000200  /*  512B */
  PDROM   (RIX): origin = 0xff8000, length = 0x008000  /*  32KB */

 PAGE 2:  /* -------- 64K-word I/O Address Space -------- */

  IOPORT (RWI) : origin = 0x000000, length = 0x020000
}
```

Memory sections

Logical memory sections are assigned to addresses.

The main sections of a compiled program are:

- .text – program code
- .stack – stack (locally declared variables)
- .data – initialized variables
- .bss – global and static variables
- .const – constants
- .sysmem – stack (dynamically allocated memory)

A programmer may create custom sections.

Memory sections

An example of memory sections definition for a compiler (C5535):

```
SECTIONS
{
   .text     >> SARAM1|SARAM2|SARAM0  /* Code                          */
   .stack    >  DARAM0                /* Primary system stack          */
   .sysstack >  DARAM0                /* Secondary system stack        */
   .data     >> DARAM0|SARAM0|SARAM1  /* Initialized vars              */
   .bss      >> DARAM0|SARAM0|SARAM1  /* Global & static vars          */
   .const    >> DARAM0|SARAM0|SARAM1  /* Constant data                 */
   .sysmem   >  DARAM0|SARAM0|SARAM1  /* Dynamic memory (malloc)       */
   .switch   >  SARAM2                /* Switch statement tables       */
   .cinit    >  SARAM2                /* Auto-initialization tables    */
   .pinit    >  SARAM2                /* Initialization fn tables      */
   .cio      >  SARAM2                /* C I/O buffers                 */
   .args     >  SARAM2                /* Arguments to main()           */
    vectors  >  VECS                  /* Interrupt vectors             */
   .ioport   >  IOPORT PAGE 2         /* Global & static ioport vars */
   .fftcode  >  SARAM0                /* Custom sections */
   .input    >  DARAM0, align(4)
}
```

Using sections in C code

This is how we can create a buffer in DARAM or SARAM,
in a default .bss section:

```
int buffer[8192];
```

If we declare an external memory in a memory map:

```
.ddr > DDR3
```

we can create a buffer in DDR memory using a compiler pragma
(example for a TI processor):

```
#pragma DATA_SECTION(bufor, "ddr");
int buffer[8192];
```

Internal and external memory

Which data in internal L2 memory (DARAM/SARAM)?

- program code, stack, heap
- most variables
- buffers that are often accessed

Which data in external L3 memory?

- large buffers that don’t fit in L2
- rarely accessed data
- archived processing results

Memory sections - remarks

In C programs:

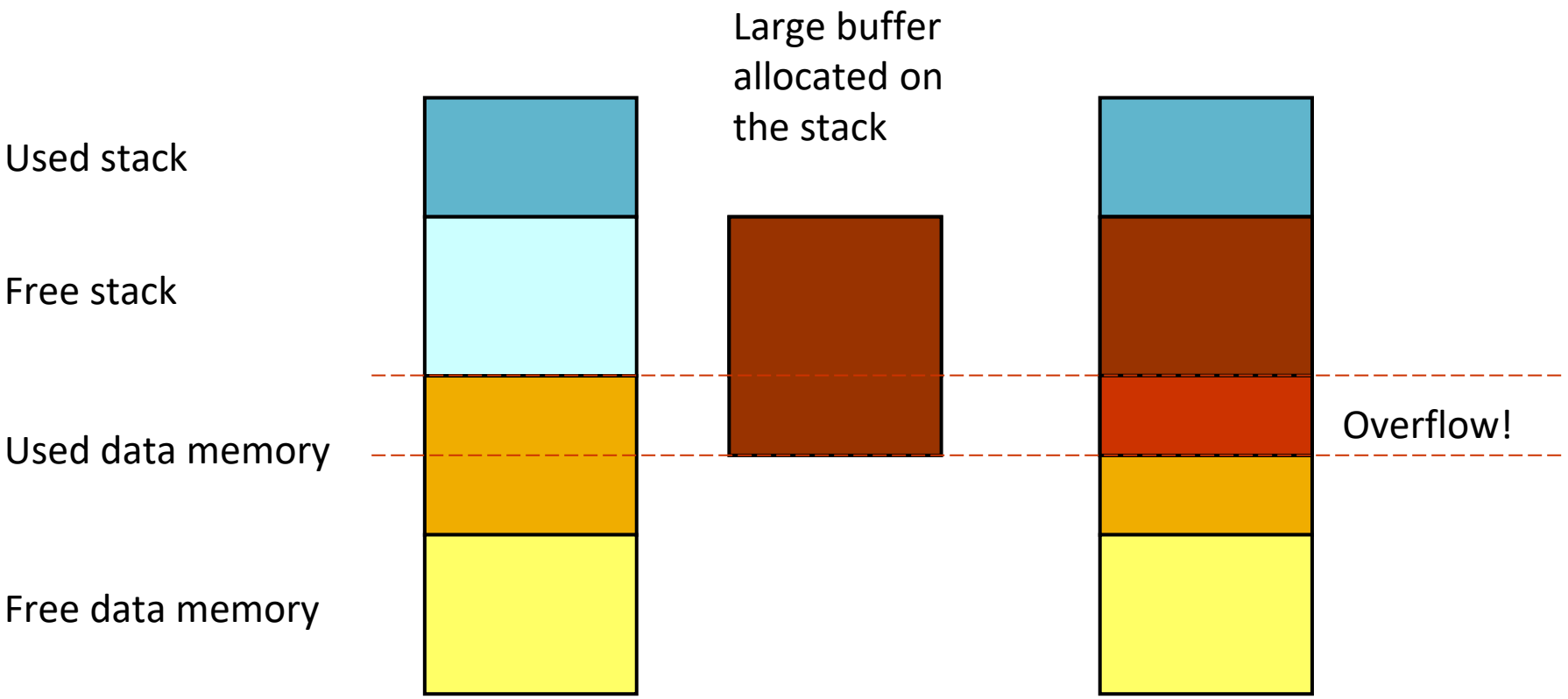
- Globally declared variables (in main code, outside functions) and static variables – in *.bss*.
- Local variables, declared inside functions (including *main*) – in *.stack*.
- Dynamically created variables (with *malloc*) – in.*sysmem*.
- Constants (e.g. filter coefficients) – in *.const*.

Practical implications:

- do not declare large buffers inside functions – the stack is small, and it may overflow
- constant data, such as filter coefficients, should be declared as *const*.

Stack overflow

Stack overflow occurs when memory allocation for a variable on the stack goes outside of a stack space.



Stack overflow

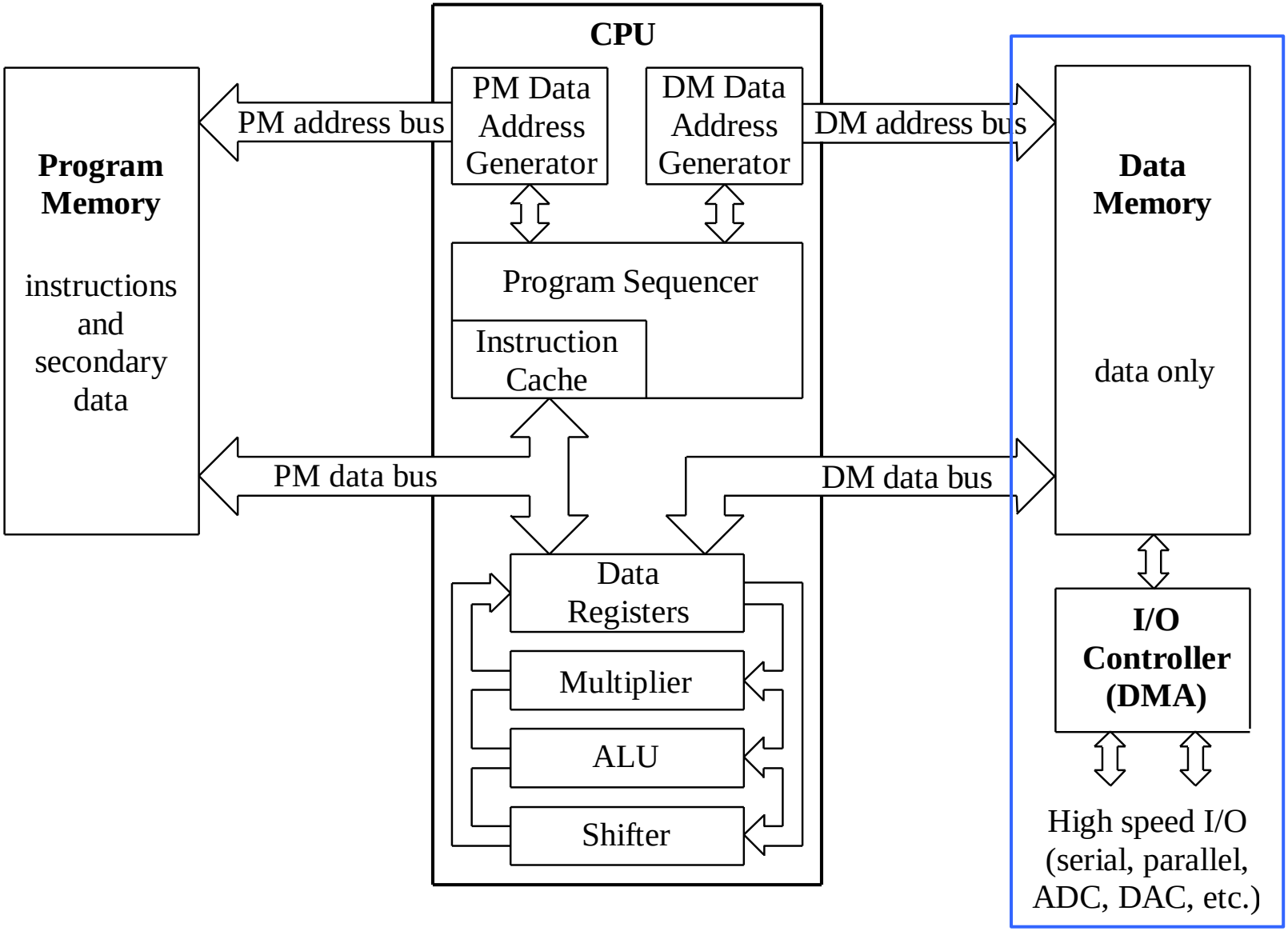
What happens when a stack overflow occurs?

- On a desktop OS (e.g. Windows), the program crashes.
- There is no protection on a DSP! The memory is overwritten without notice. What may happen:
  - if the overwritten memory section was unused, the program continues normally (for now),
  - if data were stored in this section, they are lost; the program can hang, or it may continue, but generate incorrect results!
- It's very hard to debug stack overflow errors.
- Therefore, it's best to use the rule: all buffers (tables) are declared in the main program section, not inside functions!

Direct memory access

- Signal samples that are received by the interfaces must be written to memory.
- DMA (direct memory access) – interfaces have direct access to the memory, without the processor.
- Data are transferred to/from memory without executing any code by the processor, they do not use processor cycles.
- The program executes much faster, it is not blocked by the interfaces.
- Almost all DSPs use DMA.

Direct memory access



Interrupts

How should a program know that new data are available?

- Polling – continuously checking for new data
  - uses processor cycles for checking
  - introduces delays
- Interrupts – a better approach:
  - when the DMA controller writes new data to memory, it generates an interrupt – an informational message
  - a programmer writes an interrupt handling procedure that is called when new data are available
  - interrupts have higher priority than a normal code – they interrupt program execution
  - lower delays, no processor cycles are wasted

Remarks on a C compiler (1)

- Each variable type takes a defined number of bytes, e.g. *float* typically uses 4 bytes.
- Variable address is an integer that defines the position of a variable in memory.
- Alignment – requirement that the address is divisible by the type size (*float*: by 4).
- In some cases (dual operations), alignment requires that the address is divisible by 2 × type size (*float*: by 8).
- Alignment is often a requirement from a compiler to generate an optimized code.

Remarks on a C compiler (1)

Alignment must be forced by using compiler pragmas.

Example for a TI processor – alignment to 8 bytes:

```
#pragma DATA_ALIGN(buffer, 8);
int buffer[8192];
```

Remarks on a C compiler (2)

- We want to generate code for a loop using dual MAC – two MACs in each loop iteration.
- By default, compiler usually won't do that, because it doesn't know if all the conditions are fulfilled:
  - a loop will be executed even number of times,
  - the loop won't break at some point,
  - there is no overlap of buffers in memory.
- Therefore, the compiler plays safe (according to Murphy's law) and it generates suboptimal code.
- If we write program in Assembler, we have full control over the code and we can optimize it ourselves.

Remarks on a C compiler (2)

Again, we have to use pragmas to inform the compiler:

- how many times the loop will iterate (*MUST_ITERATE*),
- how to unroll the loop (*UNROLL*),
- that buffers do not overlap (*restrict*)

Example (TI processor):

```
void vecmul(int* restrict y, int* restrict a,
            int* restrict b, int n)
{
    int i;
    #pragma MUST_ITERATE(2,,2)
    #pragma UNROLL(2)
    for (i = 0; i < n; i++)
       y[i] = a[i] * b[i];
}
```

Remarks on a C compiler (conclusion)

- In Assembler, we can generate optimal code, but it's our duty to ensure that it works correctly.
- The C compiler must ensure that the program **always** works correctly, even if it works slower. In case of any "risk", optimizations are disabled.
- A programmer must use "magic pragmas" to inform the compiler that the code can be optimized.
- However, in many cases, the compiler decides that it knows better ☺. It doesn't generate the code we want.
- In such cases, we can only write the code in Assembler (or maybe the compiler is right?).