Digital Processing Laboratory
Laboratorio de Procesamiento Digital

Real Time Digital Signal Processing

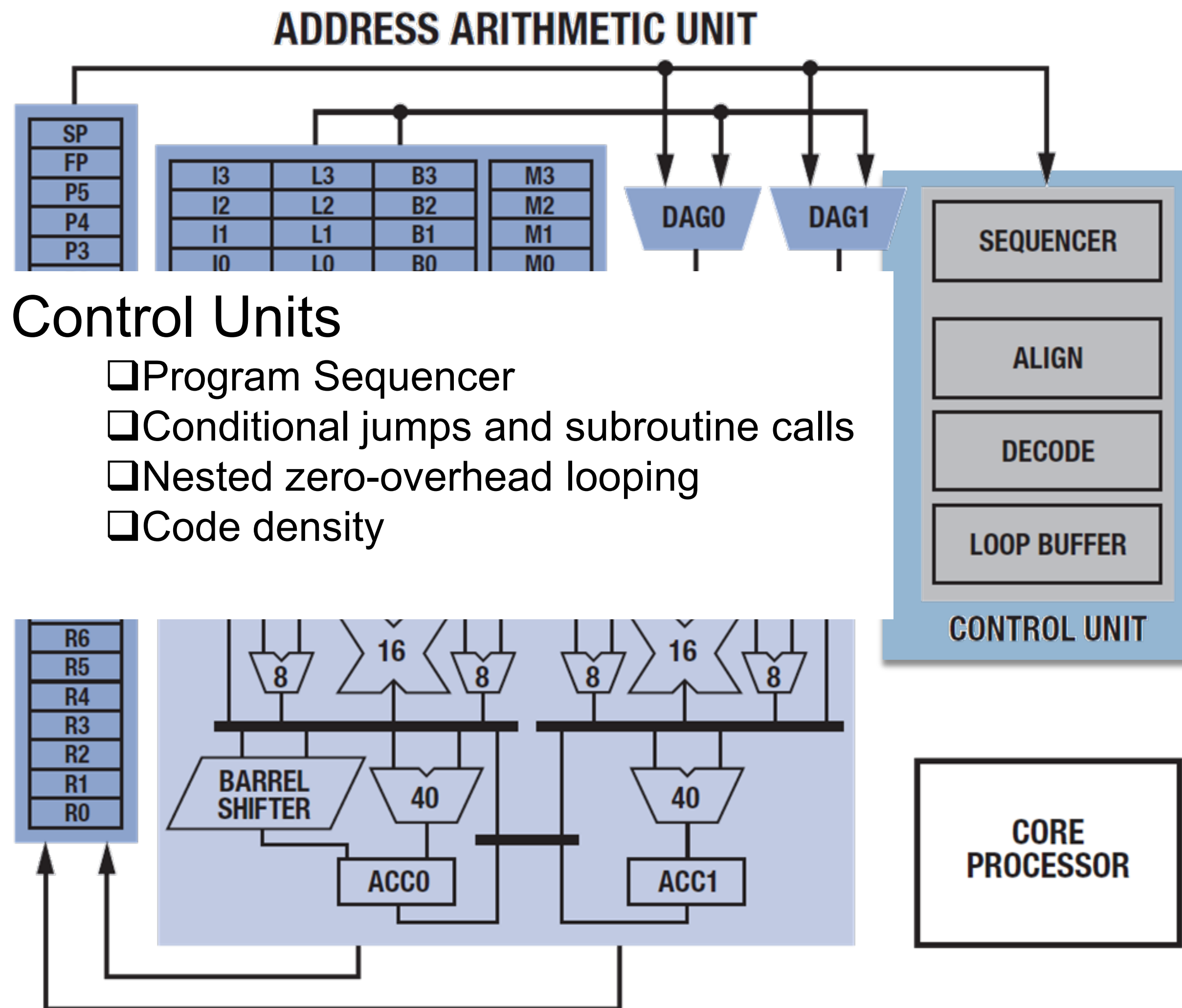
UTN-FRBA
2011

Eng. Julian Bruno

Architecture

Introduction to the Blackfin Processor

Core Architecture – BF53X



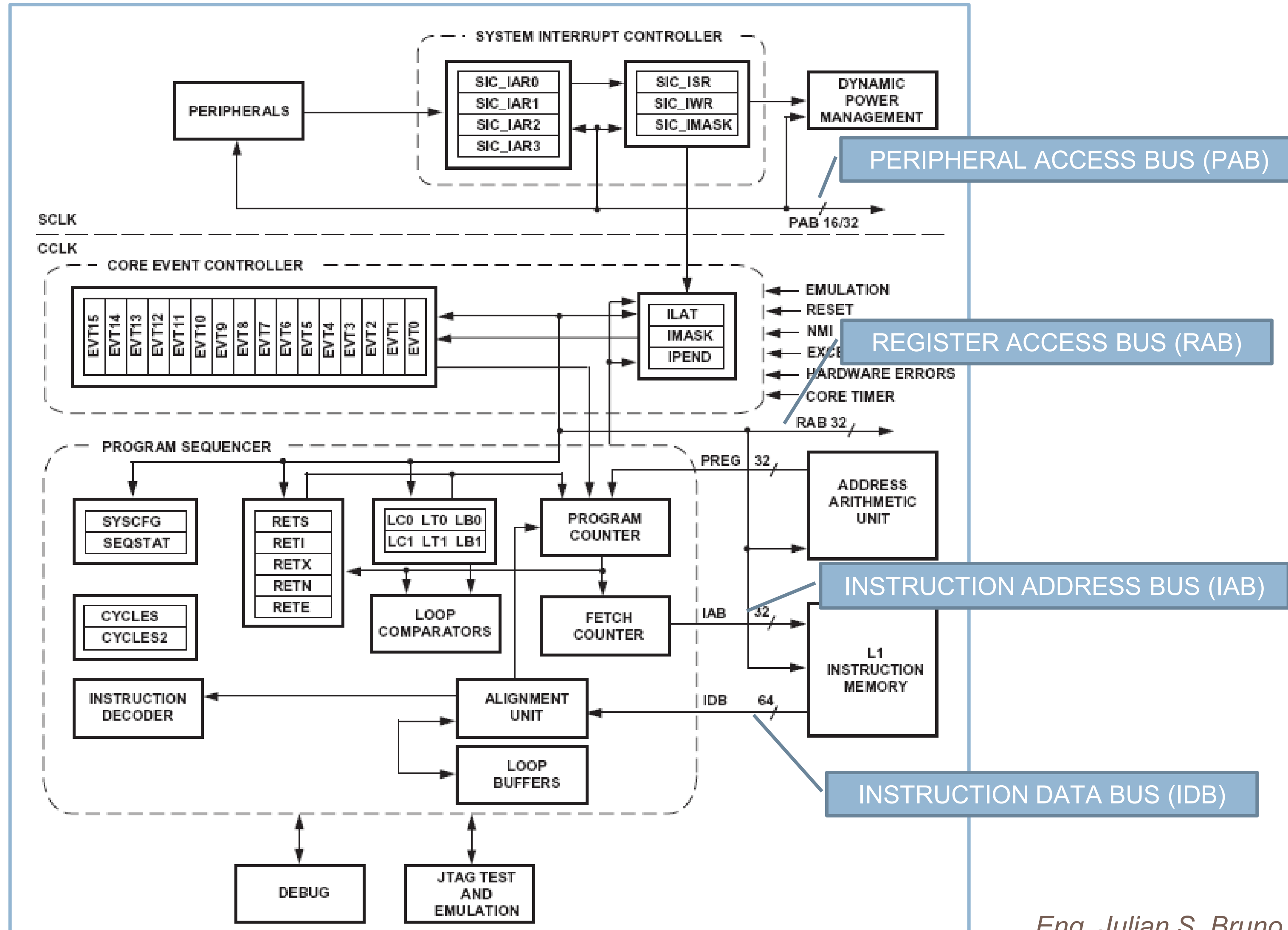
Control Units

- Program Sequencer
- Conditional jumps and subroutine calls
- Nested zero-overhead looping
- Code density

Control Units – BF53X

- Program Sequencing
- Interrupt Processing modules
- Program Flow Control
- External Event Management
- Specific interrupt sources

Program Sequencing and Interrupt Processing Block Diagram

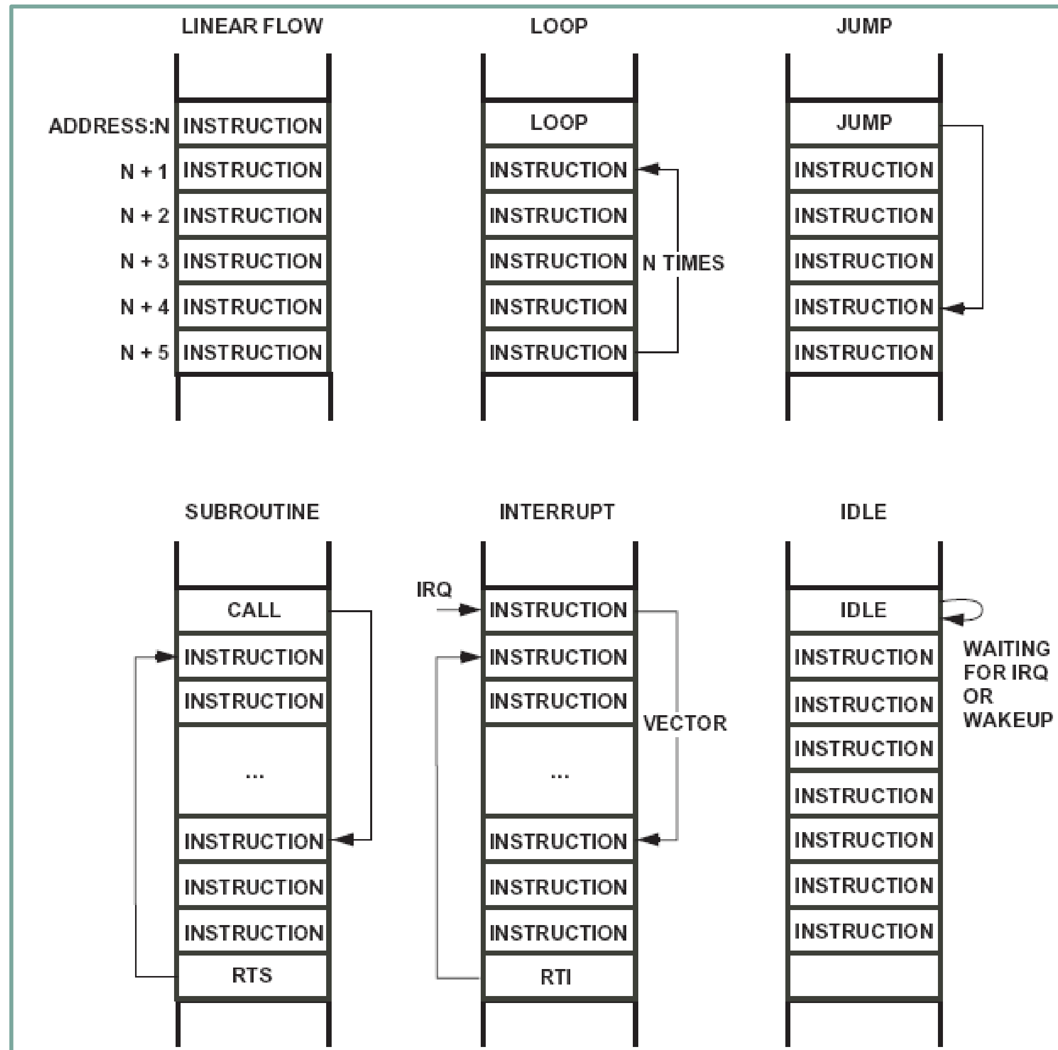


Eng. Julian S. Bruno

Program Flow

- Program flow in the chip is mostly linear
- The processor executing program instructions sequentially
- The linear flow varies occasionally when the program uses nonsequential program structures, such as:
  - Loops.
  - Subroutines.
  - Jumps.
  - Interrupts and Exceptions.
  - Idle.

Program Flow Variations



Instruction Pipeline

- The program sequencer determines the next instruction address by examining both the current instruction being executed and the current state of the processor.
- The processor has a ***ten-stage*** instruction pipeline
- The instructions can be 16, 32, or 64 bits wide
- Multi-issue ***instructions*** are ***64 bits in length*** and consist of ***one 32-bit instruction*** and ***two 16-bit instructions***

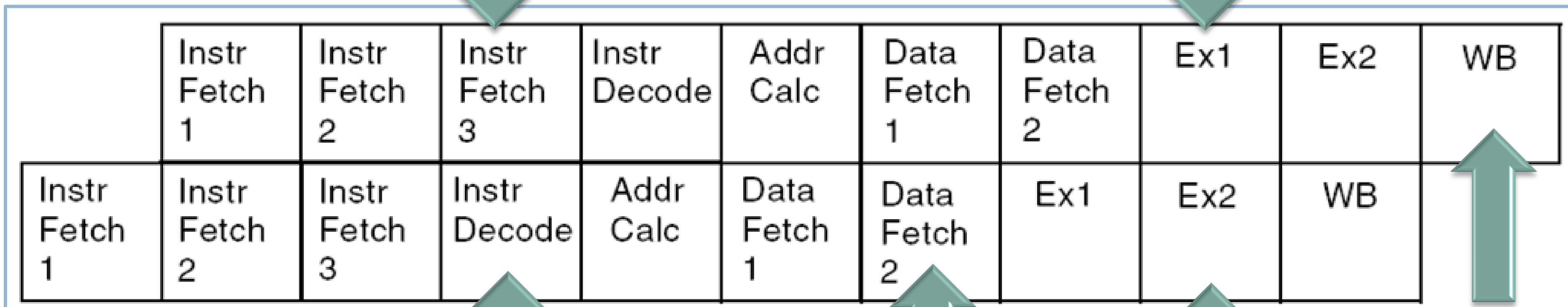
Instruction Pipeline (II)

Pipeline Stage	Description
Instruction Fetch 1 (IF1)	Issue instruction address to IAB bus, start compare tag of instruction cache
Instruction Fetch 2 (IF2)	Wait for instruction data
Instruction Fetch 3 (IF3)	Read from IDB bus and align instruction
Instruction Decode (DEC)	Decode instructions
Address Calculation (AC)	Calculation of data addresses and branch target address
Data Fetch 1 (DF1)	Issue data address to DA0 and DA1 bus, start compare tag of data cache
Data Fetch 2 (DF2)	Read register files
Execute 1 (EX1)	Read data from LD0 and LD1 bus, start multiply and video instructions
Execute 2 (EX2)	Execute/Complete instructions (shift, add, logic, etc.)
Write Back (WB)	Writes back to register files, SD bus, and pointer updates (also referred to as the “commit” stage)

Instruction Pipeline (III)

The Instruction Alignment Unit returns instructions and their width information

The multipliers and the video units are active



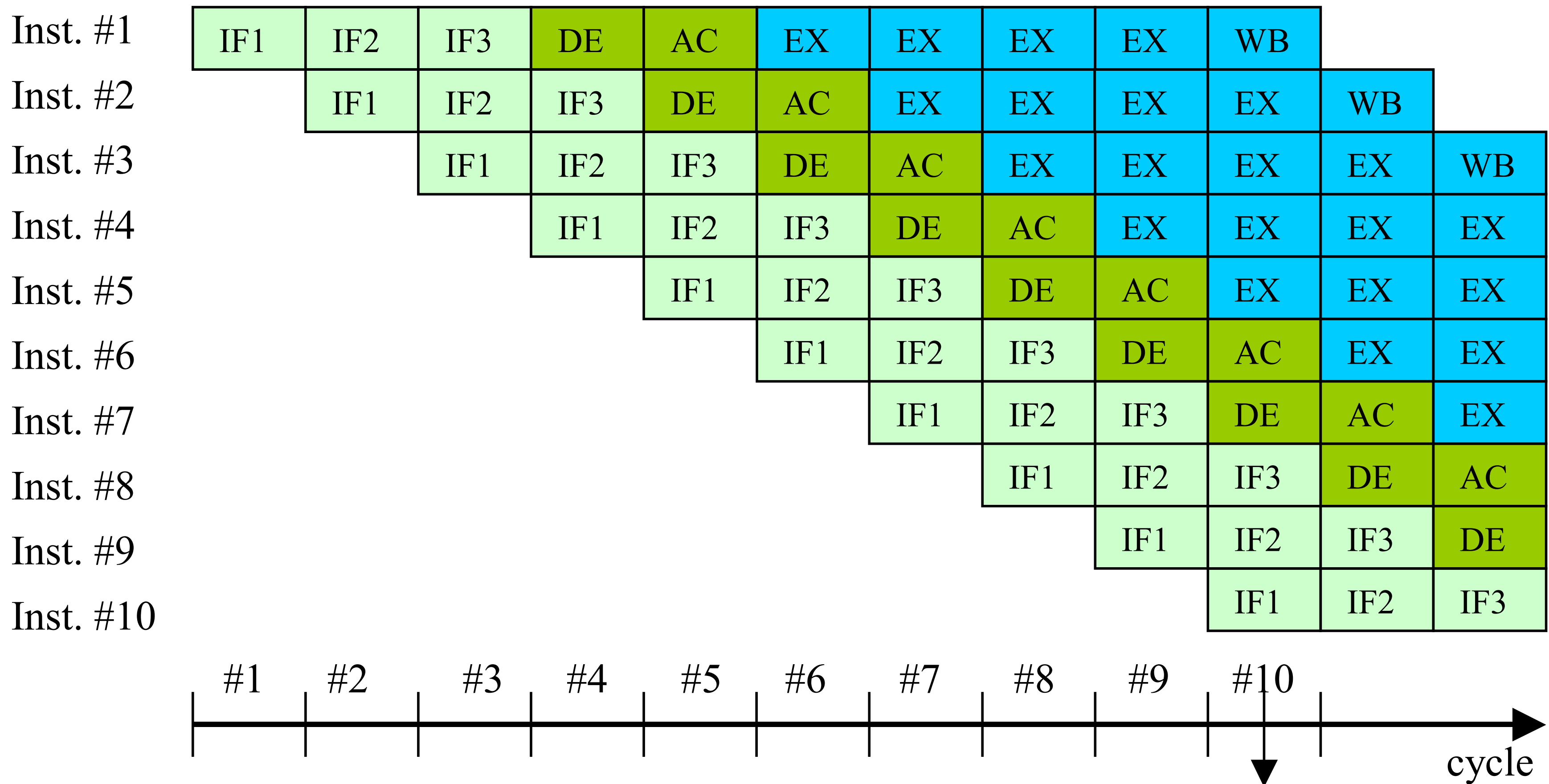
Register file reads

Register file writes

The sequencer decodes and distributes instruction data to the appropriate locations such as the register file and data memory

The ALUs and shifter are active. At the end, the accumulators are written

Instruction Pipeline (IV)



Branches

- A branch occurs when a JUMP or CALL instruction ***begins execution*** at a new location other than the next sequential address.
- The are five types of return instructions: RTS, RTI, RTX, RTN and RTE.
- Each return type has its own register for holding the return address. RETS, RETI, RETX, RETN and RETE.

Branches (II)

- The program sequencer can evaluate the CC status bit to decide whether to execute a branch.
- Conditional JUMP instructions use static branch prediction to reduce the branch latency caused by the length of the pipeline.
- Branches can be direct or indirect.
  - A direct branch address is determined solely by the instruction word. JUMP 0x30;
  - An indirect branch gets its address from the contents of a DAG register. JUMP(P3);

Branches (III)

- **Direct Short and Long Jumps**
- Direct Call
- Indirect Branch and Call
- PC-Relative Indirect Branch and Call
- Subroutines
- Condition Code Flag
- Branch Prediction

Direct Short and Long Jumps

- The target of the branch is:

  PC-relative address + Offset

- Short jump:
  - The PC-relative offset is a 13-bit immediate value that must be a multiple of two
  - Dynamic range of –4096 to +4094 bytes.
  - JUMP.S 0xnnnn
- Long jump:
  - The PC-relative offset for is a 25-bit immediate value that must also be a multiple of two.
  - Dynamic range of –16,777,216 to +16,777,214 bytes.
  - JUMP.L 0xnnnnnnn

Branches (III)

- Direct Short and Long Jumps
- **Direct Call**
- Indirect Branch and Call
- PC-Relative Indirect Branch and Call
- Subroutines
- Condition Code Flag
- Branch Prediction

Direct Call

- CALL instruction copies the address of the instruction which would have executed next into the RETS register.
- The direct CALL instruction has a 25-bit, PC-relative offset that must be a multiple of two.
- The 25-bit value gives an effective dynamic range of –16,777,216 to +16,777,214 bytes.
- A direct CALL instruction is always a 4-byte instruction.

Branches (III)

- Direct Short and Long Jumps
- Direct Call
- **Indirect Branch and Call**
- PC-Relative Indirect Branch and Call
- Subroutines
- Condition Code Flag
- Branch Prediction

Indirect Branch and Call

- The indirect instructions get their destination address from a data address generator (DAG) P-register.
- For the CALL instruction, the RETS register is loaded with the address of the instruction which would have executed.

```
P4.H = HI(mytarget);
P4.L = LO(mytarget);
JUMP (P4);
…….
mytarget:
        /* continue here */
```

Branches (III)

- Direct Short and Long Jumps
- Direct Call
- Indirect Branch and Call
- **PC-Relative Indirect Branch and Call**
- Subroutines
- Condition Code Flag
- Branch Prediction

PC-Relative Indirect Branch and Call

- The PC-relative indirect JUMP and CALL instructions use the contents of a P-register as an offset to the branch target.
- For the CALL instruction, the RETS register is loaded with the address of the instruction which would have executed next

```
JUMP (PC + P3) ;
CALL (PC + P0) ;
```

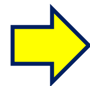
Branches (III)

- Direct Short and Long Jumps
- Direct Call
- Indirect Branch and Call
- PC-Relative Indirect Branch and Call
- **Subroutines**
- Condition Code Flag
- Branch Prediction

Subroutines

- Subroutines are code sequences that are invoked by a CALL instruction.

“leaf functions”



RETS = This address

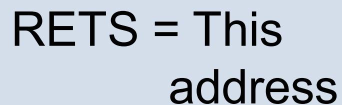
```
/* parent function */
R0 = 0x1234 (Z);        /* pass a parameter */
CALL myfunction;
/* continue here after the call */
[P0] = R0;              /* save return value */
JUMP somewhereelse;
```

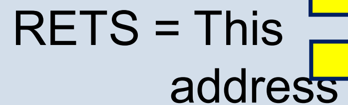
Subroutines - Leaf functions

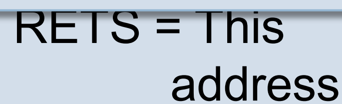
```
myfunction:                 /* subroutine label */

  [--SP] = (R7:7, P5:5);    /* multiple push instruction */
  P5.H = HI(myregister);    /* P5 used locally */
  P5.L = LO(myregister);
  R7 = [P5];                /* R7 used locally */
  R0 = R0 + R7;             /* R0 user for parameter passing*/
  (R7:7, P5:5) = [SP++];    /* multiple pop instruction */
  RTS;                      /* return from subroutine */

myfunction.end:             /* closing subroutine label */
```

Subroutines - No leaf functions

```
/* parent function */
    CALL function_a;

/* continue here after the call */
    JUMP somewhereelse;
```

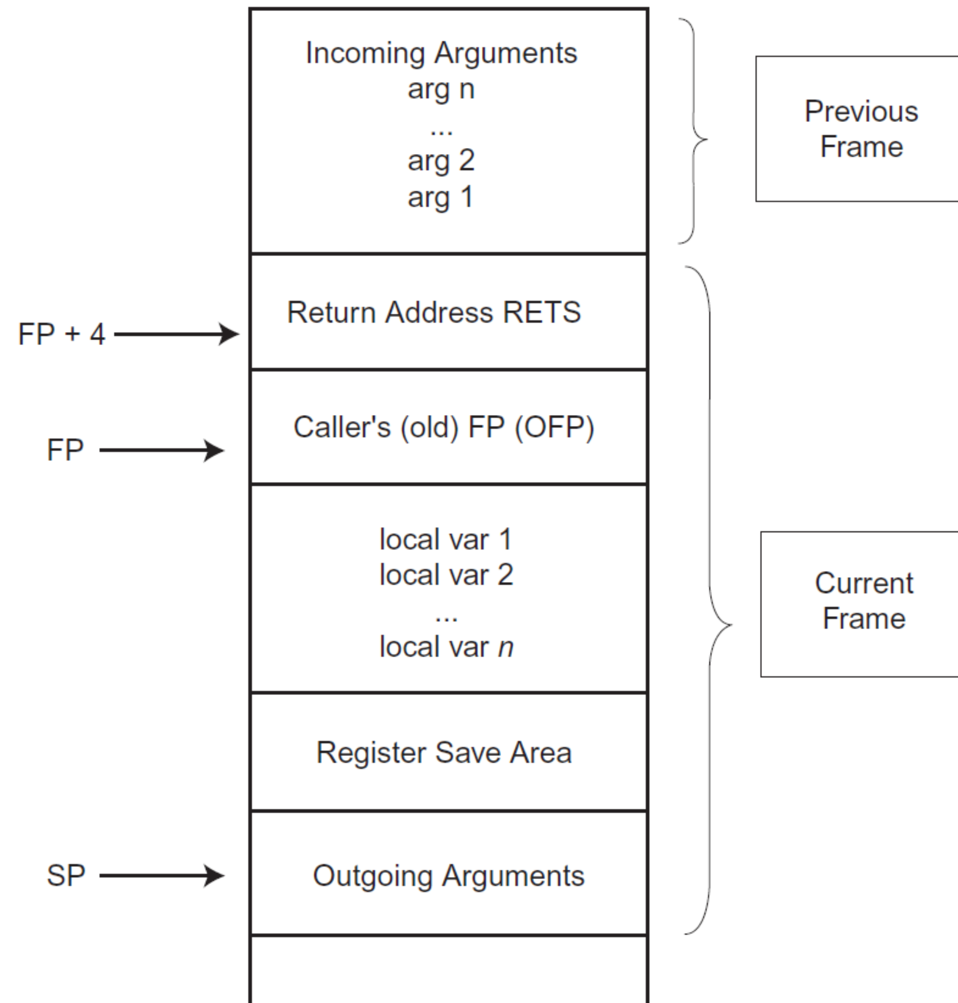






```
function_a:

    [--SP] = (R7:7, P5:5);
    [--SP] = RETS;        // save RETS onto stack

    CALL function_b;  // call further subroutines
    CALL function_c;

    RETS = [SP++];        // restore RETS
    (R7:7, P5:5) = [SP++];
    RTS;

function_b:
    /* do something */
    RTS;

function_c:
    /* do something else */
    RTS;
```

Managing the Stack



Subroutines - Parameter Passing

```
_parent:
...
R0 = 1;
R1 = 3;
[--SP] = R0;
[--SP] = R1;
CALL _sub;
R1 = [SP++]; /
R0 = [SP++]; // R
...
_parent.end:
```

```
_sub:
    [--SP] = FP;            // save frame pointer
    FP = SP;                // new frame

    [--SP] = (R7:5);        // multiple push

    R6 = [FP+4];            // R6 = 3
    R7 = [FP+8];            // R7 = 1

    R5 = R6 + R7;           // calculate anything
    R6 = R6 - R7;

    [FP+4] = R5;            // R5 = 4
    [FP+8] = R6;            // R6 = 2
    (R7:5) = [SP++];        // multiple pop

    FP = [SP++];            // restore frame pointer
    RTS;
_sub.end:
```

Stack/Frame Pointers

SP FF807F80
USP FF807FB0
FP FF807F98

Stack/Frame Pointers

SP FF807F84
USP FF807FB0
FP FF807F98

BLACKFIN Memory [Hexadeci...

0xFF807f8c

Address				
[FF807F44]	00	00	00	00
[FF807F48]	00	00	00	00
[FF807F4C]	00	00	00	00
[FF807F50]	00	00	00	00
[FF807F54]	00	00	00	00
[FF807F58]	00	00	00	00
[FF807F5C]	00	00	00	00
[FF807F60]	00	00	00	00
[FF807F64]	00	00	00	00
[FF807F68]	00	00	00	00
[FF807F6C]	00	00	00	00
[FF807F70]	00	00	00	00
[FF807F74]	00	00	00	00
[FF807F78]	00	00	00	00
[FF807F7C]	00	00	00	00
[FF807F80]	98	7F	80	FF
[FF807F84]	04	00	00	00
[FF807F88]	02	00	00	00
[FF807F8C]	00	00	00	00
[FF807F90]	00	00	00	00

Data Register File

	Whole	High
R0	00000001	0000
R1	00000003	0000
R2	00000000	0000
R3	00000000	0000
R4	00000000	0000
R5	00000000	0000
R6	00000000	0000
R7	00000000	0000

Subroutines
Link and Unlink Code Sequencer

LINK n;	UNLINK
[--SP] = RETS;	SP = FP;
[--SP] = FP;	FP = [SP++];
FP = SP;	RETS = [SP++];
SP += -n;	…..

```
_sub2:
LINK 0;
[--SP] = (R7:5);
R6 = [FP+8];        //R6 = 3
R7 = [FP+12];       //R7 = 1
R5 = R6 + R7;
R6 = R6 - R7;
[FP+8] = R5;        // R5 = 4
[FP+12] = R6;       // R6 = 2
(R7:5) = [SP++];
UNLINK;
RTS;
_sub2.end:
```

```
_sub3:
LINK 8;
[--SP] = (R7:0, P5:0);

R7 = 0 (Z);
[FP-4] = R7;
[FP-8] = R7;
...
(R7:0, P5:0) = [SP++];
UNLINK;
RTS;
_sub3.end:
```

Subroutines
Link and Unlink Code Sequencer

LINK n;	UNLINK
[--SP] = RETS;	SP = FP;
[--SP] = FP;	FP = [SP++];
FP = SP;	RETS = [SP++];
SP += -n;	…..

```
_sub2:
LINK 0;
[--SP] = (R7:5);
R6 = [FP+8];        //R6 = 3
R7 = [FP+12];       //R7 = 1
R5 = R6 + R7;
R6 = R6 - R7;
[FP+8] = R5;        // R5 = 4
[FP+12] = R6;       // R6 = 2
(R7:5) = [SP++];
UNLINK;
RTS;
_sub2.end:
```

```
_sub3:
LINK 8;
[--SP] = (R7:0, P

R7 = 0 (Z);
[FP-4] = R7;
[FP-8] = R7;
...
(R7:0, P5:0) = [S
UNLINK;
RTS;
_sub3.end:
```

If subroutines require local, private, and temporary variables beyond the capabilities of core registers, it is a good idea to place these variables on the stack as well.

Branches (III)

- Direct Short and Long Jumps
- Direct Call
- Indirect Branch and Call
- PC-Relative Indirect Branch and Call
- Subroutines
- **Condition Code Flag**
- Branch Prediction

Condition Code (CC) Flag

- The CC flag can resolve the direction of a branch or the movement of a register.

Conditional Branch

IF CC JUMP dest;

Conditional Register Move

IF CC R0 = P0;

In some cases, using this instruction instead of a branch eliminates the cycles lost because of the branch.

- There are eight ways of accessing the CC, and are used to control program flow.

Condition Code (CC) Flag (II)

1. A conditional branch is resolved by the value in CC.
2. A Data register value may be copied into CC, and the value in CC may be copied to a Data register.
3. The BITTST instruction accesses the CC flag.
4. A status flag may be copied into CC, and the value in CC may be copied to a status flag.
5. The CC flag bit may be set to the result of a Pointer register comparison.
6. The CC flag bit may be set to the result of a Data register comparison.
7. Some shifter instructions (rotate or BXOR) use CC as a portion of the shift operand/result.
8. Test and set instructions can set and clear the CC bit.

Branches (III)

- Direct Short and Long Jumps
- Direct Call
- Indirect Branch and Call
- PC-Relative Indirect Branch and Call
- Subroutines
- Condition Code Flag
- **Branch Prediction**

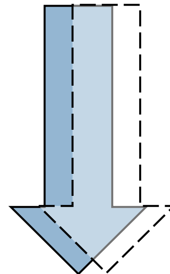
Branch Prediction

- Static branch strategy based on CC state.

Predicted-not-taken (default)

/* previous instructions */

If CC JUMP dest



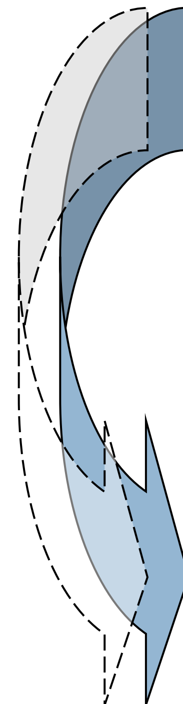
/* following instructions */

dest:

/* branch instructions */

Predicted-taken (bp)

/* previous instructions */



If CC JUMP dest (bp)

/* following instructions */

dest:

/* branch instructions */

Branch Prediction (II)

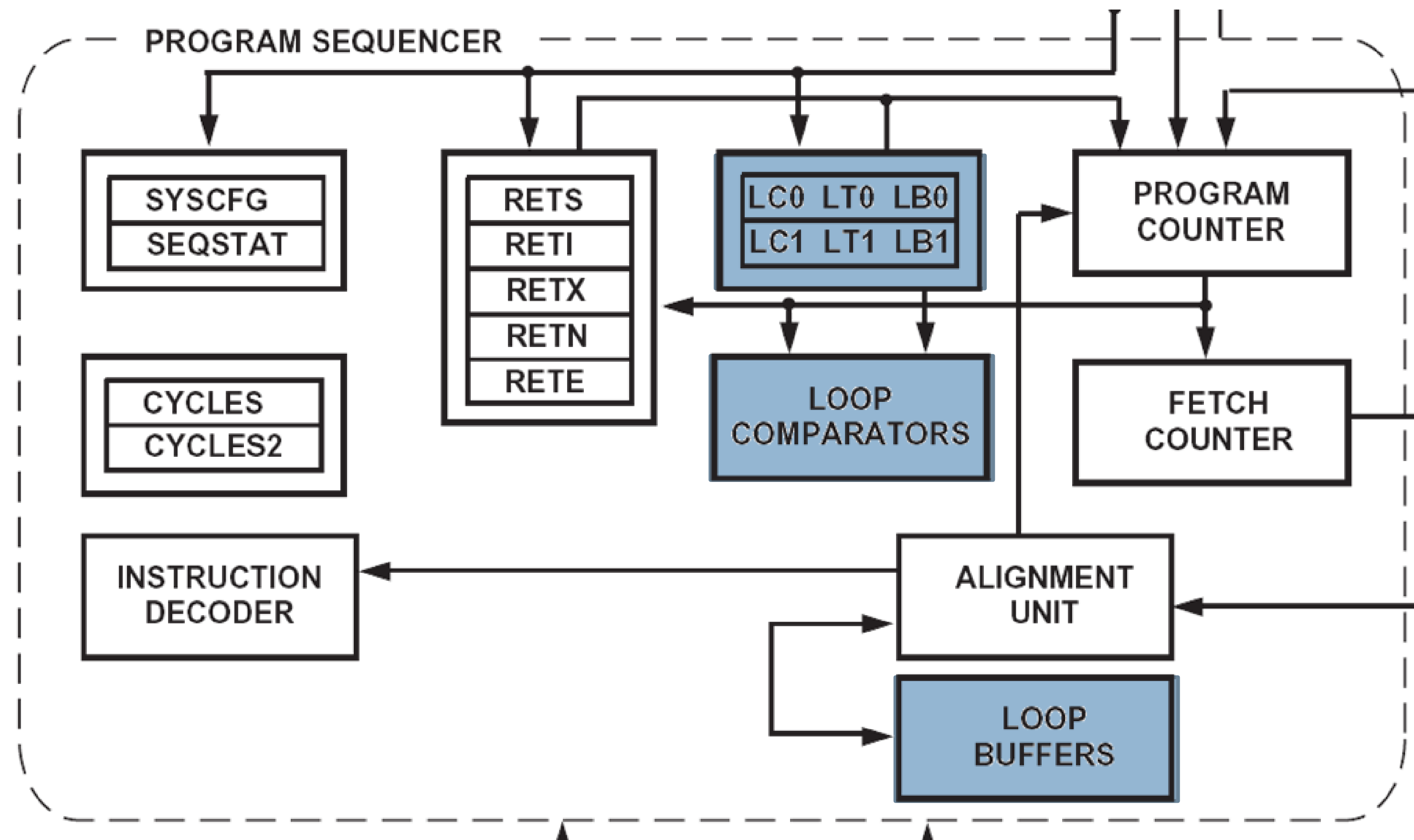
Branch latency (CPU cycles)

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="2">True</td></tr>
<tr><td>Not-taken</td><td>Taken</td></tr>
<tr><td rowspan="2">Prediction</td><td>Not-taken</td><td>0</td><td>8</td></tr>
<tr><td>Taken</td><td>8</td><td>4</td></tr>
</table>

Conclusion

Typically, code analysis shows that a good default condition is to predict branch-taken for branches to a prior address (backwards branches), and to predict branch-not-taken for branches to subsequent addresses (forward branches).

Hardware Loops



Hardware Loops (II)

- The sequencer supports a mechanism of **zero-overhead** looping.
- The sequencer contains two loop units, each containing three registers:
  - *Loop Top register* (LT0, LT1)

    Holds the address of the first instruction within a loop.
  - *Loop Bottom register* (LB0,LB1)

    Holds the address of the last instruction of the loop.
  - *Loop Count register* (LC0, LC1).

    Maintains a count of the remaining iterations of the loop.
- Loop unit 1 has a higher priority than loop unit 0.
- Loop unit 1 is used for the inner loop and loop unit 0 is used for the outer loop.

Hardware Loops (II)

- The sequencer supports a mechanism of **zero ov**
- Th ... containing three registers.

> When LCx = 0, the loop is disabled, and a single pass of the code executes. If the loop counter is derived from a variable with a range that may include zero, it is recommended to guard the loop against the zero case.

  - *Loop Top register* (LT0, LT1)

    Holds the address of the first instruction within a loop.
  - *Loop Bottom register* (LB0,LB1)

    Holds the address of the last instruction of the loop.
  - *Loop Count register* (LC0, LC1).

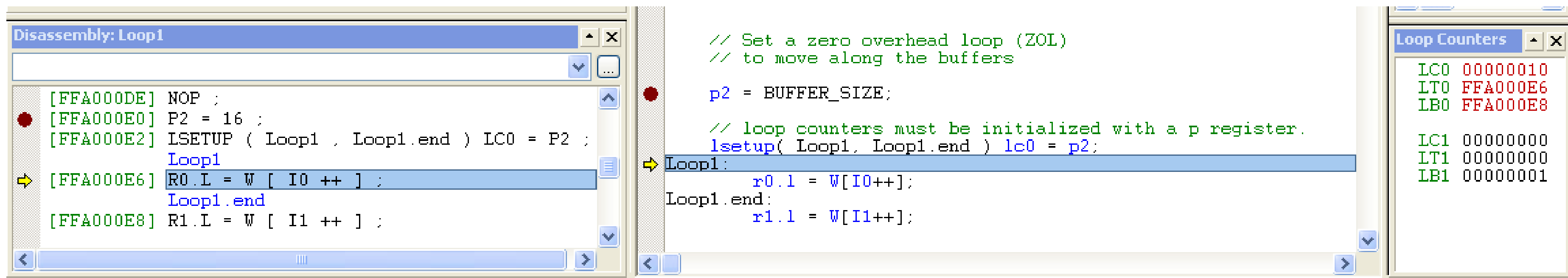
    Maintains a count of the remaining iterations of the loop.
- Loop unit 1 has a higher priority than loop unit 0.
- Loop unit 1 is used for the inner loop and loop unit 0 is used for the outer loop.

Hardware Loops (III)

- The processor supports a 4-location instruction ***loop buffer*** that reduces instruction fetches while in loops.
- If the loop code is <=4 instructions, then ***no fetches*** to instruction memory are necessary.
- The loop buffer effectively ***eliminates*** the instruction ***fetch time*** in loops with more than 4 instructions by allowing fetches to take place while instructions in the loop buffer are being executed.

Hardware Loops (IV)

- The LSETUP instruction can be used to load all three registers of a loop unit at once.
- Each loop register can also be loaded individually with a register transfer
- Loading individually incurs a significant overhead.



Hardware Loops (V)
Some restrictions

First/Last Address of the Loop	PC-Relative Offset Used to Compute the Loop Start Address	Effective Range of the Loop Start Instruction
Top / First	5-bit signed immediate; must be a multiple of 2.	0 to 30 bytes away from LSETUP instruction.
Bottom / Last	11-bit signed immediate; must be a multiple of 2.	0 to 2046 bytes away from LSETUP instruction (the defined loop can be 2046 bytes long).

- A ***4-cycle latency*** occurs on the first loopback when the LSETUP specifies a nonzero start offset (lp_start).
- The processor has no restrictions regarding which instructions can occur in a ***loop end position***. Branches and calls are allowed in that position.

Hardware Loops (VI)
Loop Unrolling

- Loops are often unrolled in order to pass only N-1 times.
  - The initial data fetch is executed before the loop is entered.
  - The final calculations are done after the loop terminates.
- This technique has two advantage:
  - Data is fetched exactly N times
  - I-Registers have their initial value after processing.
- The “algorithm” sequence can be executed multiple times without any need to initialize DAG-Registers again.

Hardware Loops (VI)
Loop Unrolling

- Loops are often unrolled in order to pass only N-1 time
  - The ... ntered.
  - The ... ates.
- This
  - Da
  - I-R
- The ... ultiple time ... sters agai

```
#define N 1024

// setup
I0.H = 0xFF80; I0.L = 0x0000; B0 = I0; L0 = N*2 (Z);
I1.H = 0xFF90; I1.L = 0x0000; B1 = I1; L1 = N*2 (Z);
P5 = N-1 (Z);

// algorithm

A0 = 0 || R0.H = W[I0++] || R1.L = W[I1++];

LSETUP (lp,lp) LC0 = P5;
lp:        A0+= R0.H * R1.L || R0.H = W[I0++] || R1.L = W[I1++];

A0+= R0.H * R1.L;
```

Hardware Loops (VII) Saving and Resuming Loops

- Is needed an special care when:
  - If the loop is interrupted by an ***interrupt service*** routine that also contains a hardware loop and ***requires*** the same ***loop unit***.
  - If the loop is interrupted by a preemptive task switch.
  - If the loop contains a ***CALL instruction*** that invokes an unknown subroutine that ***may have local loops***.
- This environment can be saved and restored by pushing and popping the loop registers.
- This takes multiple cycles, as the loop buffers must also be prefilled again.

Hardware Loops (VII)
Saving and Resuming Loops

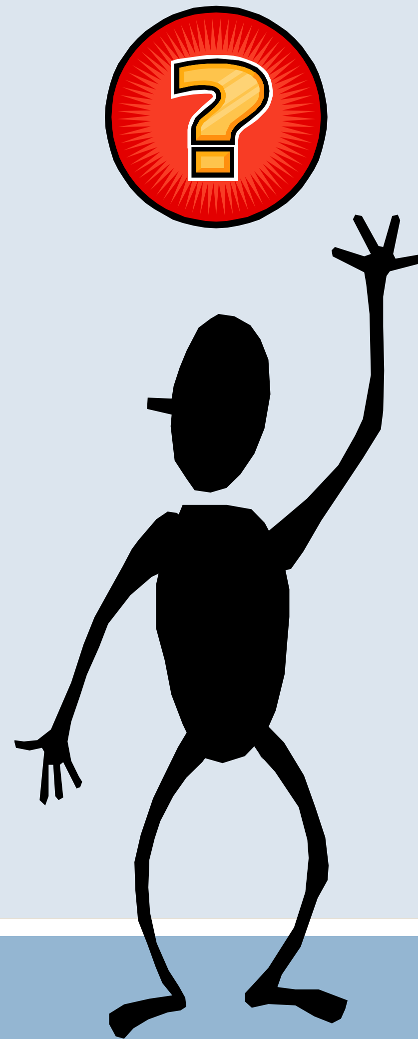
- Is neede
  - If the loo ... e routine that also ... es the same *loop un*...
  - If the loo ... witch.
  - If the loo ... vokes an unknown ... s.
- This envi ... ored by pushing ...
- This take ... uffers must also be p...

```
lhandler:

.....  //Save other registers here

[--SP] = LC0;     // save loop 0
[--SP] = LB0;
[--SP] = LT0;

.....  //Handler code here

LT0 = [SP++];
LB0 = [SP++];
LC0 = [SP++];    /* This will cause a "replay,"
                    that is, a ten-cycle refetch. */

.....  //Restore other registers here

RTI;
```

Recommended bibliography

- Blackfin Processor Programming Reference, Revision 1.3, September 2008
  - Ch4: PROGRAM SEQUENCER
  - Ch7: PROGRAM FLOW CONTROL
  - Ch10: STACK CONTROL
- WS Gan, SM Kuo. Embedded Signal Processing with the MSA. John Wiley and Sons. 2007
  - Ch 6: Introduction to the Blackfin Processor

- NOTE: Many images used in this presentation were extracted from the recommended bibliography.



Questions?

Thank you!