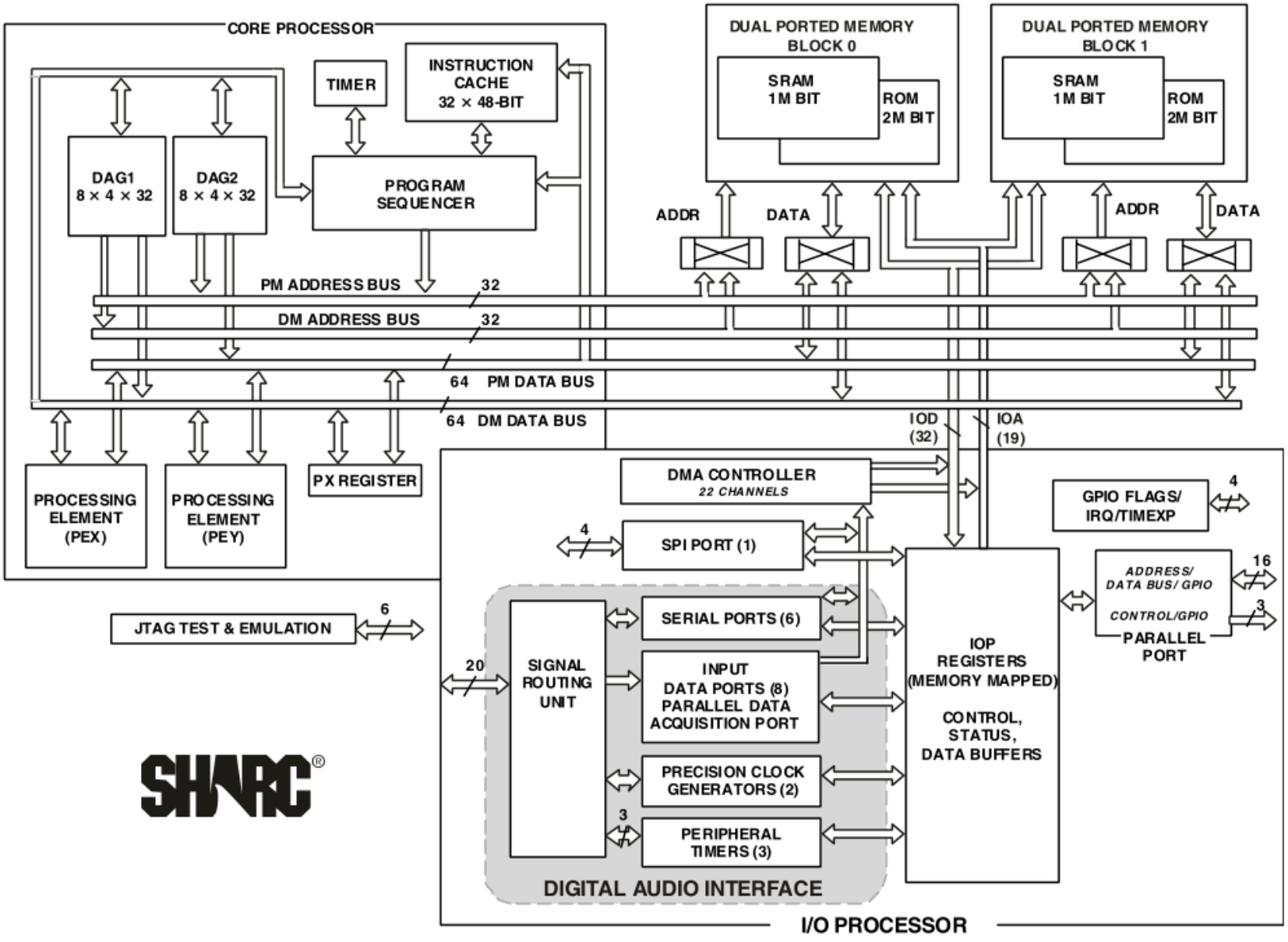
ETIN80 — Algorithms in Signal Processors

Signal Processor

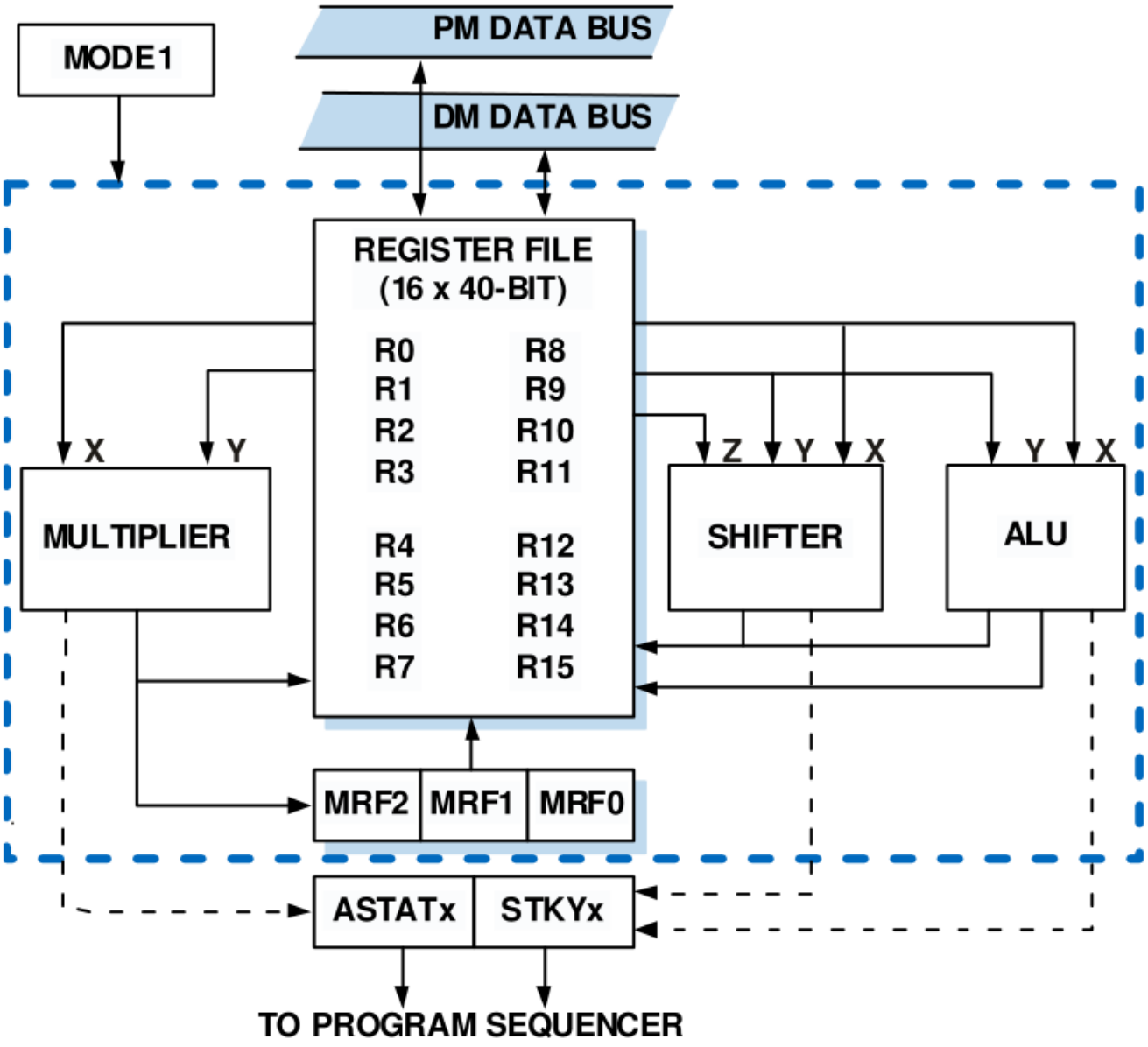
Tekn.Dr. Mikael Swartling

Lund Institute of Technology
Department of Electrical and Information Technology

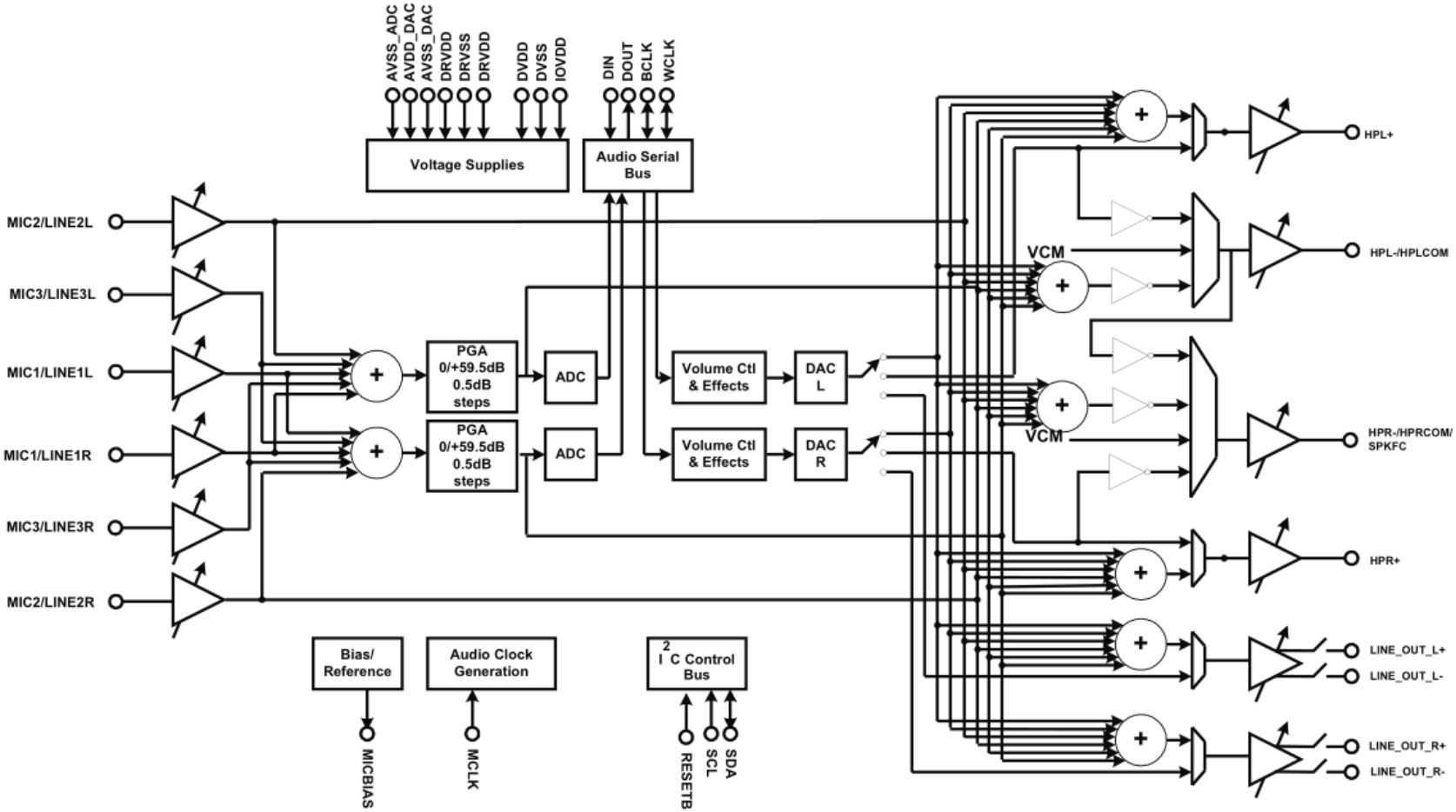
Hardware Architecture



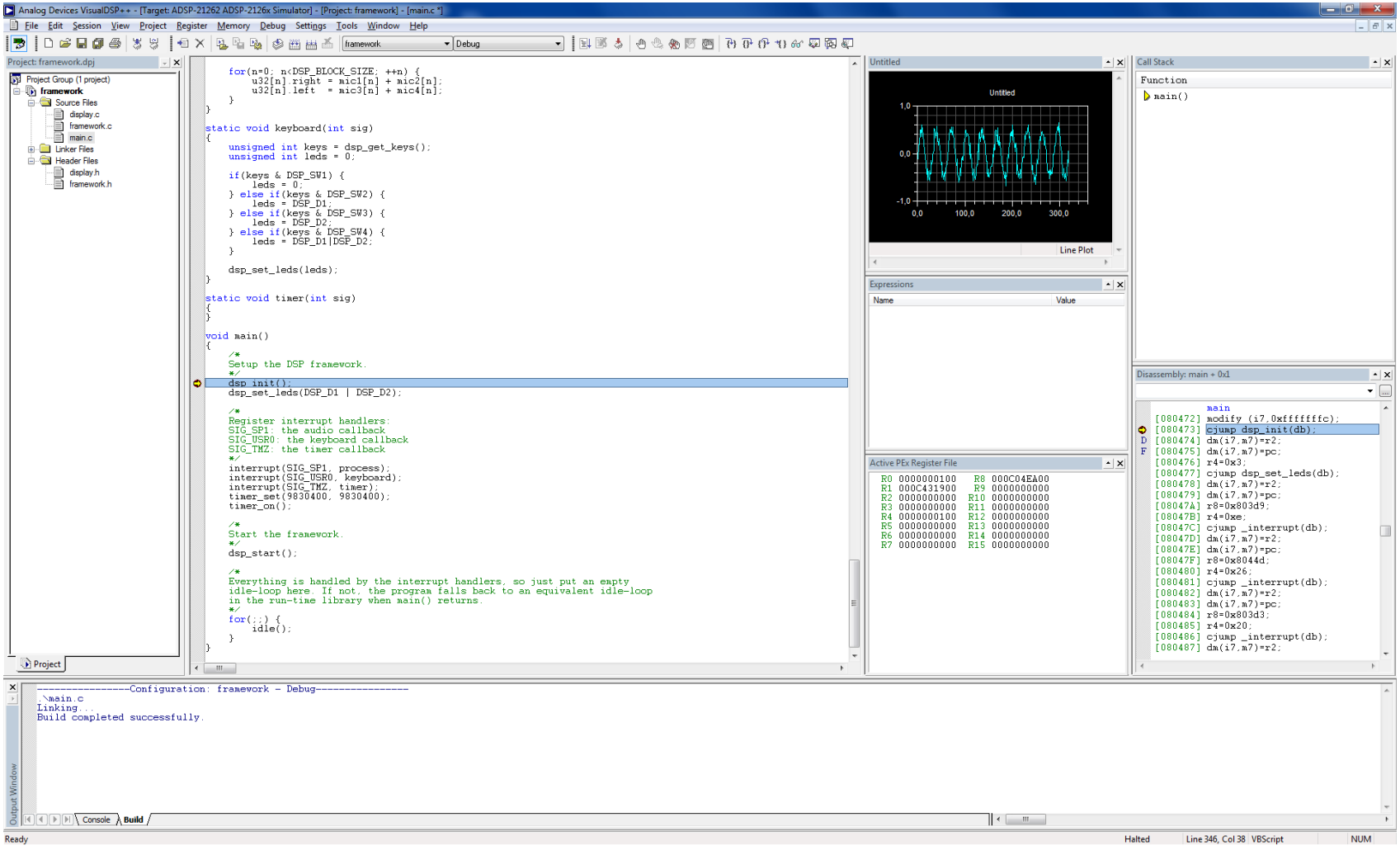
Hardware Architecture



Hardware Architecture



Integrated Development Environment



Integrated Development Environment

Visual DSP++ 5.

Workspace and project manager.

- Optimizing compiler for C, C++ and assembly.
- Simulator and in-circuit emulator.
- Automation scripting.

Extensive debugger.

- Expression evaluation.
- Core register views.
- Graphs and image view of memory.

Integrated Development Environment

Standard libraries.

Complete C and C++ run-time libraries.

- In-circuit file and console I/O.

Signal processing library.

- Matrix and vector functions.
- Real and complex data.
- Filter functions.
- Fourier transforms.

Data Types

Common data types are supported.

- 32 bit integer types.

  `char`, `short`, `int`, `long`
- 32 bit IEEE 754-compliant floating point types.

  `float`, `double`
- 32 bit Q1.31 fractional fixed point types.

  `fract`

Common operators are supported.

- Division is software emulated.
- Trigonometry and other functions are software emulated.

Data Types

Some types are extended length.

Software emulated extended types.

- 64 bit integers and floating point values.

  `long long`, `long double`

Hardware compute registers only available in assembly.

- 40 bit registers including 8 extension bits.
- 80 bit accumulator including 16 guard bits.

Guard bits are used to prevent overflow in accumulation loops.

Data Types

Fractional fixed point values.

Integers values from positive powers of 2.

$$v = -b_{31} \cdot 2^{31} + b_{30} \cdot 2^{30} + \cdots + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

Fixed point values from positive and negative powers.

$$v = -b_{31} \cdot 2^0 + b_{30} \cdot 2^{-1} + \cdots + b_1 \cdot 2^{-30} + b_0 \cdot 2^{-31}$$

Tradeoff between integers and floating point values.

Interrupts

Interrupt driven design.

A signal from the hardware or software indicating an event that needs immediate attention.

- Asynchronous program execution.
- Interrupt-driven design preferred.

  The DSP informs the program when something happens.
- Poll-driven design when necessary.

  The program asks the DSP if something has happened.

Register functions that are called in response to an interrupt.

Interrupts

Program sequence during interrupts.

Assume an example with main thread and three interrupts.

- High-priority audio process callback.
- Medium-priority timer callback.
- Low-priority keyboard callback.
- Idle-priority main thread.

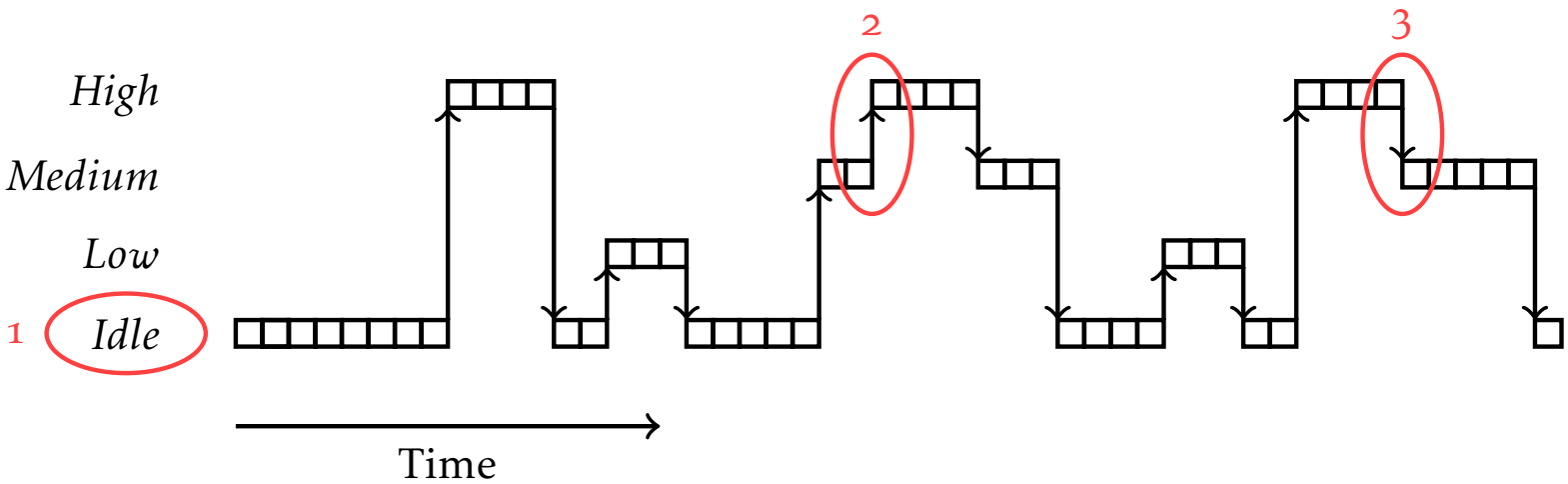
```
void main() {
    ...
    interrupt(SIG_SP1, process);
    interrupt(SIG_TMZ, timer);
    interrupt(SIG_USR0, keyboard);

    for(;;) {
        idle();
    }
}
```

Interrupts

Program sequence during interrupts.

Interrupt sequencing is automatic.

1. Main thread runs when no interrupts are active.
2. Higher procedures interrupts lower procedures.
3. Lower procedures waits for higher procedures.



Addressing Modes

Addressing modes determine how memory is accessed.

Normal addressing such as pointers and array indexing.

- `*ptr`
- `*(ptr+offset)`
- `ptr[offset]`

Normal addressing with pointer and index advancing.

- `*ptr++`
- `ptr[offset++]`

Advanced addressing such as circular and bit-reversal.

- `ptr[(start+offset) % size]`

Program Memory for Constant Buffers

The ADSP-21262 has two memory banks.

The two memory banks can be read in parallel.

- Code uses the PM bank.
- Data uses the DM bank by default.
- Data can also be put in the PM bank.

Persistent States

An FIR filter example.

Implement an FIR filter.

$$y(n) = \sum_{k=0}^{K-1} x(n-k)h(k)$$

The filter state has to persist between signal frames.

- Previous samples $(K-1)$ has to be preserved.
- Any information or state that is updated over time.

Persistent States

An FIR filter example.

See the function `filter` in Matlab about persistent states.

- `[y, zf] = filter(b, a, x, zi)`

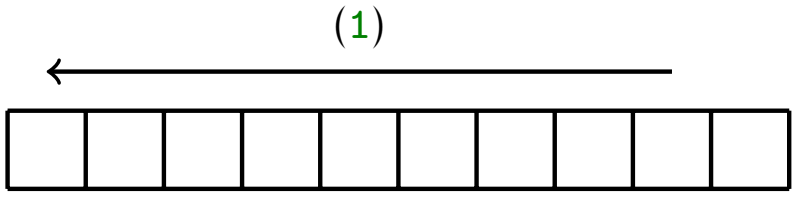
```
function myproject
    x = audioread('input.wav');

    xb = buffer(x, 320);
    [M, N] = size(xb);
    yb = zeros(M, N);

    [b, a] = ...
    z = [];

    for n = 1:N
        [yb(:, n), z] = filter(b, a, xb(:, n), z);
    end

    y = yb(:);
end
```

Circular Addressing

An FIR filter example.

Implementation using buffer shift.

```
float const pm coeff[10] = {...};
float          state[10] = {0};

float filter(float x) {
    int k;
    float y = 0;

    // Shift (1)
    for(k=0; k<9; ++k) {
        state[k] = state[k+1];
    }

    // Insert (2)
    state[9] = x;

    // Index (3)
    for(k=0; k<10; ++k) {
        y += state[k] * coeff[k];
    }

    return y;
}
```

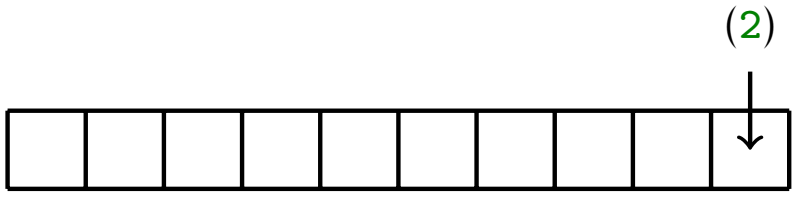


Circular Addressing

An FIR filter example.

Implementation using buffer shift.

```
float const pm coeff[10] = {...};
float          state[10] = {0};

float filter(float x) {
    int k;
    float y = 0;

    // Shift (1)
    for(k=0; k<9; ++k) {
        state[k] = state[k+1];
    }

    // Insert (2)
    state[9] = x;

    // Index (3)
    for(k=0; k<10; ++k) {
        y += state[k] * coeff[k];
    }

    return y;
}
```

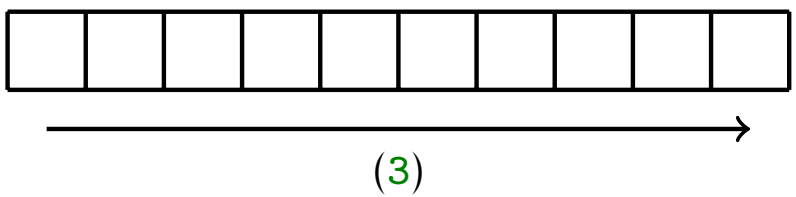


Circular Addressing

An FIR filter example.

Implementation using buffer shift.

```
float const pm coeff[10] = {...};
float          state[10] = {0};

float filter(float x) {
    int k;
    float y = 0;

    // Shift (1)
    for(k=0; k<9; ++k) {
        state[k] = state[k+1];
    }

    // Insert (2)
    state[9] = x;

    // Index (3)
    for(k=0; k<10; ++k) {
        y += state[k] * coeff[k];
    }

    return y;
}
```

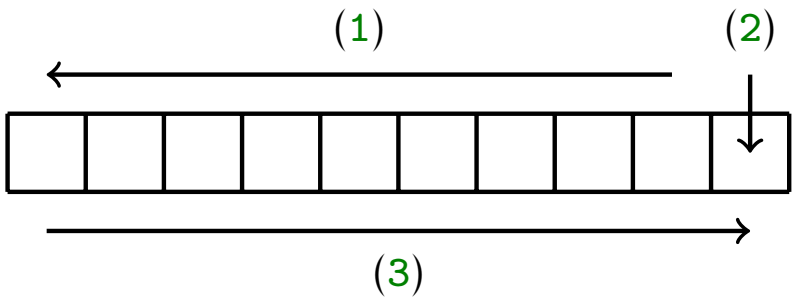


Circular Addressing

An FIR filter example.

Implementation using buffer shift.

```
float const pm coeff[10] = {...};
float          state[10] = {0};

float filter(float x) {
    int k;
    float y = 0;

    // Shift (1)
    for(k=0; k<9; ++k) {
        state[k] = state[k+1];
    }

    // Insert (2)
    state[9] = x;

    // Index (3)
    for(k=0; k<10; ++k) {
        y += state[k] * coeff[k];
    }

    return y;
}
```

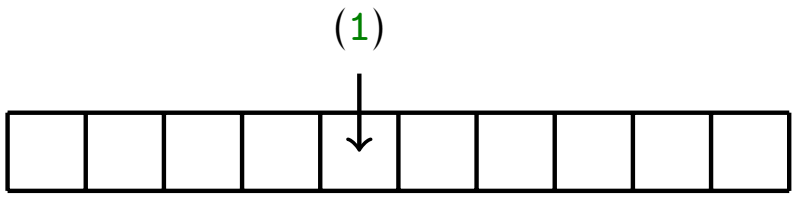


Circular Addressing

An FIR filter example.

Introducing circular addressing.

```
float const pm coeff[10] = {...};
float          state[10] = {0};
int            index = 0;

float filter(float x) {
    int k;
    float y = 0;

    // Insert (1)
    state[index] = x;

    // Advance (2)
    index = circindex(index, 1, 10);

    // Index (3)
    for(k=0; k<10; ++k) {
        y += state[index] * coeff[k];
        index = circindex(index, 1, 10);
    }

    return y;
}
```

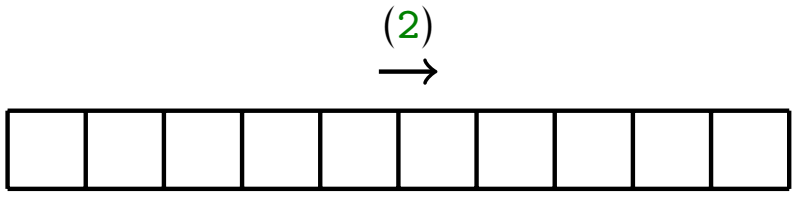


Circular Addressing

An FIR filter example.

Introducing circular addressing.

```
float const pm coeff[10] = {...};
float          state[10] = {0};
int            index = 0;

float filter(float x) {
    int k;
    float y = 0;

    // Insert (1)
    state[index] = x;

    // Advance (2)
    index = circindex(index, 1, 10);

    // Index (3)
    for(k=0; k<10; ++k) {
        y += state[index] * coeff[k];
        index = circindex(index, 1, 10);
    }

    return y;
}
```

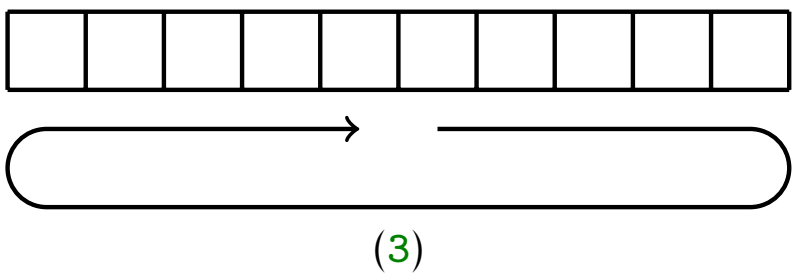


Circular Addressing

An FIR filter example.

Introducing circular addressing.

```
float const pm coeff[10] = {...};
float          state[10] = {0};
int            index = 0;

float filter(float x) {
    int k;
    float y = 0;

    // Insert (1)
    state[index] = x;

    // Advance (2)
    index = circindex(index, 1, 10);

    // Index (3)
    for(k=0; k<10; ++k) {
        y += state[index] * coeff[k];
        index = circindex(index, 1, 10);
    }

    return y;
}
```

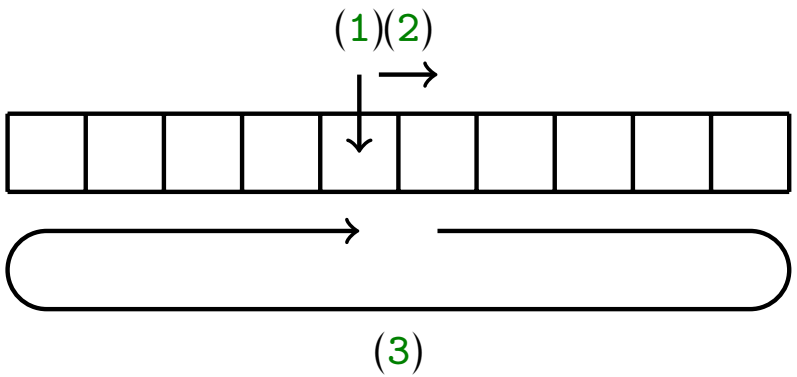


Circular Addressing

An FIR filter example.

Introducing circular addressing.

```
float const pm coeff[10] = {...};
float          state[10] = {0};
int            index = 0;

float filter(float x) {
    int k;
    float y = 0;

    // Insert (1)
    state[index] = x;

    // Advance (2)
    index = circindex(index, 1, 10);

    // Index (3)
    for(k=0; k<10; ++k) {
        y += state[index] * coeff[k];
        index = circindex(index, 1, 10);
    }

    return y;
}
```



Circular Addressing

An FIR filter example.

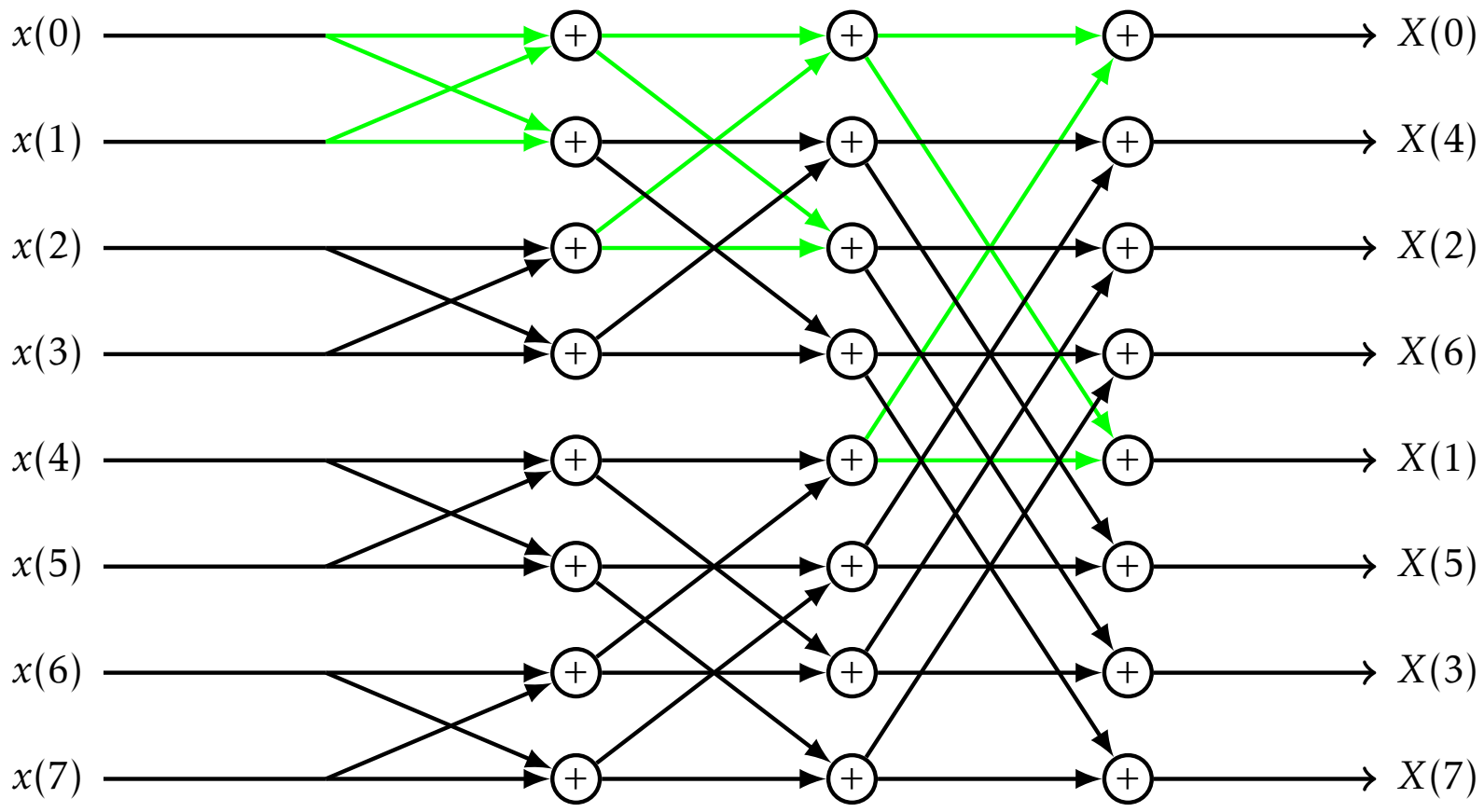
Introducing circular addressing.

```
float const pm coeff[10] = {...};
float          state[10] = {0};
int            index = 0;

float filter(float x) {
    int k;
    float y = 0;

    // Insert
    state[index] = x;

    // Advance
    index = circindex(index, 1, 10);

    // Index
    for(k=0; k<10; ++k) {
        y += state[(index+k) % 10] * coeff[k];
    }

    return y;
}
```

Bit-Reversed Addressing

Butterfly-structures and bit-reversed addressing.

Typical example is the fast Fourier transform.



Bit-Reversed Addressing

Butterfly-structures and bit-reversed addressing.

Index values are bit-reversed.

Base index	Bits	Bit reversed	Reversed index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Optimizing The FIR Filter

An FIR filter example.

Implement and FIR filter.

$$y(n) = \sum_{k=0}^{K-1} x(n-k)h(k)$$

Optimizations by the compiler in C and C++ when possible.

- Zero-overhead loops.
- Parallel memory reads.
- Parallel execution.
- Delayed branching.

Optimizing The FIR Filter

An FIR filter example.

Manual loop control and single execution.

```
// float conv(float *x, float *h, int K);
_conv:
    entry;

    f0  = 0;            // return value in r0
    r1  = 0;            // loop counter
    i4  = r4;           // first parameter x in r4
    i12 = r8;           // second parameter h in r8
loop:
    f4 = dm(i4, 1);     // read x
    f3 = dm(i12, 1);    // read h
    f4 = f4 * f3;       // multiply
    f0 = f0 + f4;       // accumulate
    r1 = r1 + 1;        // advance loop counter
    comp(r1, r12);      // third parameter K in r12
    if lt jump loop;

    exit;
._conv.end:
```

Optimizing The FIR Filter

An FIR filter example.

Zero-overhead loops.

```
// float conv(float *x, float *h, int K);
_conv:
    entry;

    f0  = 0;
    i4  = r4;
    i12 = r8;

    lcntr = r12, do (loop-1) until lce;
    f4 = dm(i4, 1);
    f3 = dm(i12, 1);
    f4 = f4 * f3;
    f0 = f0 + f4;
loop:

    exit;
._conv.end:
```

Optimizing The FIR Filter

An FIR filter example.

Parallel memory reads.

```
// float conv(float *x, float const pm *h, int K);
_conv:
    entry;

    f0  = 0;
    i4  = r4;
    i12 = r8;

    lcntr = r12, do (loop-1) until lce;
    f4 = dm(i4, 1),     // read both x and h in parallel
    f3 = pm(i12, 1);    // requires h to be stored in pm-memory
    f4 = f4 * f3;
    f0 = f0 + f4;
loop:

    exit;
._conv.end:
```

Optimizing The FIR Filter

An FIR filter example.

Parallel execution and loop rotation.

```
// float conv(float *x, float const pm *h, int K);
_conv:
    entry;

    r1  = r12 - 1;
    i4  = r4;
    i12 = r8;
    r8  = 0;                        // (1)

    r12 = r12 - r12,                // (2)
    f0  = dm(i4, m6),               // (3)
    f4  = pm(i12, m14);             // (4)

    lcntr = r1, do (loop-1) until lce;
    f12 = f0 * f4,                  // (5)
    f8  = f8 + f12,                 // (6)
    f0  = dm(i4, m6),               // (7)
    f4  = pm(i12, m14);             // (8)
loop:

    f12 = f0 * f4,                  // (9)
    f8  = f8 + f12;                 // (10)
    f0  = f8 + f12;                 // (11)

    exit;
._conv.end:
```

Optimizing The FIR Filter

Loop rotation.

The effective order of the parallel instruction is *add-mult-read*.

- Reset accumulator (1) and product (2) for first iteration.
- Perform initial read (3-4).
- Loop one less iteration:
  - Accumulate previous product (6).
  - Multiply current values (5).
  - Read next values (7-8).
- Multiply last values (9) and add previous product (10).
- Accumulate last product (11).

Optimizing The FIR Filter

An FIR filter example.

The original order of loop execution is *read-mult-add*.

	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7
Read	R(0)			R(1)			R(2)	
Multiply		M(0)			M(1)			M(2)
Accumulate			A(0)			A(1)		

Optimizing The FIR Filter

Order of operations

- R(n) before R(n+1)
- M(n) before M(n+1)
- A(n) before A(n+1)
- R(n) before M(n)
- M(n) before A(n)

	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7
Read	R(0)	R(1)	R(2)	R(3)	R(4)	R(5)		
Multiply		M(0)	M(1)	M(2)	M(3)	M(4)	M(5)	
Accumulate			A(0)	A(1)	A(2)	A(3)	A(4)	A(5)

Optimizing The FIR Filter

Delayed branching.

The ADSP-21262 has a three-cycle instruction pipeline.

- A jump forces the instruction pipeline to flush.
- A two-cycle stall is required to refill the pipeline.

```
loop:
    f4 = dm(i4, 1);
    f3 = pm(i12, 1);
    f4 = f4 * f3;
    f0 = f0 + f4;
    r1 = r1 + 1;
    comp(r1, r12);
    if lt jump loop;
```

Optimizing The FIR Filter

Delayed branching.

A delayed branch does not flush the instruction pipeline.

- Executes two additional instructions before jumping.
- Eliminates the two-cycle stall.

```
loop:
    f4 = dm(i4, 1);
    f3 = pm(i12, 1);
    r1 = r1 + 1;
    comp(r1, r12);
    if lt jump loop (db);
    f4 = f4 * f3;
    f0 = f0 + f4;
```