ETIN80 — Algorithms in Signal Processors

Signal Processor Details

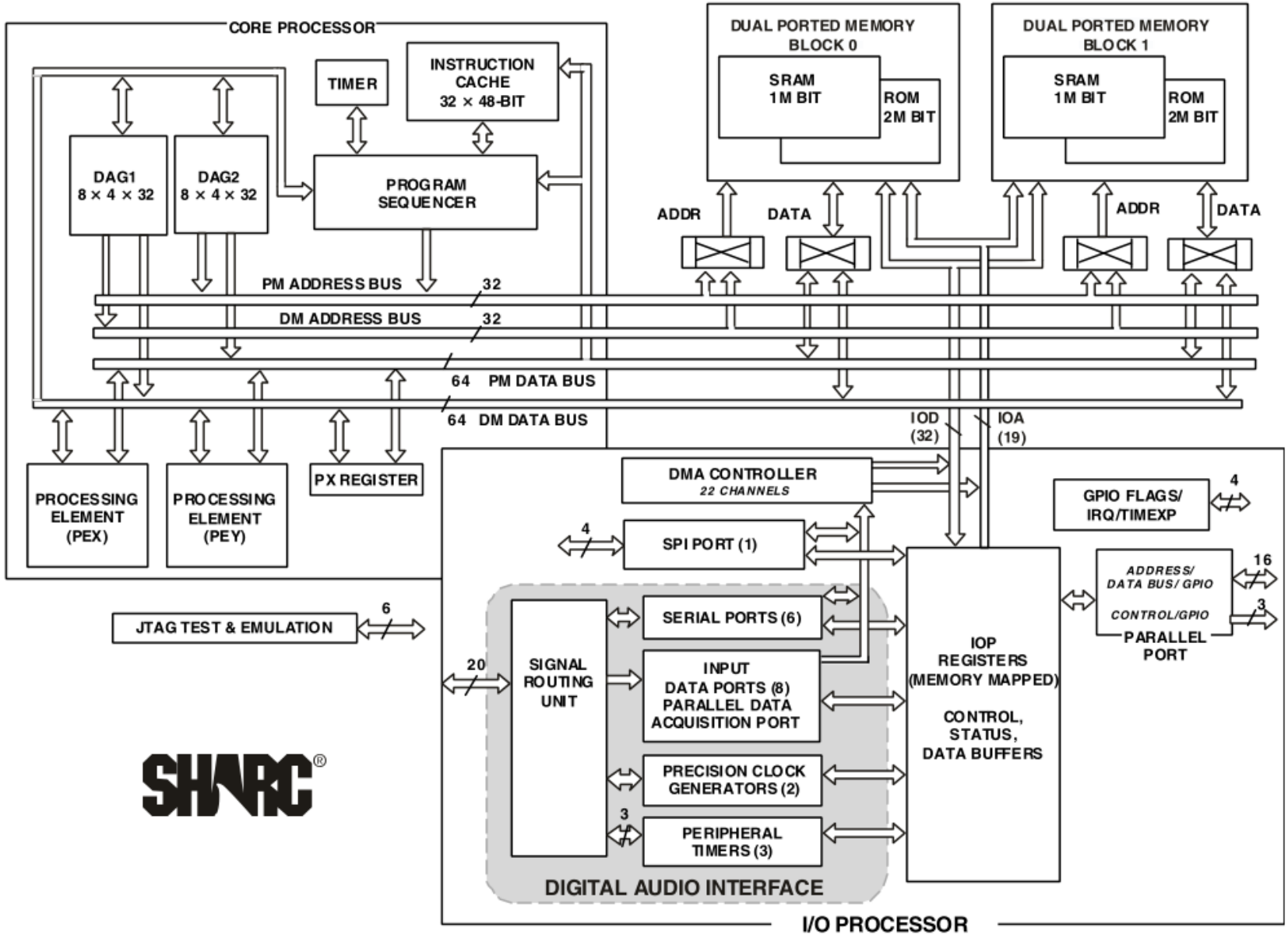
Tekn.Dr. Mikael Swartling

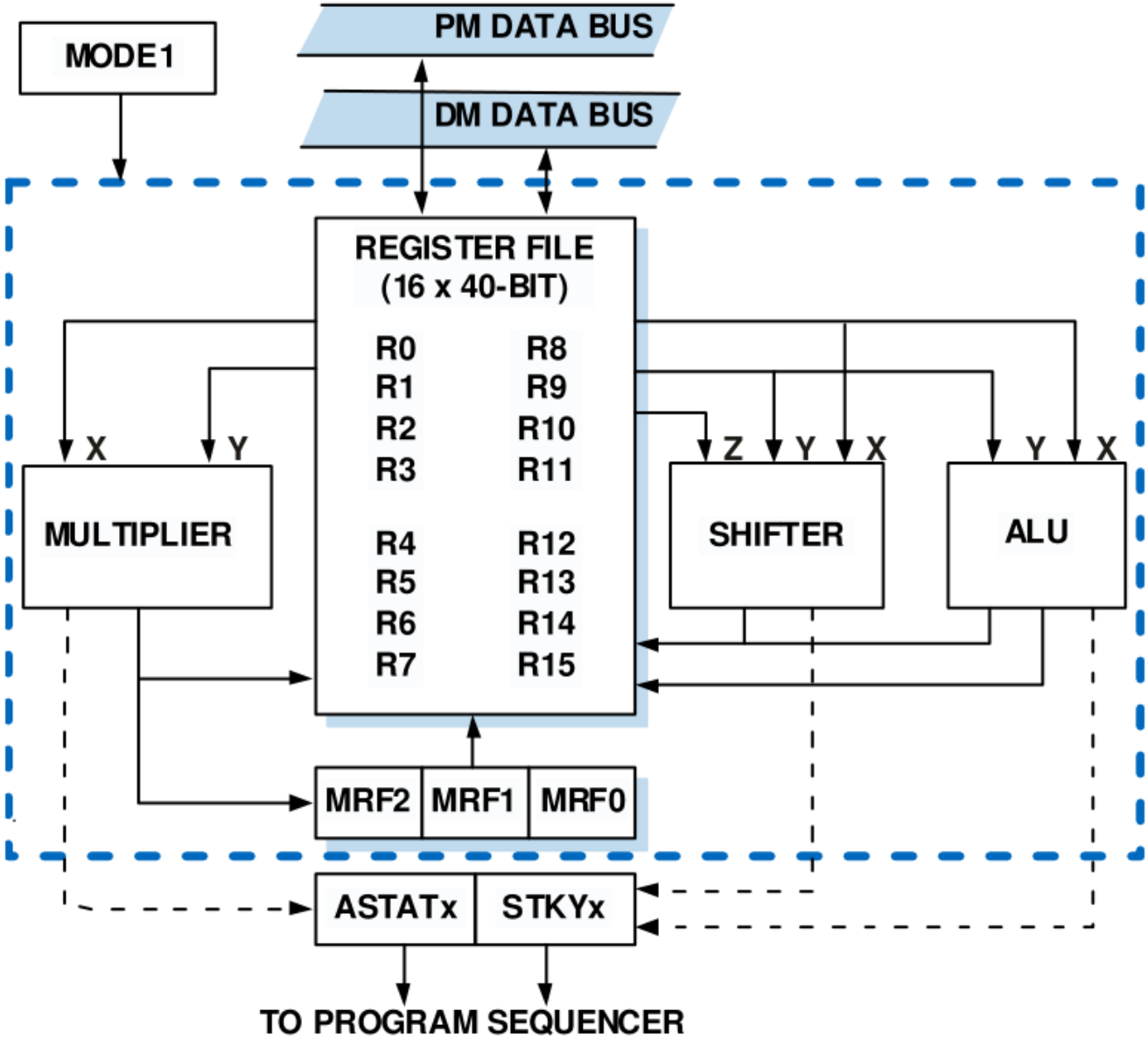
Lund Institute of Technology
Department of Electrical and Information Technology

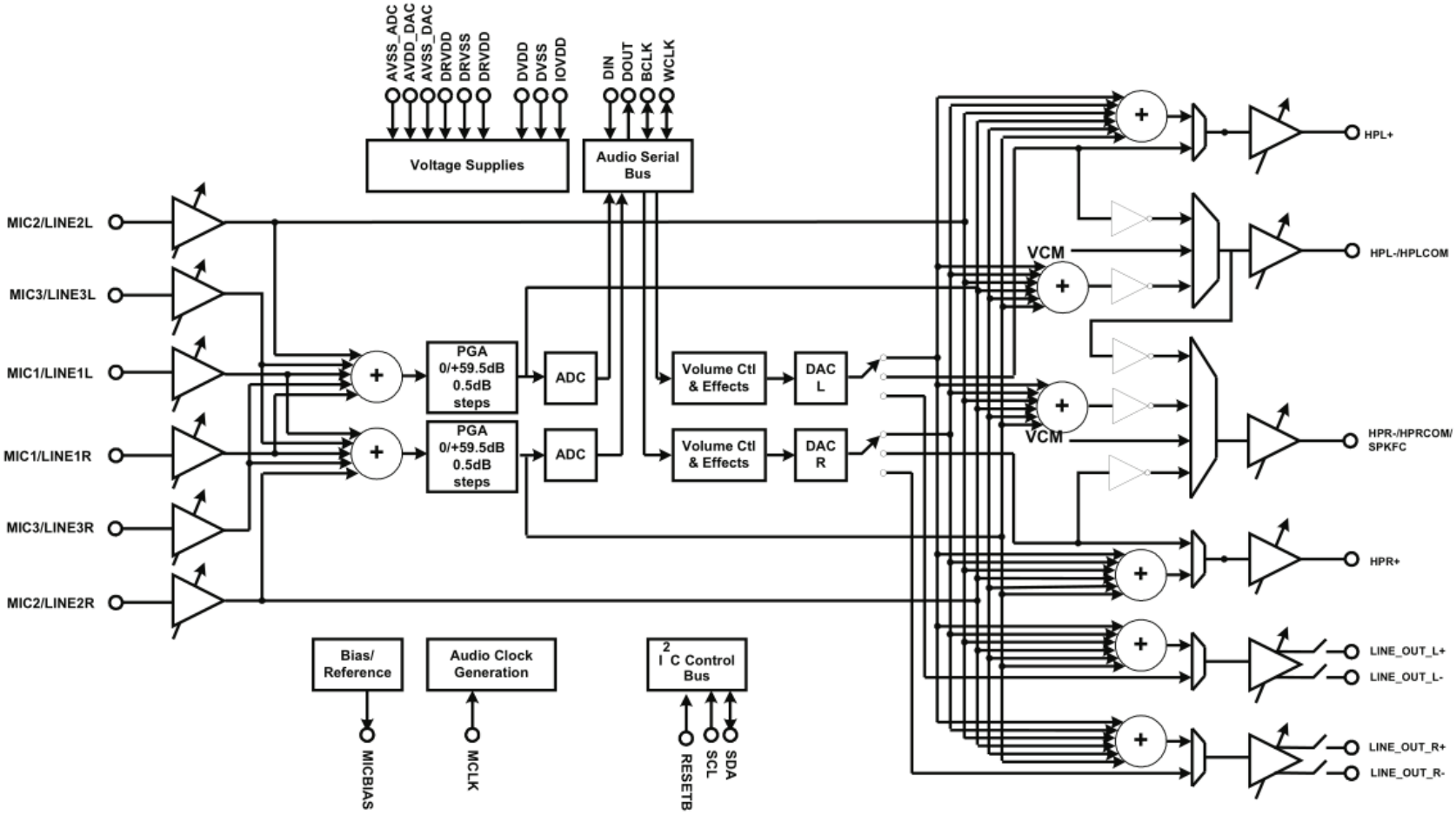
January 27, 2014

Hardware

- Analog Devices ADSP-21262 DSP.
  - 200 MHz maximum core
  - 2 MiB memory
  - 4 MiB non-volatile memory
  - 32 bit computational units
  - integer, fixed point and floating point
  - dual processing units
- Texas Instruments TLV320AIC32 audio codec.
  - 32 bit stereo codec
  - 8 kHz to 96 kHz sampling rate
  - line and microphone input
  - line and power output
- Four semi-independent audio codecs.
  - 8 input channels.
  - 8 output channels.


CORE PROCESSOR
TIMER
INSTRUCTION CACHE 32 × 48-BIT
DAG1 8 × 4 × 32
DAG2 8 × 4 × 32
PROGRAM SEQUENCER
DUAL PORTED MEMORY BLOCK 0
SRAM 1M BIT
ROM 2M BIT
DUAL PORTED MEMORY BLOCK 1
SRAM 1M BIT
ROM 2M BIT
ADDR
DATA
ADDR
DATA
PM ADDRESS BUS 32
DM ADDRESS BUS 32
64 PM DATA BUS
64 DM DATA BUS
IOD (32)
IOA (19)
PROCESSING ELEMENT (PEX)
PROCESSING ELEMENT (PEY)
PX REGISTER
DMA CONTROLLER 22 CHANNELS
GPIO FLAGS/ IRQ/TIMEXP
4
4
SPI PORT (1)
ADDRESS/ DATA BUS/ GPIO
16
CONTROL/GPIO
3
PARALLEL PORT
JTAG TEST & EMULATION
6
SERIAL PORTS (6)
20
SIGNAL ROUTING UNIT
INPUT DATA PORTS (8) PARALLEL DATA ACQUISITION PORT
IOP REGISTERS (MEMORY MAPPED)
CONTROL, STATUS, DATA BUFFERS
PRECISION CLOCK GENERATORS (2)
3
PERIPHERAL TIMERS (3)
DIGITAL AUDIO INTERFACE
I/O PROCESSOR
SHARC®

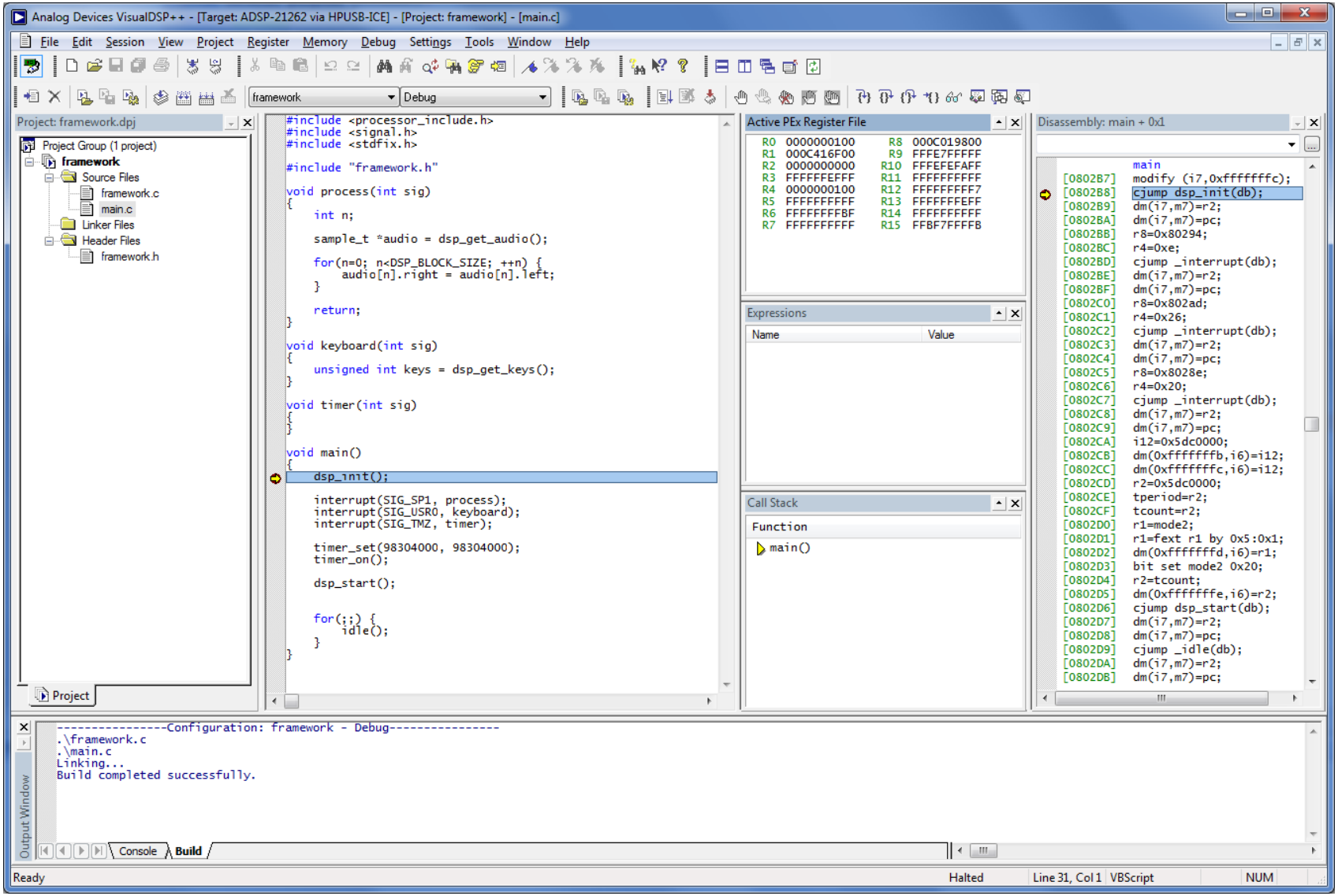

MODE1
PM DATA BUS
DM DATA BUS
REGISTER FILE
(16 x 40-BIT)
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
X
Y
MULTIPLIER
Z
Y
X
SHIFTER
Y
X
ALU
MRF2
MRF1
MRF0
ASTATx
STKYx
TO PROGRAM SEQUENCER


AVSS_ADC
AVDD_DAC
AVSS_DAC
DRVDD
DRVSS
DRVDD
DVDD
DVSS
IOVDD
DIN
DOUT
BCLK
WCLK
Voltage Supplies
Audio Serial Bus
MIC2/LINE2L
MIC3/LINE3L
MIC1/LINE1L
MIC1/LINE1R
MIC3/LINE3R
MIC2/LINE2R
PGA 0/+59.5dB 0.5dB steps
PGA 0/+59.5dB 0.5dB steps
ADC
ADC
Volume Ctl & Effects
Volume Ctl & Effects
DAC L
DAC R
VCM
VCM
HPL+
HPL-/HPLCOM
HPR-/HPRCOM/ SPKFC
HPR+
LINE_OUT_L+
LINE_OUT_L-
LINE_OUT_R+
LINE_OUT_R-
Bias/ Reference
Audio Clock Generation
I2C Control Bus
MICBIAS
MCLK
RESETB
SCL
SDA

Integrated Development Environment

- Visual DSP++ 5.0.
  - workspace and project manager
  - optimizing compiler for C, C++ and assembly
  - simulator and in-circuit emulator
  - automation scripting
- Extensive debugger.
  - expressions
  - register views
  - graphs and images
- Run-time library.
  - standard C library
  - standard C++ library
  - signal processing library
  - file and console I/O

Integrated Development Environment



Standard Library

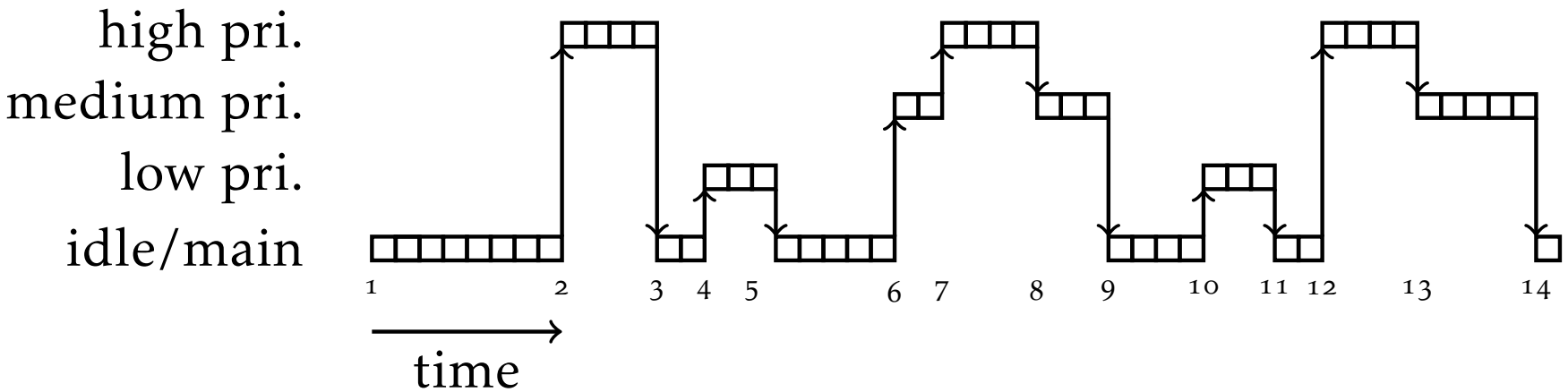
- Complete C and C++ run-time libraries.
- Library for DSP primitives.
  - matrix and vector functions
  - real and complex data
  - filter functions
  - Fourier transforms

Interrupts

- A signal from a hardware of software indicating an event that needs immediate attention.
- Interrupt-driven design.
  - cpu informs the program when something happens
- Poll-driven design.
  - program asks the cpu if something has happened
- Interrupt-driven design is preferred.
- Poll-driven design is sometimes necessary.

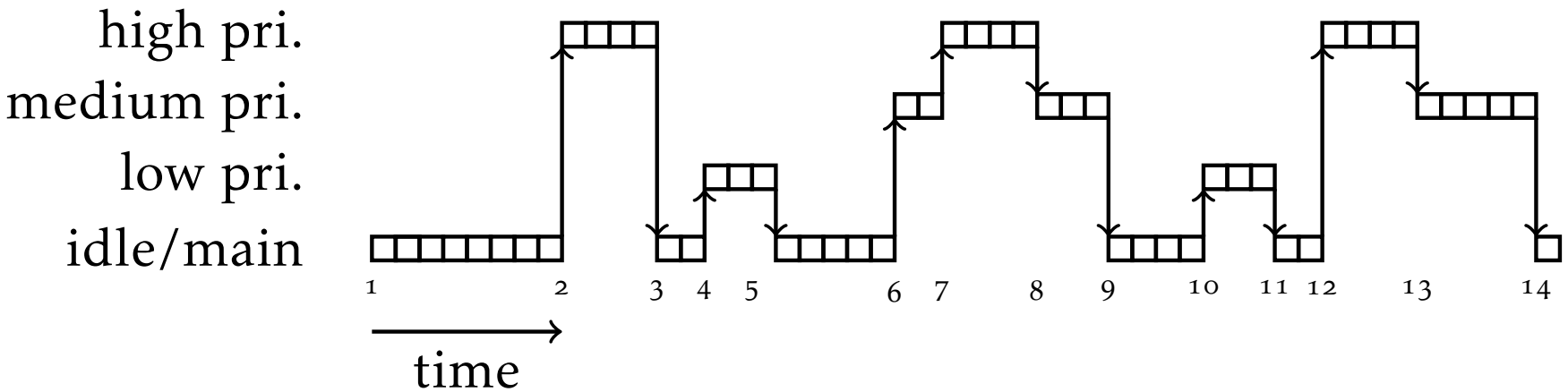
Interrupts

- Program sequence during interrupts.
  - 1 program execution starts as normal in `main()`
  - 2,3 HP breaks and resumes `main()`
  - 4,5 LP breaks and resumes `main()`
- Interrupt sequencing is automatic.



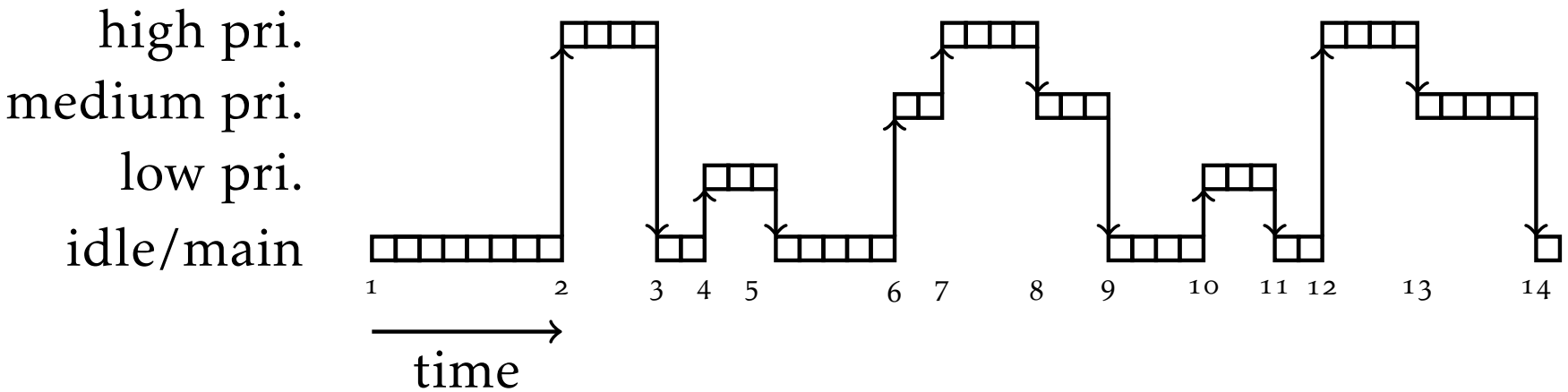
Interrupts

- Program sequence during interrupts.
  - 6,9 MP breaks and resumes `main()`
  - 7,8 HP breaks and resumes MP
  - 10,11 MP breaks and resumes `main()`
- Interrupt sequencing is automatic.



Interrupts

- Program sequence during interrupts.
  - 12 HP breaks `main()`
  - 13 MP happens during HP, control is returned to MP
  - 14 MP resumes `main()`
- Interrupt sequencing is automatic.



Inter-Frame Filtering States

- Implement $y(n) = \sum_{k=0}^{K-1} x(n-k)h(k)$
- Up to $K-1$ previous samples has to be preserved.
- Any persistent state has to be preserved.

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

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

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

    state[9] = x;

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

    return y;
}
```

Information Feedback

- Information feedback is a problem on a device without a display.
- For run-time feedback:
  - audible cues
- For debugging feedback:
  - console I/O
  - file I/O
- Use the in-circuit debugger for proper debugging.

Keypad Bouncing

- The keypad is a mechanical switch.
- Multiple triggers as the switch open or close.
- Handle bounces:
  - delayed sampling using a timer
  - ignore repetitive triggers
- Ignore bounces:
  - assign single and discrete events per key

Data Types

- Almost ever data type is 32 bits.

  `long long` and `long double` are 64 bit and emulated
- You have access to:
  - integer types
  - floating point types
  - fractional types
- Fractional types are defined in `stdfix.h`
  - type `fract` for a fractional value
  - type `accum` for a fractional accumulator
- Fractional values represent values from -1 to $1 - 2^{-31}$.

Program Memory for Constant Buffers

- The ADSP-21262 has two memory banks.
- Code uses the PM bank.
- Data uses the DM bank by default.
- Put constant data in the PM bank.

```
float const pm coeff[10] = {...};   // PM bank
float          state[10] = {0};     // DM bank by default
```

- Example: filter coefficients that can be read in parallel with sample data.

Circular and Bit-Reversed Addressing

- Implement $y(n) = \sum_{k=0}^{K-1} x(n-k)h(k)$
- Naive implementation performs a buffer shift.
- Circular addressing is automatic.

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

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

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

    state[9] = x;

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

    return y;
}
```

Circular and Bit-Reversed Addressing

- Implement $y(n) = \sum_{k=0}^{K-1} x(n-k)h(k)$
- Aware implementation uses circular addressing.
- Circular addressing is automatic.

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

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

    state[current] = x;
    current = circindex(current, 1, 10);

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

    return y;
}
```

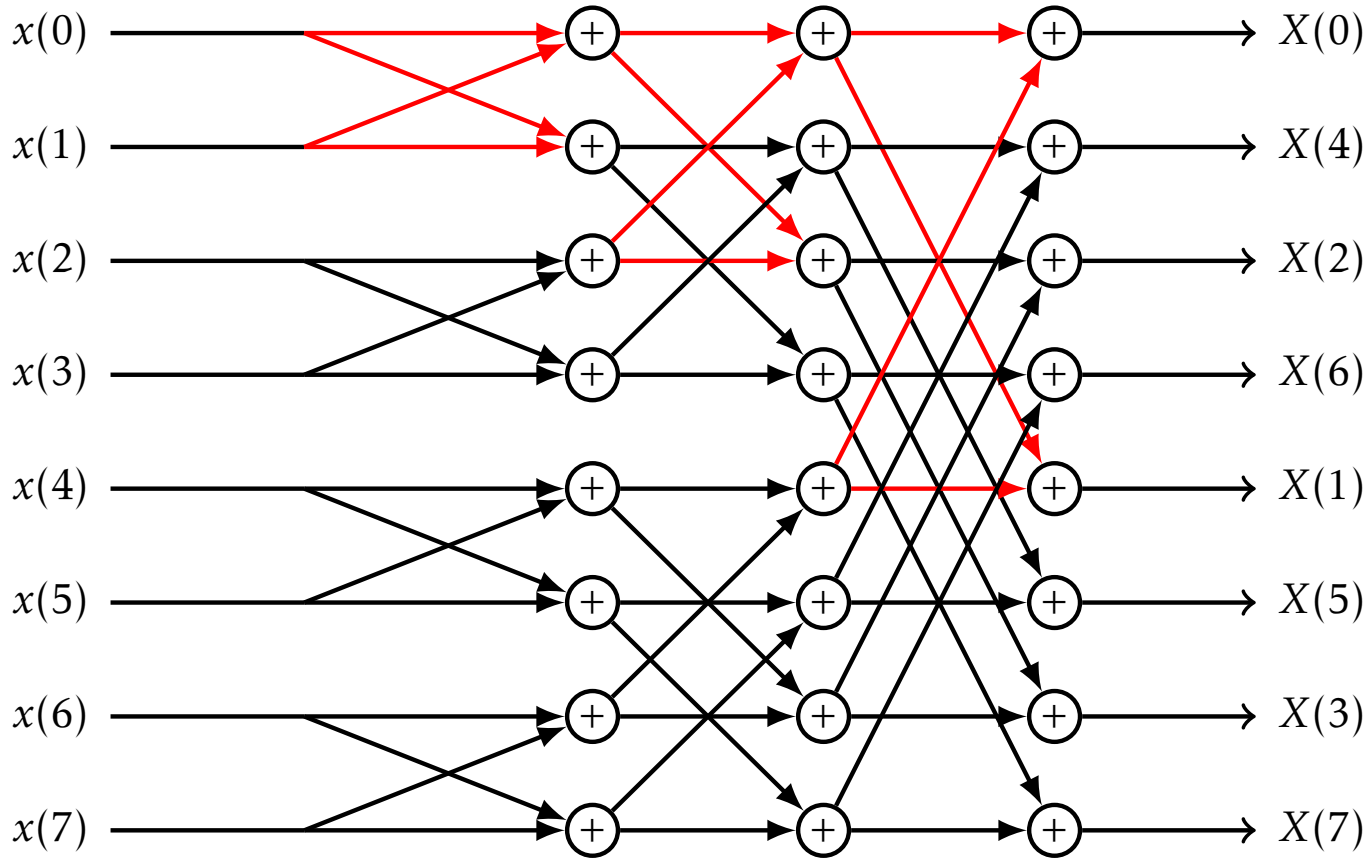
Circular and Bit-Reversed Addressing

- Implement $y(n) = \sum_{k=0}^{K-1} x(n-k)h(k)$
- Aware implementation uses circular addressing.
- Circular addressing is automatic.

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

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

    state[current] = x;
    current = circindex(current, 1, 10);

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

    return y;
}
```

Circular and Bit-Reversed Addressing

- Butterfly-filters end up with bit-reversed addressing.
- Typical example is the Fourier transform.



Circular and Bit-Reversed Addressing

- Index values are bit-reversed.
- Bit-reversal is automatic.

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

Filtering Case Study

- Implement $y(n) = \sum_{k=0}^{K-1} x(n-k)h(k)$ in assembly.
- Separate data and program memory.
- Zero-Overhead Loops
- Parallel execution.
- Delayed branching.
- Automatic by the compiler in C and C++ when possible.
- Calling convention:
  - return value in register r0
  - first parameter in register r4
  - second parameter in register r8
  - third parameter in register r12

Zero-Overhead Loops

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

    f0 = 0;      // accumulator
    r1 = 0;      // loop counter
    i4 = r4;     // x
    i12 = r8;    // h
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;

    exit;
._conv.end:
```

- naive implementation

Zero-Overhead Loops

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

    f0 = 0;      // accumulator
    i4 = r4;     // x
    i12 = r8;    // h
    lcntr = r12, do (loop-1) until lce;
        f4 = dm(i4, 1);
        f3 = pm(i12, 1);
        f4 = f4 * f3;
        f0 = f0 + f4;
loop:

    exit;
._conv.end:
```

- zero-overhead loops

Parallel Execution

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

    f0 = 0;      // accumulator
    i4 = r4;     // x
    i12 = r8;    // h
    lcntr = r12, do (loop-1) until lce;
        f4 = dm(i4, 1);
        f3 = pm(i12, 1);
        f4 = f4 * f3;
        f0 = f0 + f4;
loop:

    exit;
._conv.end:
```

- naive memory transfers
- single operation per cycle
- full parallelization requires loop-rotation

Parallel Execution

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

    r1 = r12-1;
    i4 = r4;
    i12 = r8;
    r8 = 0;

    r12 = r12-r12,
    f0 = dm(i4, m6), f4 = pm(i12, m14);

    lcntr = r1, do (loop1-1) until lce;
        f12 = f0 * f4, f8 = f8 + f12,
        f0 = dm(i4, m6), f4 = pm(i12, m14);
loop1:
    f12 = f0 * f4, f8  = f8 + f12;
    f0 = f8+f12;

    exit;
._conv.end:
```

- dm and pm in parallel
- loop has been rotated:
  - initial read data before loop
  - loop one less interation
  - final operations after loop
- data and computation in parallel

Delayed Branching

- `exit` is a macro expanding to:

```
i12=dm(m7,i6);
jump (m14,i12) (db);
rframe;
nop;
```

- The ADSP-21262 has a three-cycle instruction pipeline.
  - jump forces the instruction pipeline to flush
  - two-cycle stall to refill the pipeline
- A delayed branch does not flush the instruction pipeline.
  - executes two additional instructions before jumping
  - eliminates the two-cycle stall