EE 382M.8
VLSI–II
Class Overview
Spring 2017

Mark McDermott

EE 382M VLSI - II

- **Instructor(s):**
  - Mark McDermott
    - email: mcdermot@ece.utexas.edu
    - Office: UTA 7.220
    - Phone: 512.471.3253
    - Office hours: 5-6:00 pm (before class)
  - Guest Lecturers: Gian Gerosa, Jacob Abraham, Alan Drake, Syed Alam
- **TA:**
  - Wuxi Li (wuxi.li@utexas.edu)
  - Office hours: Mon/Wed 16:00-17:30
  - Office: AHG 122
- **Prerequisite: Graduate VLSI-I, logic design, basic computer architecture.**
- **CLASS WEB site:**
  - www.ece.utexas.edu/~mcdermot/
- **PROJECT Directory:**
  - /home/projects/courses/spring_17/ee382m-16810/project_spring_17/

Do's and Don'ts

- **Do's**
  - Update your email address on Canvas and in ECE-LRC.
  - Show up for class on time. Class starts promptly at 6:30 pm.
  - Read the lecture notes before coming to class and ask lots of questions.

- **Do NOTS**
  - Leave your cell phone ON during class.

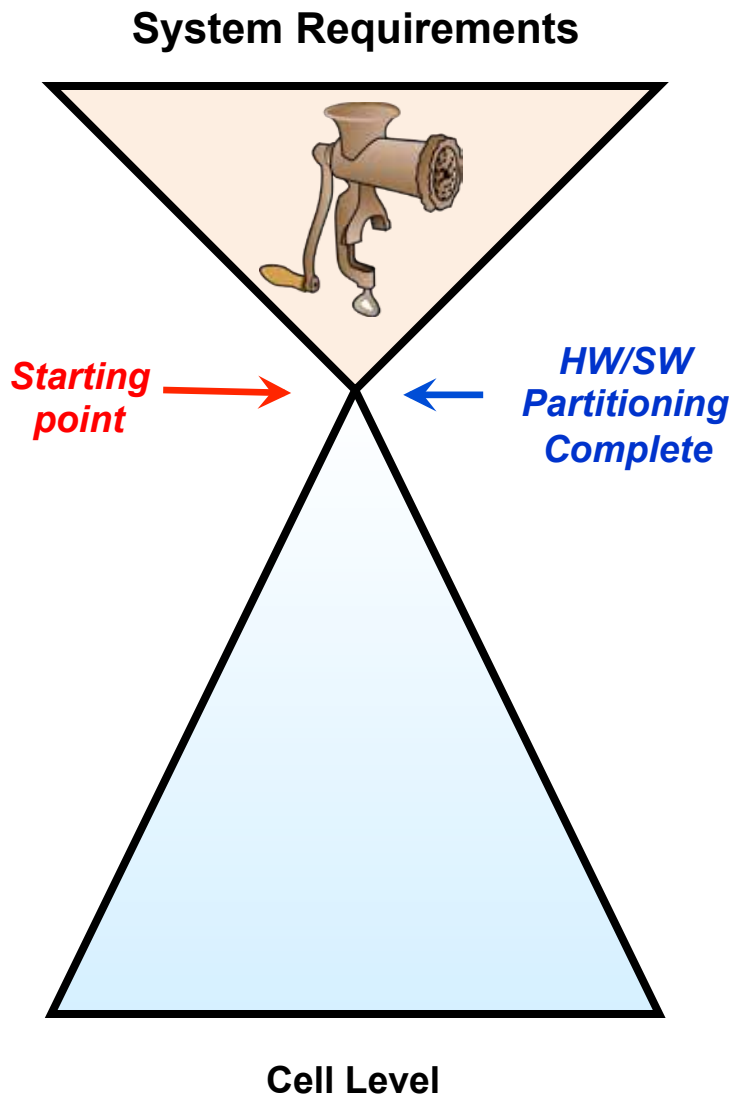
Goals of this class

This class will introduce you to the tools that are needed to solve VLSI design planning, circuit design and integration problems.

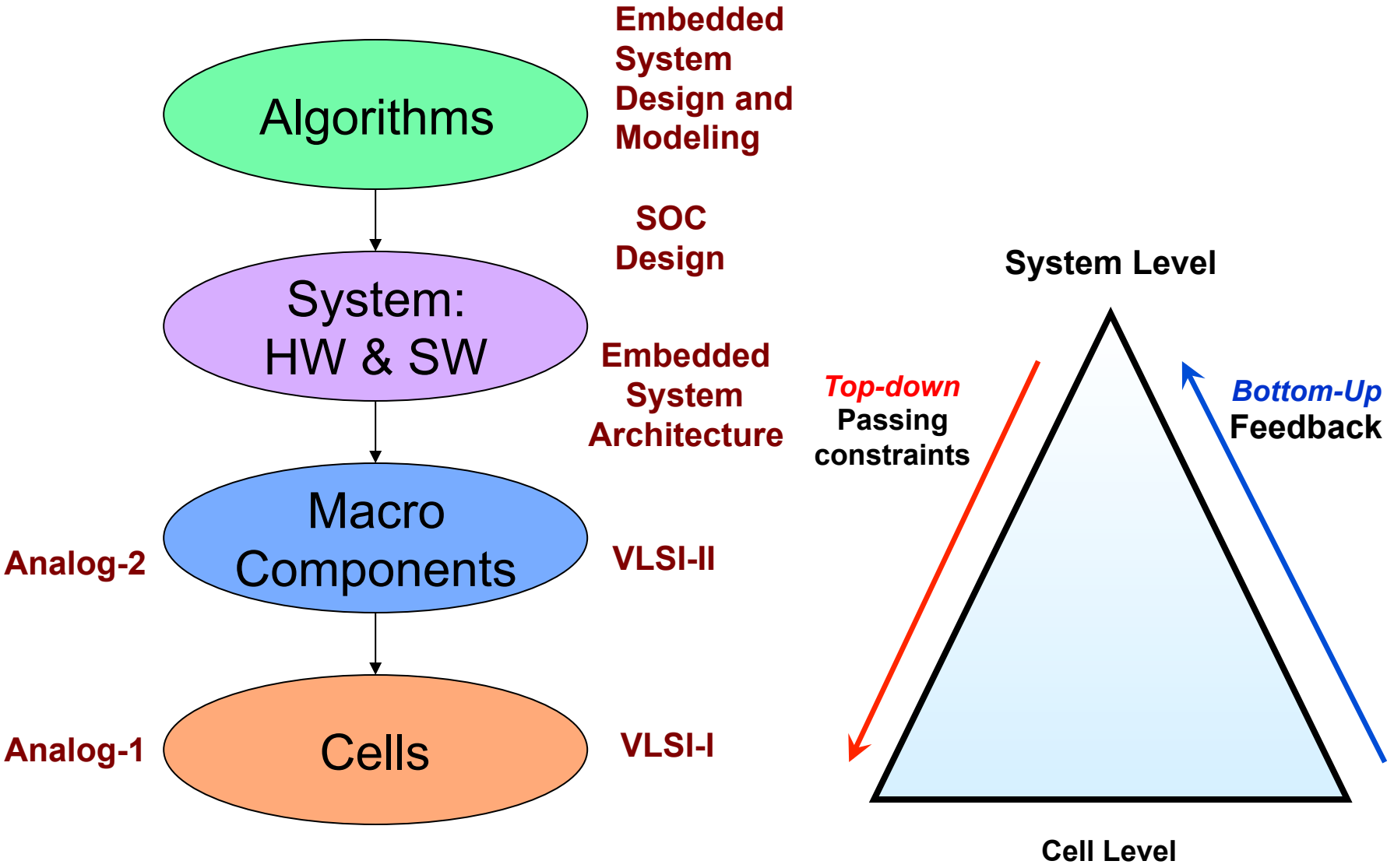
There will be 3 homework problems and an individual class project which will help you to practice using these tools and techniques.

The class provides examples of current design techniques, so that you can evaluate them and come to your own conclusions about their application in the real world. This experimentation will help you with building the foundation you need, to choose the appropriate circuits and simulation methods to solve design problems.

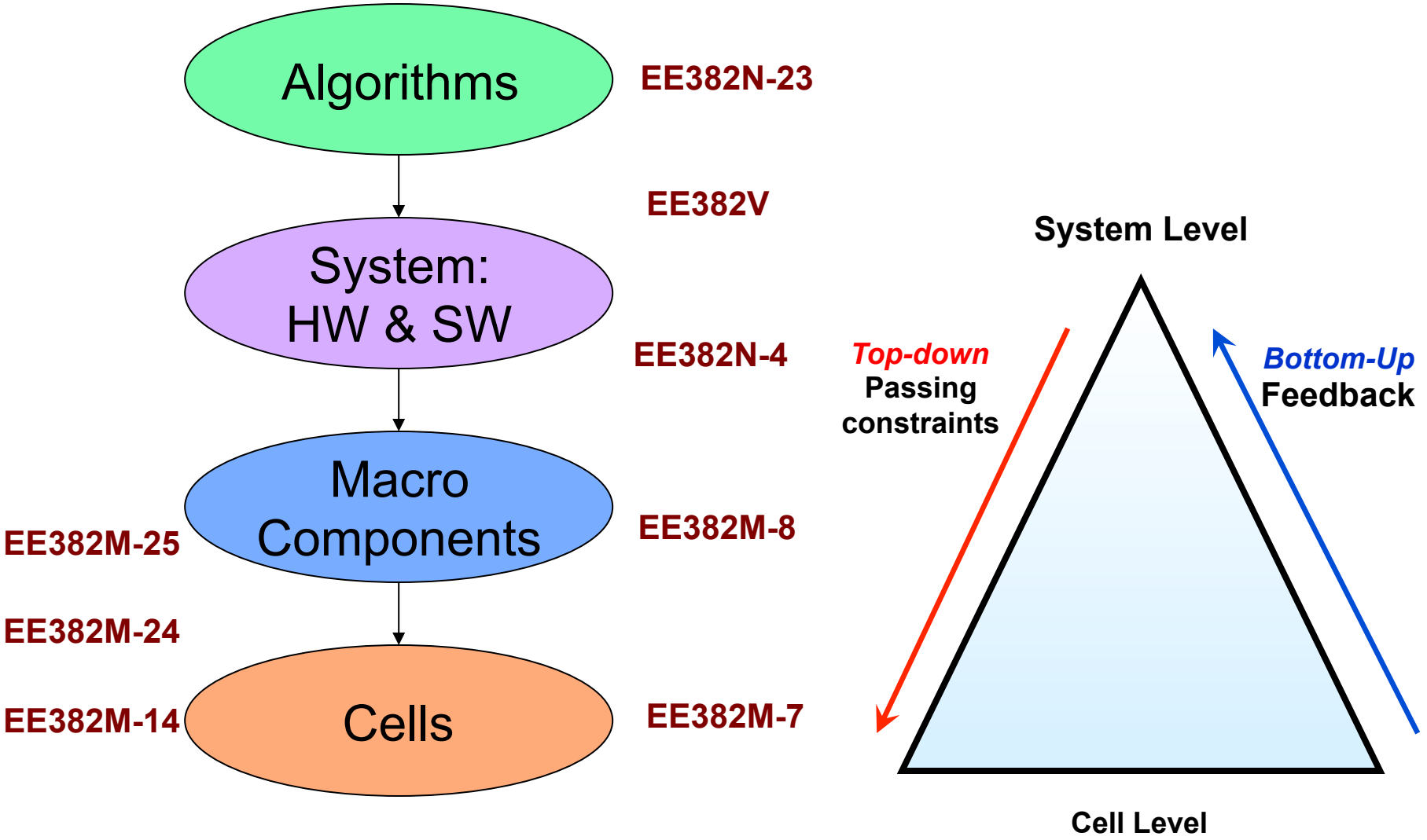
Design of SoC Systems



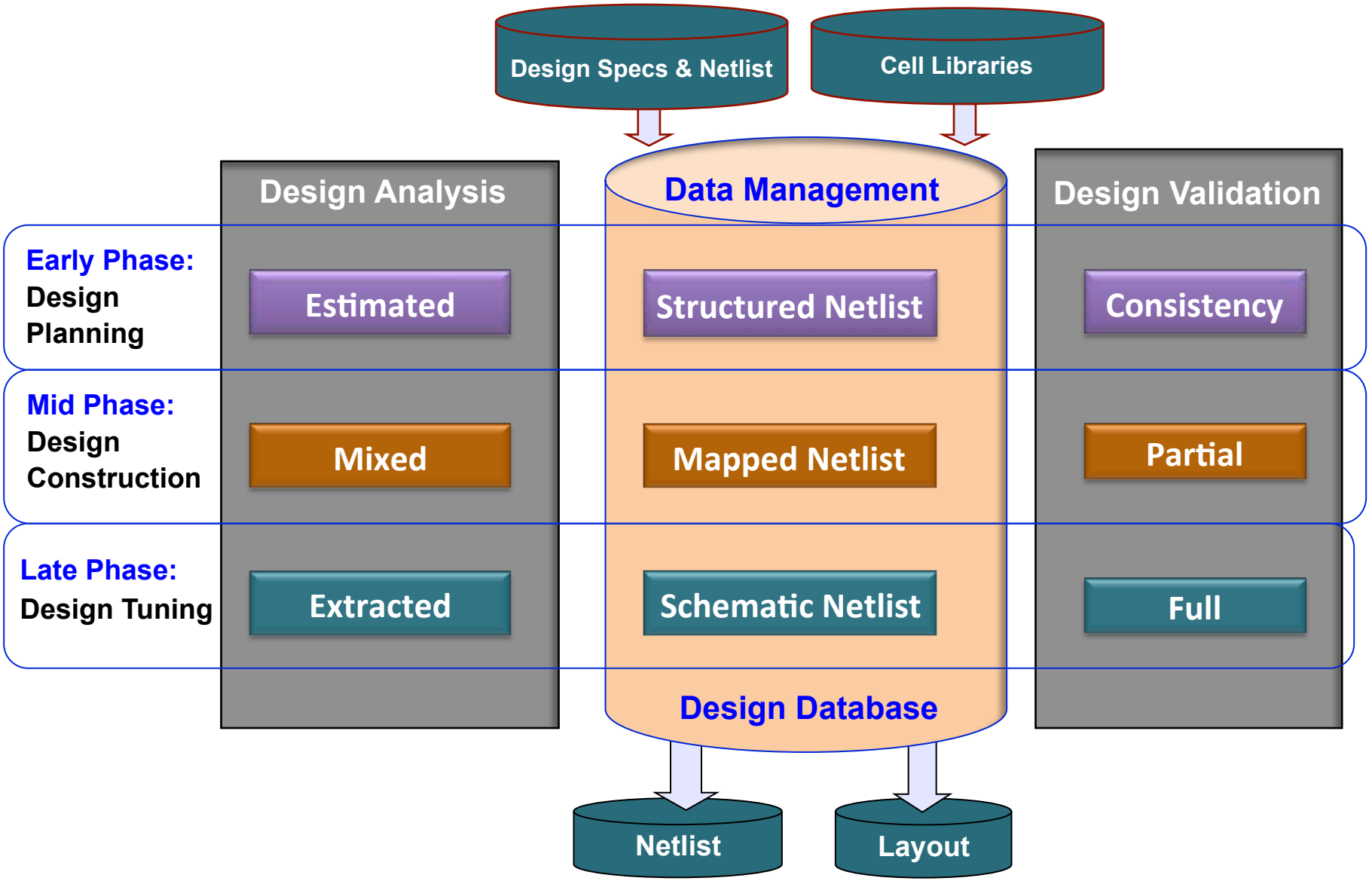
Design of SoC Systems



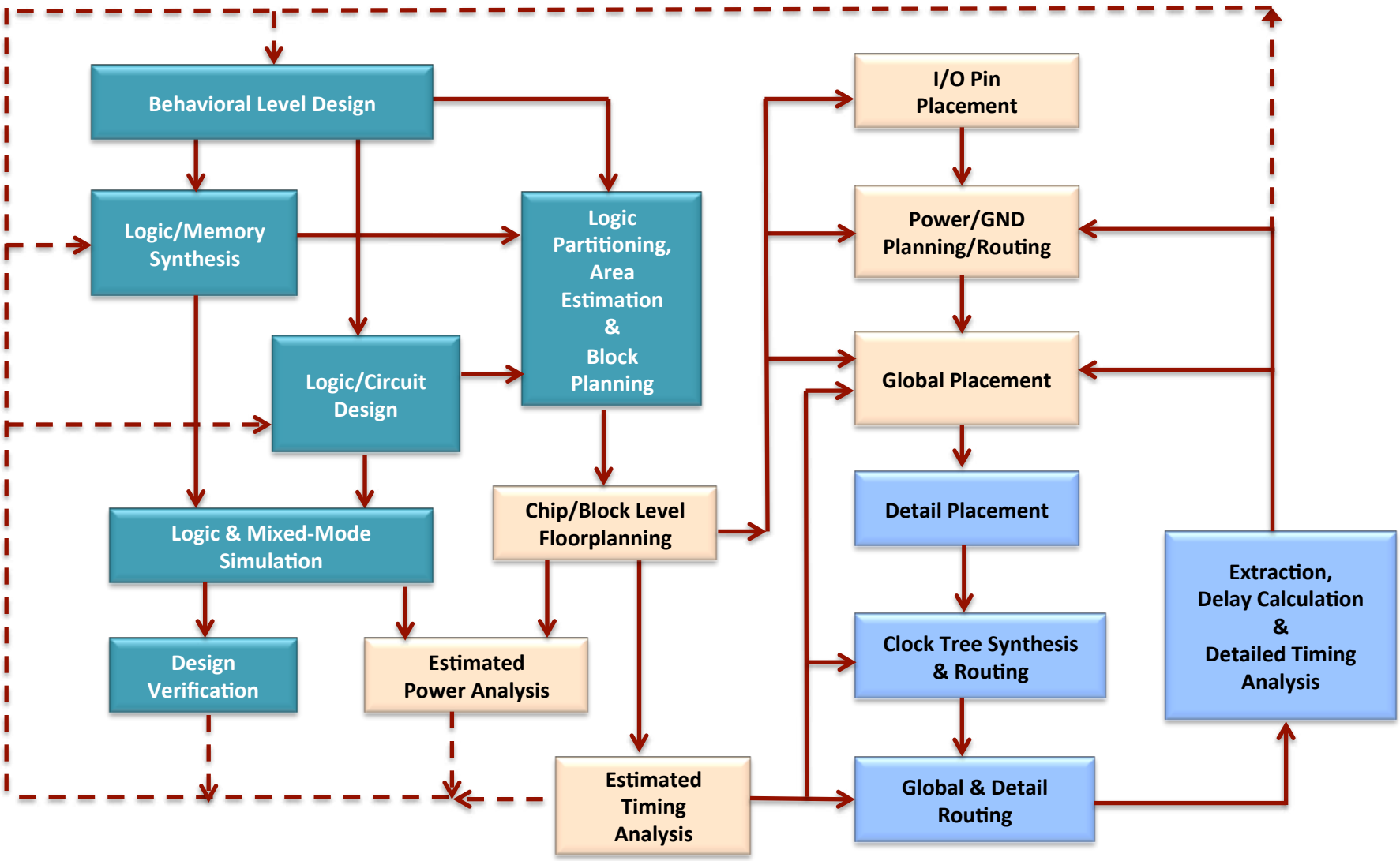
Design of SoC Systems



Design Phases



EE382M Design Flow



Class Project

- **Project assignment:**
  - Class project is focused on early design planning covering RTL synthesis, static timing, floor-planning including place & route + routing, power estimation and memory compilation for an ARM processor core and peripherals (will go into more details next). Final design reviews will be held on April 25th and May 2nd.
- **TOOLS:**
  - Synopsys HSPICE: for homework assignments
  - Synopsys Design Compiler: for RTL synthesis with open 32/28 nm standard cell library
  - Synopsys ICC: Auto-Place & Route
  - Synopsys PRIMETIME: Static timing analysis
  - Synopsys PRIMETIME –PX: Power estimation
  - Synopsys Open Memory Compiler: Memory design compiler
- **PROJECT Details:**
  - http://users.ece.utexas.edu/~mcdermot/vlsi2/Project_Details.pdf

Homework

- Homework is designed for you to get experience with Deep Sub-micron CMOS technology.
- Homework problems will be sent to you via email (PDF) and are due 4 weeks after. Problems MUST be worked out on your own. DO NOT share your work.
- No late assignments - Late homework submission will result in a 25% penalty per day. (Max penalty: 100% - Yes it is tough, but so is the real world).
- Other homework you must do: Read the course notes. Be prepared before coming to class.

HOMEWORK Directory:

/home/projects/courses/spring_17/ee382m-16810/hw/

Exams & Grading

- There will be one Mid-Semester Exam and a Final Exam. Due to the flow of the class these dates cannot be changed for any reason.

- Both exams are in class, closed notes, closed book, closed computer, closed iPhone, open mind, etc.

- Final Grading:
  - Homework: 15%
  - Mid Semester Exam: 20%
  - Final Exam: 25%
  - Project: 40%

TENTATIVE SCHEDULE for Spring 2017

Week	Date	Day	Time	Lecture Topic	Instructor	LECTURE #	Homework
1	Jan. 17	Tue	6:30 - 7:45pm	Introduction & Class Project Overview	McDermott	1	
1	Jan. 17	Tue	8:00 - 9:15pm	Transistor and Process Technology	McDermott	2	Deliver HW #1
2	Jan. 24	Tue	6:30 - 7:45pm	EDP: Front End	McDermott	3	
2	Jan. 24	Tue	8:00 - 9:15pm	EDP: Timing Analysis	McDermott	4	
3	Jan 31	Tue	6:30 - 7:45pm	EDP: Back End incl. Memories	McDermott	5	
3	Jan 31	Tue	8:00 - 9:15pm	Overview of Process and Design Variation	McDermott	6	
4	Feb. 7	Tue	6:30 - 7:45pm	Static Timing Analysis (STA)	McDermott	7	
4	Feb. 7	Tue	8:00 - 9:15pm	Statistical STA + PRIMETIME review	McDermott	8	
5	Feb. 14	Tue	6:30 - 7:45pm	Flip-flop design	Gerosa	9	Deliver HW #2
5	Feb. 14	Tue	8:00 - 9:15pm	CMOS Level Shifters, FLOP-based dividers	Gerosa	10	Collect HW #1
6	Feb. 21	Tue	6:30 - 7:45pm	Design for Testability	Abraham	11	
6	Feb. 21	Tue	8:00 - 9:15pm	Clocking	McDermott	12	
7	Feb 28	Tue	6:30 - 7:45pm	Variability Aware Circuit Design	Drake	13	
7	Feb 28	Tue	8:00 - 9:15pm	Low Power Circuit Design	Drake	14	
8	Mar. 7	Tue	6:30 - 10:00pm	Mid-Semester Project Review			Deliver HW # 3

TENTATIVE SCHEDULE for Spring 2017 (Cont'd)

9				SPRING BREAK (Mar 14 - 19)			
10	Mar. 21	Tue	6:30 - 10:00pm	Mid-Semester EXAM: Based on Lectures 2-14			Collect HW #2
11	Mar 28	Tue	6:30 - 7:45pm	Noise Analysis	McDermott	15	
11	Mar 28	Tue	8:00 - 9:15pm	Array Circuit Design	McDermott	16	
12	Apr 4	Tue	6:30 - 7:45pm	IO Design	McDermott	17	
12	Apr 4	Tue	8:00 - 9:15pm	Interconnect & Fabrics	McDermott	18	
13	Apr 11	Tue	6:30 - 7:45pm	Non-Volatile Memories 1	Alam	19	Collect HW #3
13	Apr 11	Tue	8:00 - 9:15pm	Non-Volatile Memories 2	Alam	20	
14	Apr 18	Tue	6:30 - 7:45pm	Power Gating 1	McDermott	21	
14	Apr 18	Tue	8:00 - 9:15pm	Power Gating 2	McDermott	22	
15	Apr 25	Tue	6:30 - 9:00pm	Final Project Review			
16	May 2	Tue	6:30 - 9:00pm	Final Project Review			

Caveats (courtesy of Mark Horowitz)

- Never take anything on blind faith. Work it out -- make sure it works. Many clever circuits that are published either don't work or are very sensitive to certain conditions. Be careful.
- There are NO RIGHT answers, and there are no PERFECT circuits. Everything has its warts. A good circuit simply has the right set of warts to meet the constraints of the problem.
- Simulation is NO substitute for thinking. HSPICE can make your job much easier, but it also can make it incredibly harder. Like all tools it helps only if you use it well, AND that requires thinking. Work it out on paper and then let HSPICE validate it.

DO NOT BECOME a SLAVE to the TOOLS.
Tools can (and will) tell you wrong answers.

Collaboration is good! Cheating is not !

- **Collaboration is good!**
  - Discussing issues with your classmates is a good way to learn and a study group is a very effective learning tool
  - Helping each other learn is particularly satisfying
  - But....Individual assignments and exams must be done by individuals

- **Cheating is a serious breach of trust and will not be tolerated**
  - If ever in doubt, don't do it or ask me immediately for a clarification
    - See University Policies for further detail
  - Take some free advice from me: Don't cheat, its not worth it

Suggested Reading

- Chandrakasan, Bowhill, Fox, *Design of High-Performance Microprocessor Circuits*, IEEE Press, 2000.
- Bernstein, et al., *High Speed CMOS Design Styles*, Kluwer Academic
- Harris, *Skew Tolerant Circuit Design*, Morgan Kaufmann Publishers
- Weste and Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective* (second edition), Addison Wesley
- Bakoglu, *Circuits, Interconnections and Packaging for VLSI*, Addison Wesley

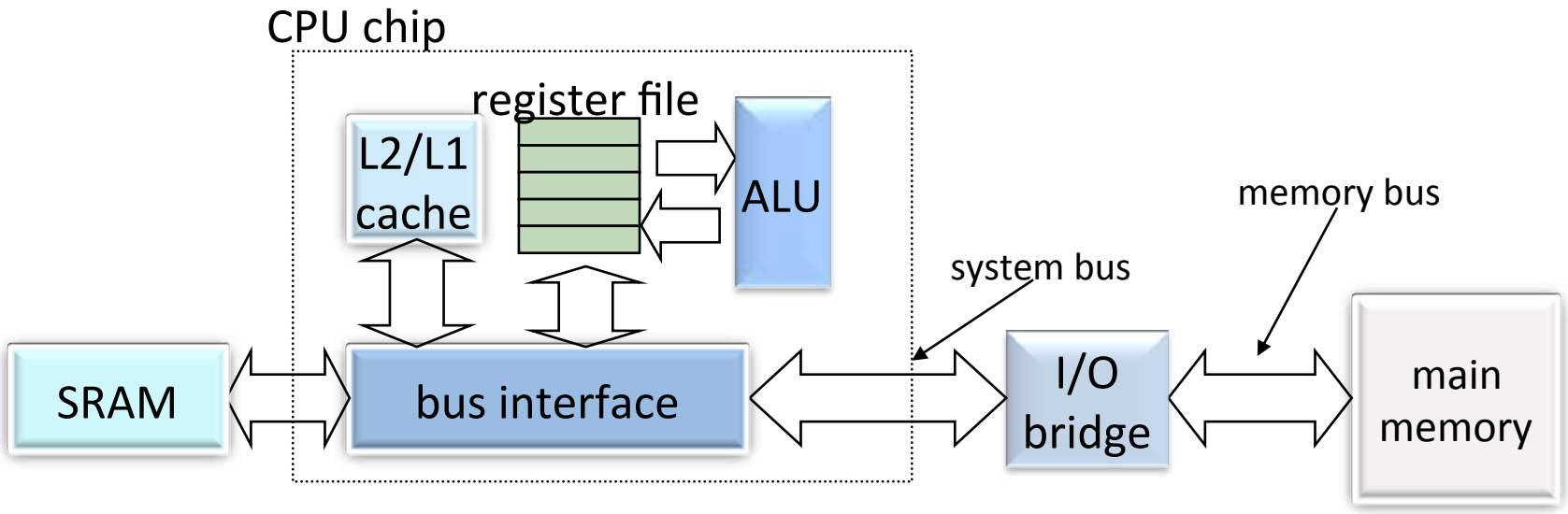
Questions?

Amber ARM Overview

But first, an architecture review

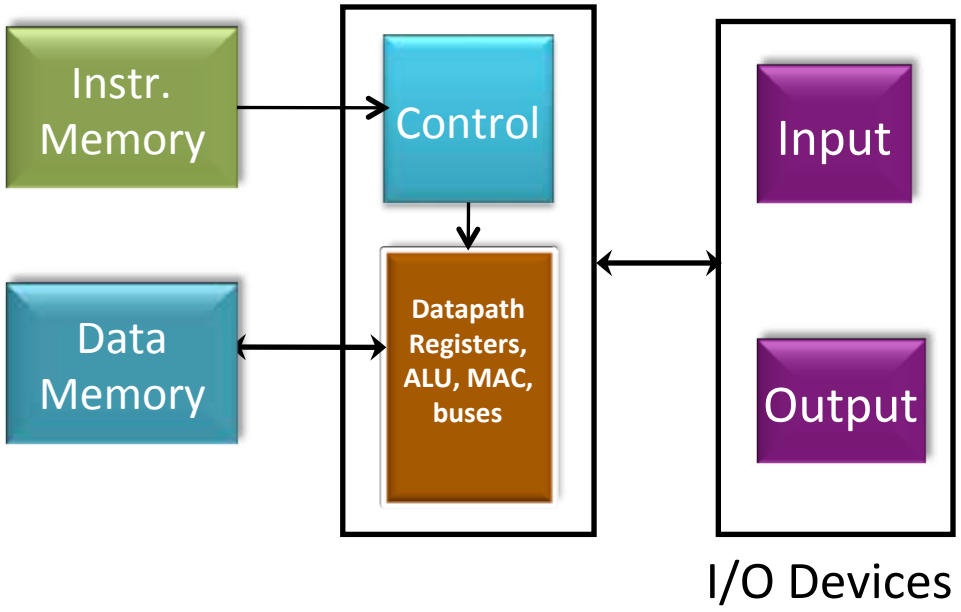
The Von Neumann Computer Model

- **Partitioning of the programmable computing engine into components:**
  - Central Processing Unit (CPU): Control Unit (instruction decode , sequencing of operations), Datapath (registers, arithmetic and logic unit, buses).
  - Memory: Instruction and data operand storage.
  - Input/Output (I/O) sub-system: I/O bus, interfaces, devices.
  - The stored program concept: Instructions from an instruction set are fetched from a common memory and executed one at a time

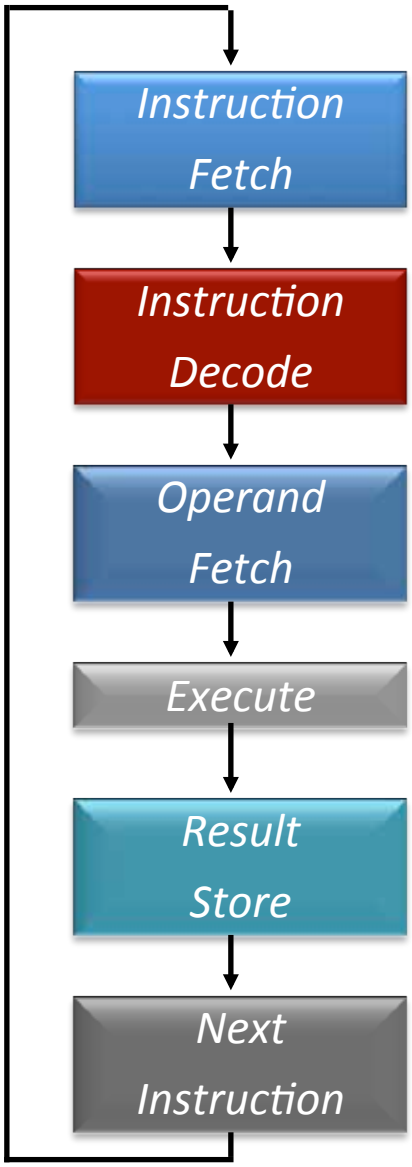


Harvard Computer Model

- **Similar to Von Neumann computer model with the following exceptions:**
  - **Instruction memory and data memory are separate**
  - **Self modifying code is not possible**



Generic CPU Machine Instruction Processing Steps



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

Compute result value or status

Deposit results in storage for later use

Determine successor or next instruction

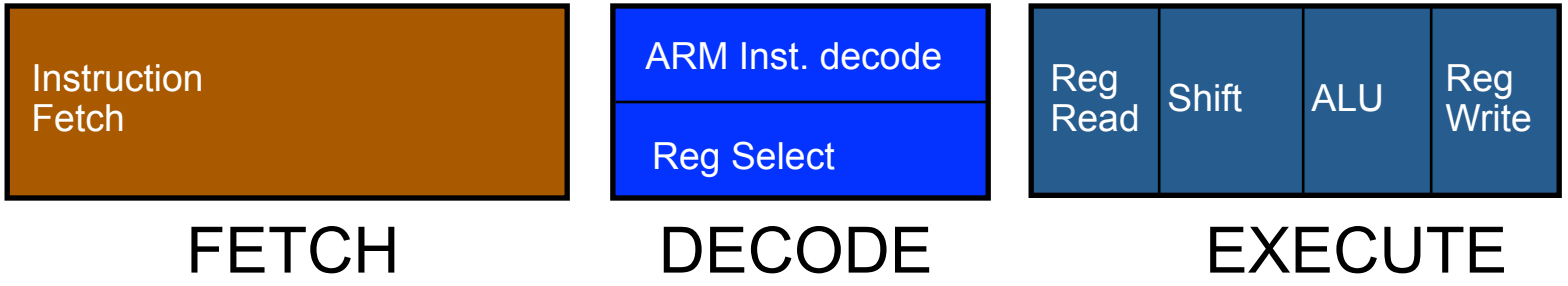
NOTE: These steps can be performed in a single clock cycle. There are numerous issues with single cycle machines.

Take Dr. Patt's class for details.

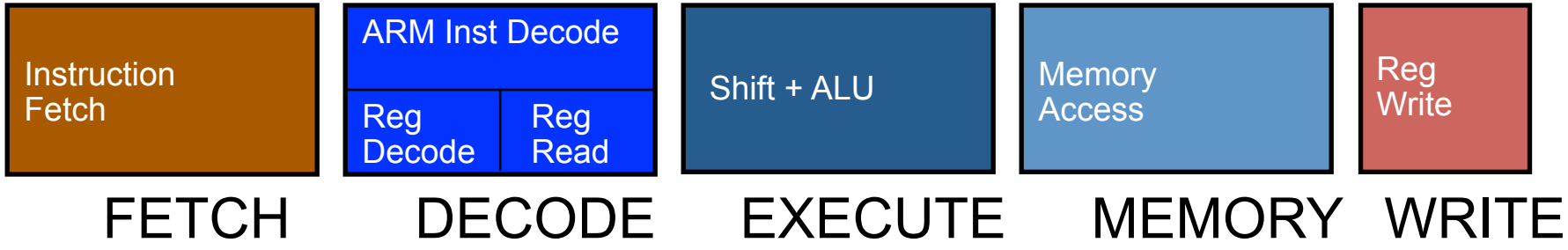
Now the Amber ARM Overview

ARM 3-Stage and 5-stage Pipelines

3-Stage



5-Stage



Amber Overview

- **Amber processor core is an ARM-compatible 32-bit RISC processor**
  - Fully compatible with the ARM. v2a instruction set architecture (ISA)
  - Supported by the GNU toolset
  - Two cores are available: Amber-23 and Amber-24
- **Amber 23:**
  - 3-stage pipeline.
  - 32-bit Wishbone system bus.
  - Unified instruction and data cache, with write through and a read-miss replacement policy. The cache can have 2, 3, 4 or 8 ways and each way is 4kB.
  - Multiply and multiply-accumulate operations with 32-bit inputs and 32-bit output in 34 clock cycles using the Booth algorithm. This is a small and slow multiplier implementation.
  - Little endian only, i.e. Byte 0 is stored in bits 7:0 and byte 3 in bits 31:24.

Amber Overview (cont)

- **Amber 25**
  - 5-stage pipeline.
  - 32-bit Wishbone system bus.
  - Separate instruction and data caches. Each cache can be either 2,3,4 or 8 ways and each way is 4kB. Both caches use a read replacement policy and the data cache operates as write through. The instruction cache is read only.
  - Multiply and multiply-accumulate operations with 32-bit inputs and 32-bit output in 34 clock cycles using the Booth algorithm. This is a small and slow multiplier implementation.
  - Little endian only, i.e. Byte 0 is stored in bits 7:0 and byte 3 in bits 31:24.

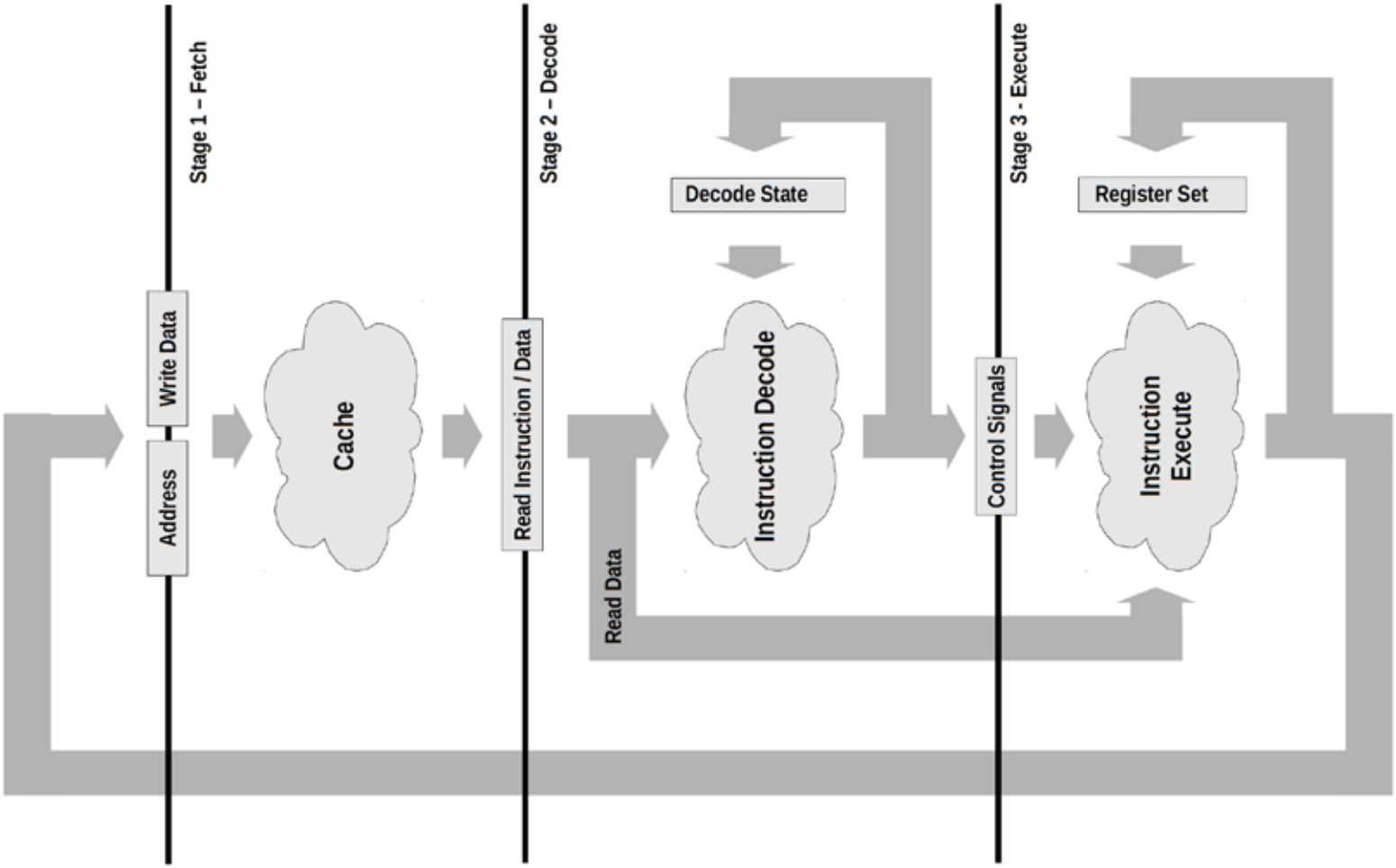
- **Both cores implement exactly the same ISA and are 100% software compatible.**

Amber Overview (cont)

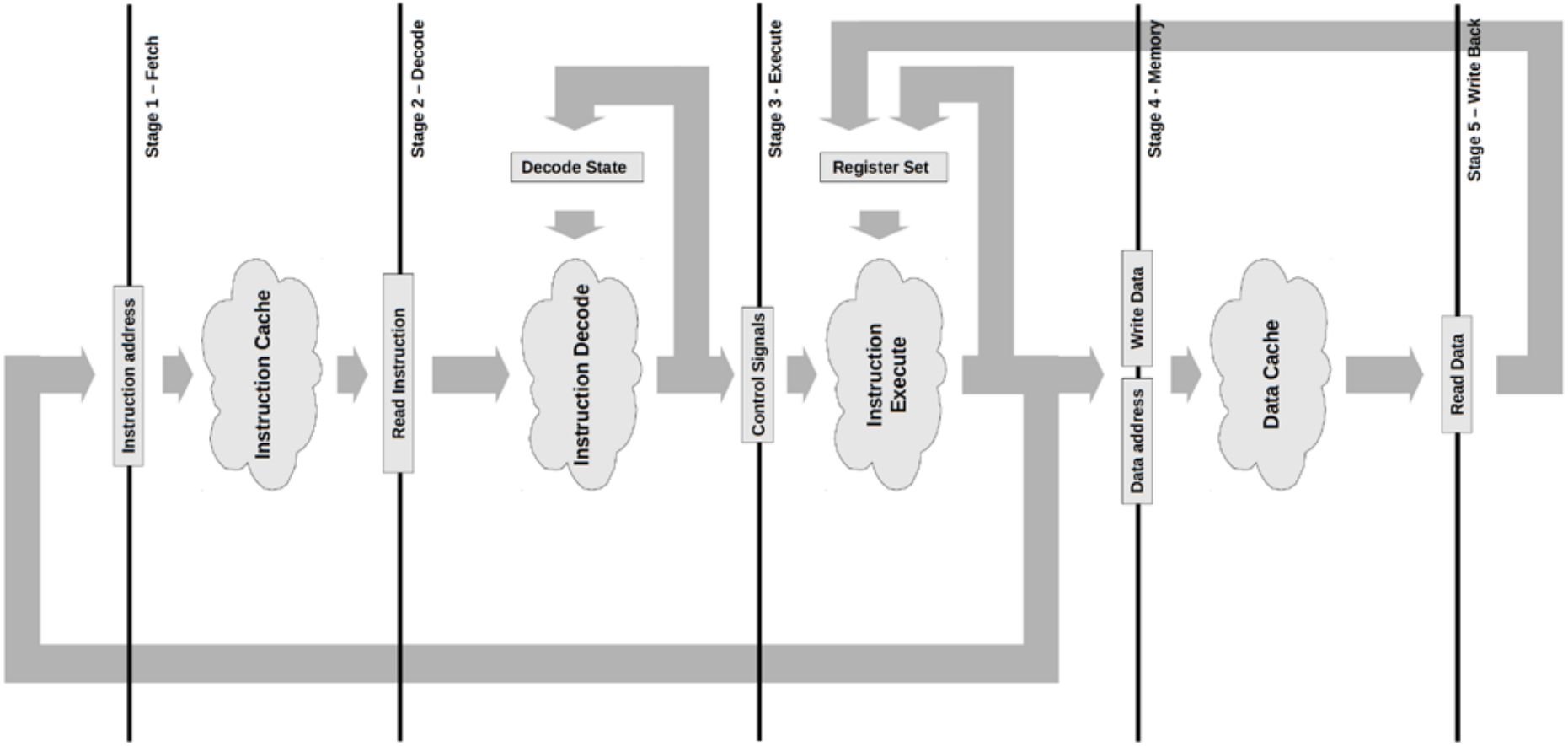
- Cores were developed in Verilog 2001
  - optimized for FPGA synthesis.
  - There is no reset logic, all registers are reset as part of FPGA initialization.

- Complete system has been tested extensively on the Xilinx SP605 Spartan-6 FPGA board. The full Amber system with the A23 core uses 32% of the Spartan-6 XC6SLX45T-3 FPGA Look Up Tables (LUTs).
  - The maximum frequency is limited by the execution stage of the pipeline which includes a 32-bit barrel shifter, 32-bit ALU and address incrementing logic.

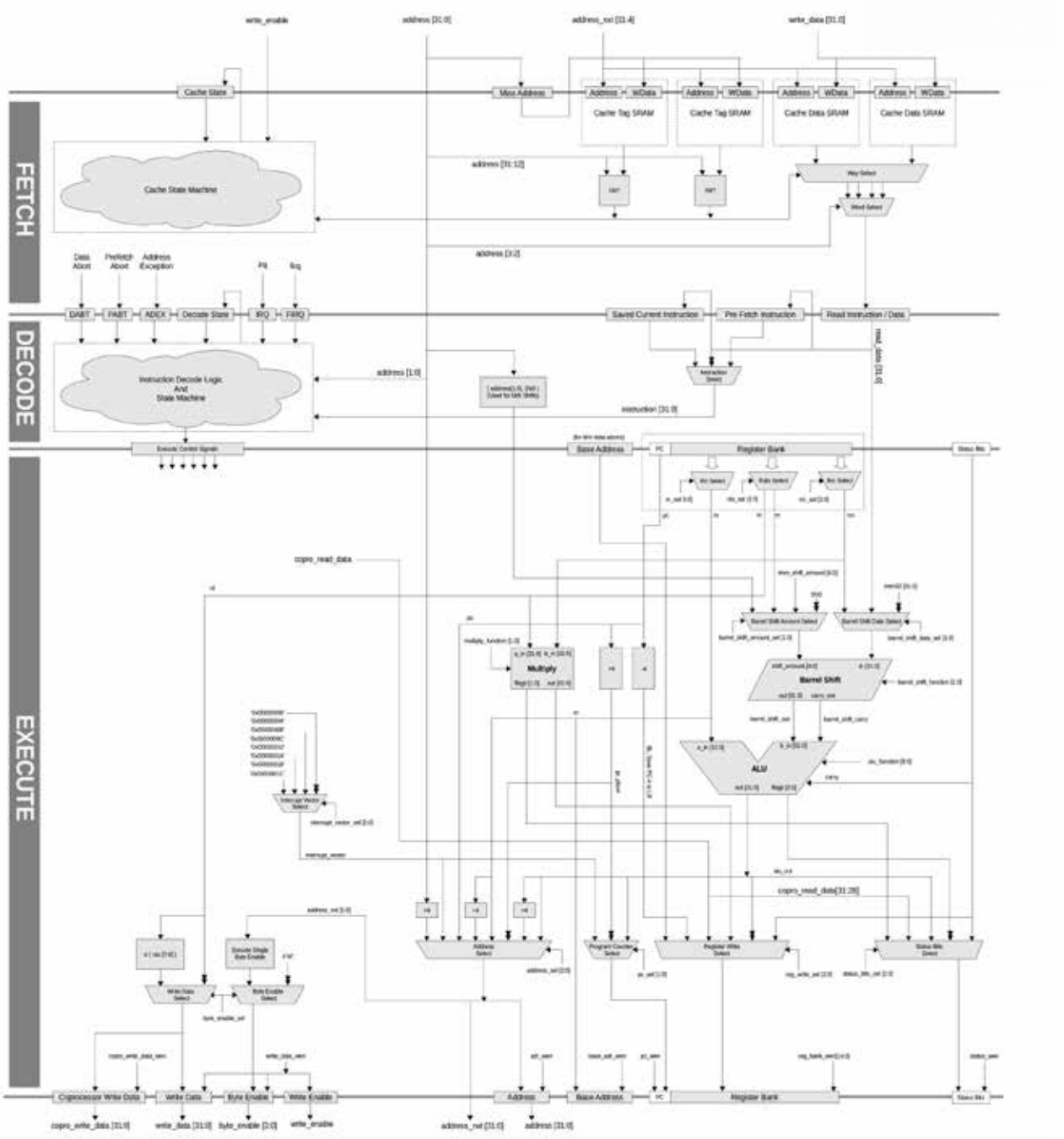
Amber 23 Pipeline



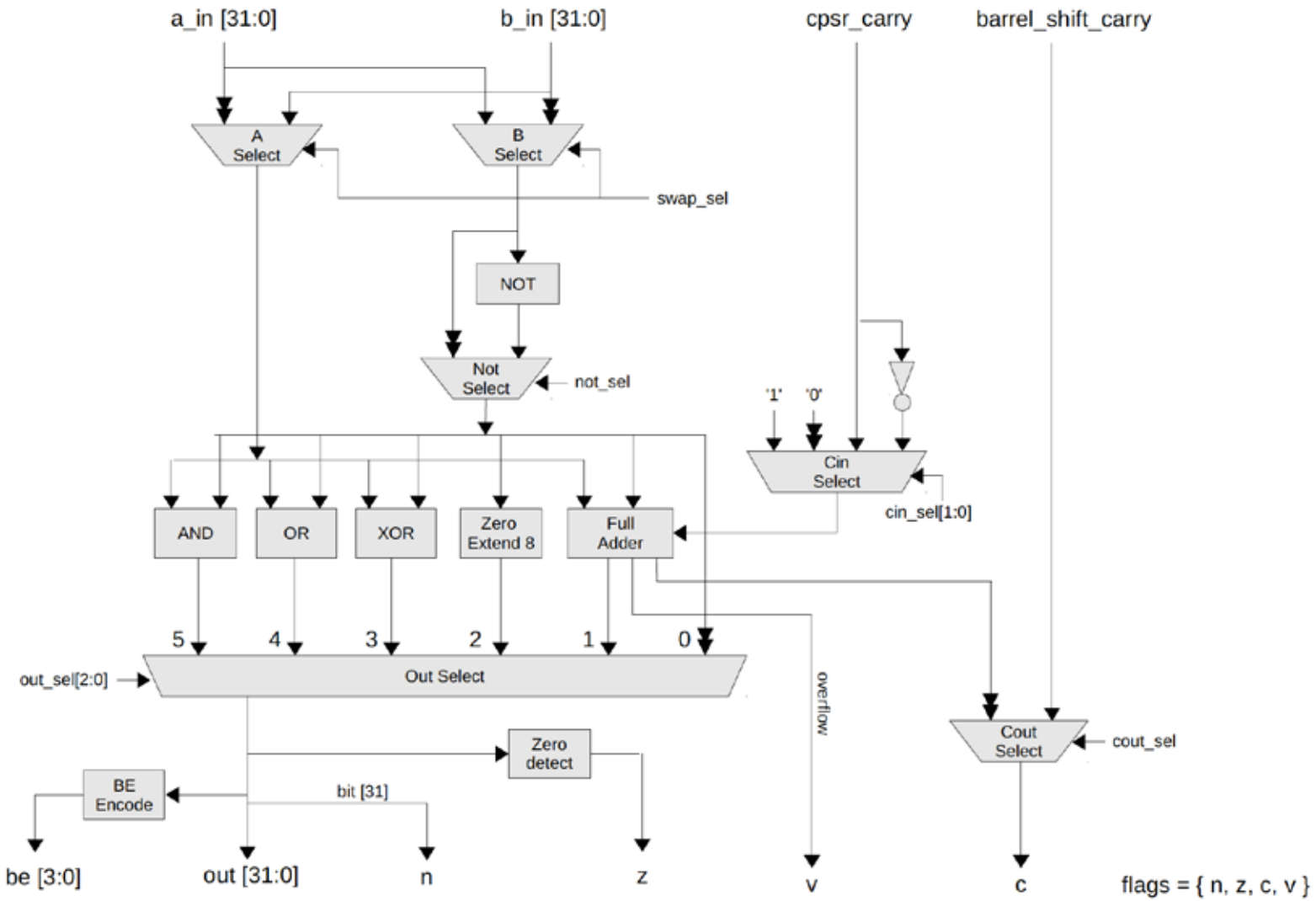
Amber-25 Pipeline



Detailed Amber23 Pipeline



Amber-23 ALU

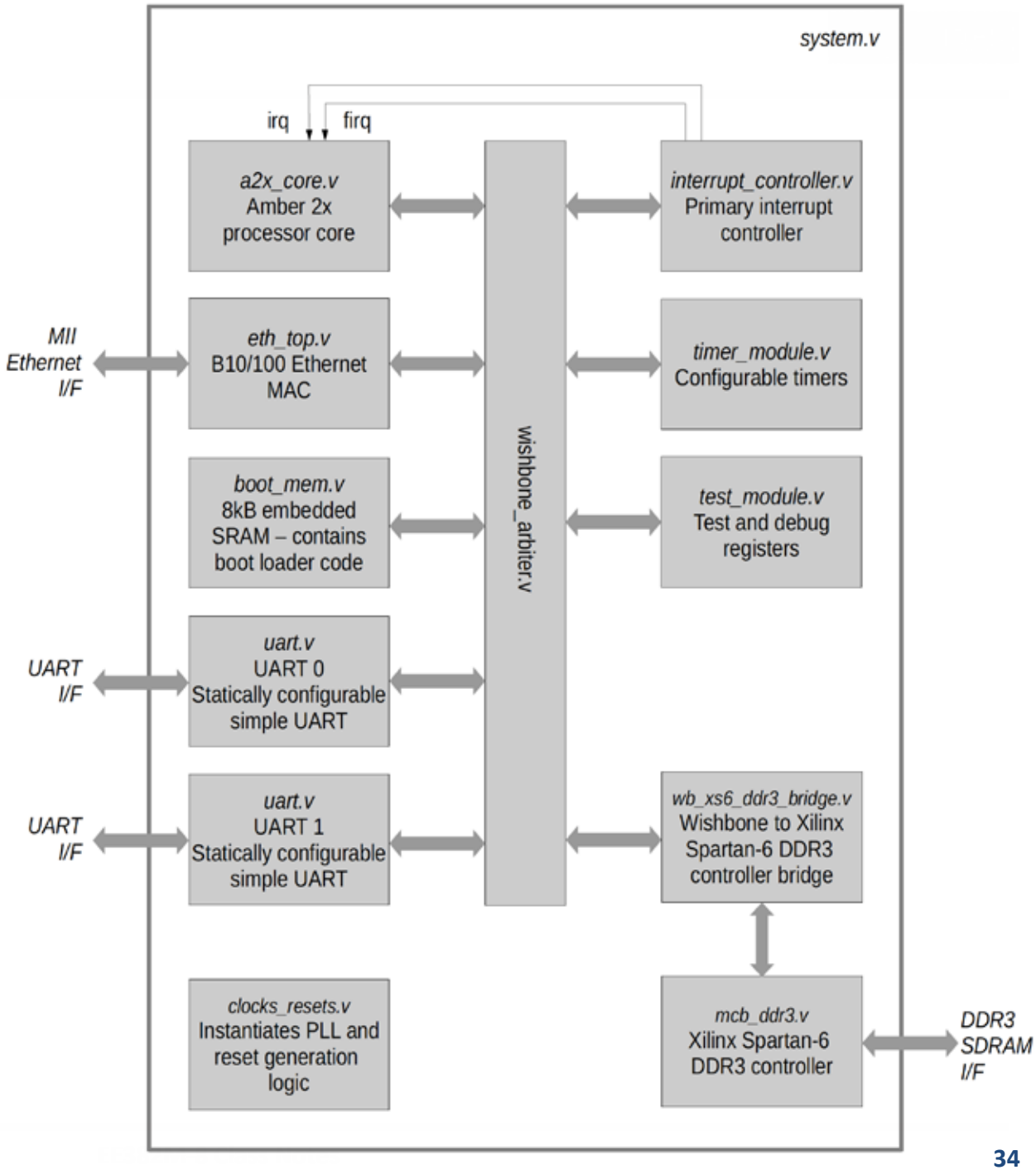

a_in [31:0]
b_in [31:0]
cpsr_carry
barrel_shift_carry
A Select
B Select
swap_sel
NOT
Not Select
not_sel
'1'
'0'
Cin Select
cin_sel[1:0]
AND
OR
XOR
Zero Extend 8
Full Adder
5
4
3
2
1
0
out_sel[2:0]
Out Select
overflow
Cout Select
cout_sel
Zero detect
BE Encode
bit [31]
be [3:0]
out [31:0]
n
z
v
c
flags = { n, z, c, v }

Caches

- **The Amber cache size is optimized to use FPGA Block RAMs. Each way has 256 lines of 16 bytes. 256 lines x 16 bytes x 2 ways = 8k bytes. The address tag is 20 bits. Each cache can be configured with either 2, 3, 4 or 8 ways.**

Ways	2	3	4	8
Lines per way	256	256	256	256
Words per line	4	4	4	4
Total words	2048	3072	4096	8192
Total bytes	8192	12288	16384	32768
FPGA 9K Block RAMs	8 + 2 = 10	12 + 3 = 15	16 + 4 = 20	32 + 8 = 40

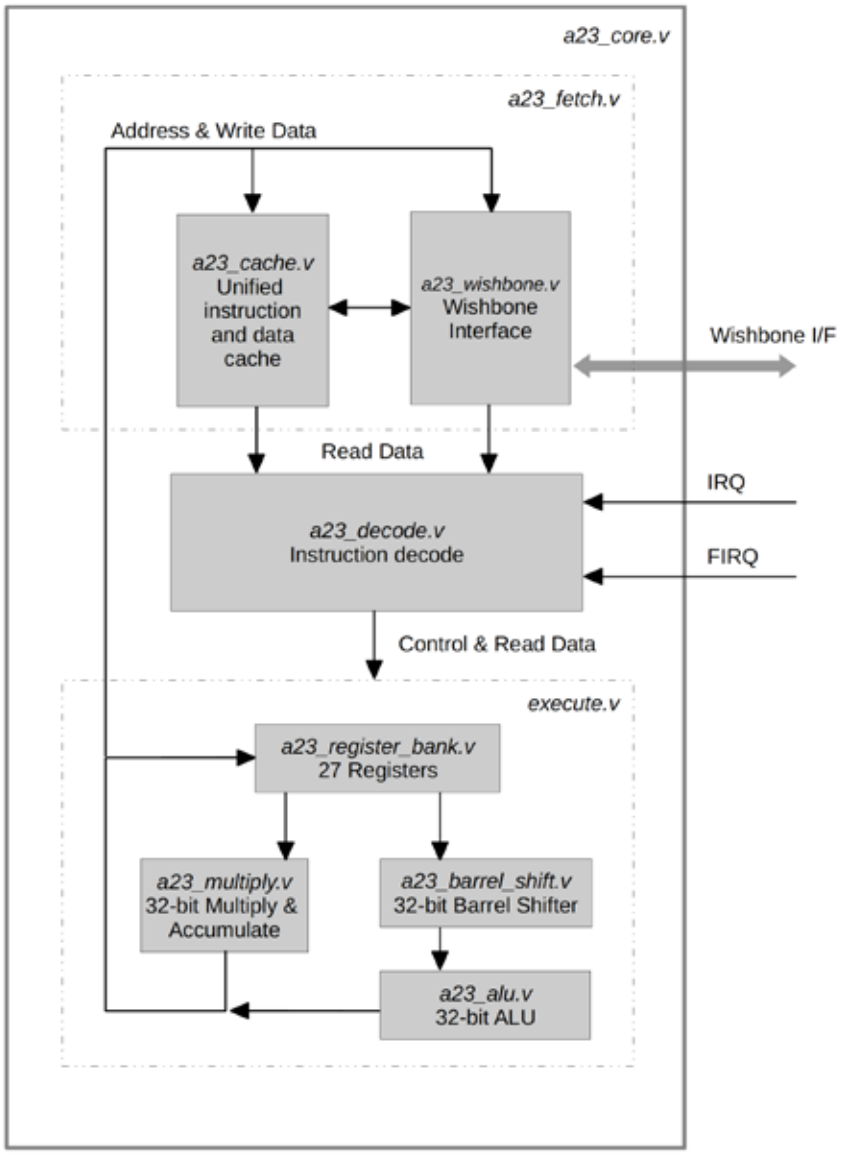
Amber System Level Configuration



Amber Wishbone Interface Signals

Name	Width	Direction	Description
i_clk	1	in	Clock input. The core only has a single clock. The Wishbone interface also works on this clock.
i_irq	1	in	Interrupt request, active high. Causes the core to switch to IRQ mode and jump to the IRQ address vector when asserted. The switch does not occur until the end of the current instruction. For example if the core is executing a stm instruction it could take 40 or 50 cycles to complete this instruction. Once the instruction has completed the core will jump to the IRQ vector and execute the instruction at that location.
i_firq	1	in	Fast Interrupt request, active high. Causes the core to switch to FIRQ mode and jump to the FIRQ address vector when asserted. Again the core makes the switch after the current instruction has completed.
i_system_rdy	1	in	Connected to the stall signal that stalls the decode and execute stages of the core. The system uses this signal to freeze the core until the DDR3 main memory initialization has completed.
Wishbone Interface			
o_wb_adr	32	out	Byte address. Note that the core only generates 26-bit instruction addresses but can generate full 32-bit data addresses.
o_wb_sel	4	out	Byte enable for writes. Bit 0 corresponds to byte 0 which is bits [7:0] on the data buses.
o_wb_we	1	out	Write enable, active high.
i_wb_dat	32	in	Read data. Active when i_wb_ack is asserted in a read cycle.
o_wb_dat	32	out	Write data. Active when o_wb_stb is high.
o_wb_cyc	1	out	Holds bus ownership during multi-cycle accesses.
o_wb_stb	1	out	Per-cycle strobe.
i_wb_ack	1	in	Used to terminate read and write accesses.
i_wb_err	1	in	Used to indicate an error on an access. Currently not used within the Amber 2 core.

Amber-23 Verilog Code Organization



Name	Description
a23_config_definesv	Defines used to configure the amber core. The number of ways in the cache is configurable. Also contains a set of debug switches which enable debug messages to be printed during simulation.
a23_localparams.v	Local parameters used in various amber source files.
a23_wishbone.v	The Wishbone interface connecting the Execute stage and Cache to the rest of the system. Instantiated in Fetch.
a23_alu.v	The arithmetic logic unit. Includes a 32-bit 2's compliment adder/subtractor as well as logical functions such as AND and XOR.
a23_functions.v	Common Verilog functions.
a23_core.v	Top-level Amber module.
a23_barrel_shifter.v	32-bit barrel shifter instantiated in Execute.
a23_cache.v	Synthesizable cache. Instantiated in Fetch. Cache misses cause the core to stall. The cache then issues a quad-word read on the wishbone bus, starting with the word that missed, and wrapping at the quad-word boundary.
a23_coprocessor.v	Co-processor 15 registers and control signals. Instantiated in Amber.
a23_decode.v	The instruction decode pipeline stage. Instantiated in Amber.
a23_decompile.v	The decompiler. This is a non-synthesizable debug module. It creates the amber.dis file which lists every instruction executed by the core.
a23_execute.v	The execute pipeline stage. Instantiated in Amber. It contains the alu, multiply, and register_bank sub-modules.
a23_fetch.v	The Fetch stage. This contains the Cache and Wishbone interface modules. It is instantiated in Amber.
a23_multiply.v	32-bit 2's compliment multiply and multiply-accumulate unit. Uses the Booth algorithm and takes 34 cycles to complete a signed multiply-accumulate operation but is quite small in logic area.
a23_register_bank.v	Contains all 27 registers r0 to r15 for each mode of operation. Registers are implemented as real flipflops in the FPGA. This allows multiple read and write access to the bank simultaneously.

ARM Instruction Set Architecture

Main features of the ARM Instruction Set

- All instructions are 32 bits long.
- Most instructions execute in a single cycle.
- Most instructions can be conditionally executed.
- A load/store architecture
  - Data processing instructions act only on registers
    - Three operand format
    - Combined ALU and shifter for high speed bit manipulation
  - Specific memory access instructions with powerful auto-indexing addressing modes.
    - 32 bit and 8 bit data types
      - and also 16 bit data types on ARM Architecture v4.
    - Flexible multiple register load and store instructions
- Instruction set extension via coprocessors
- Very dense 16-bit compressed instruction set (Thumb)

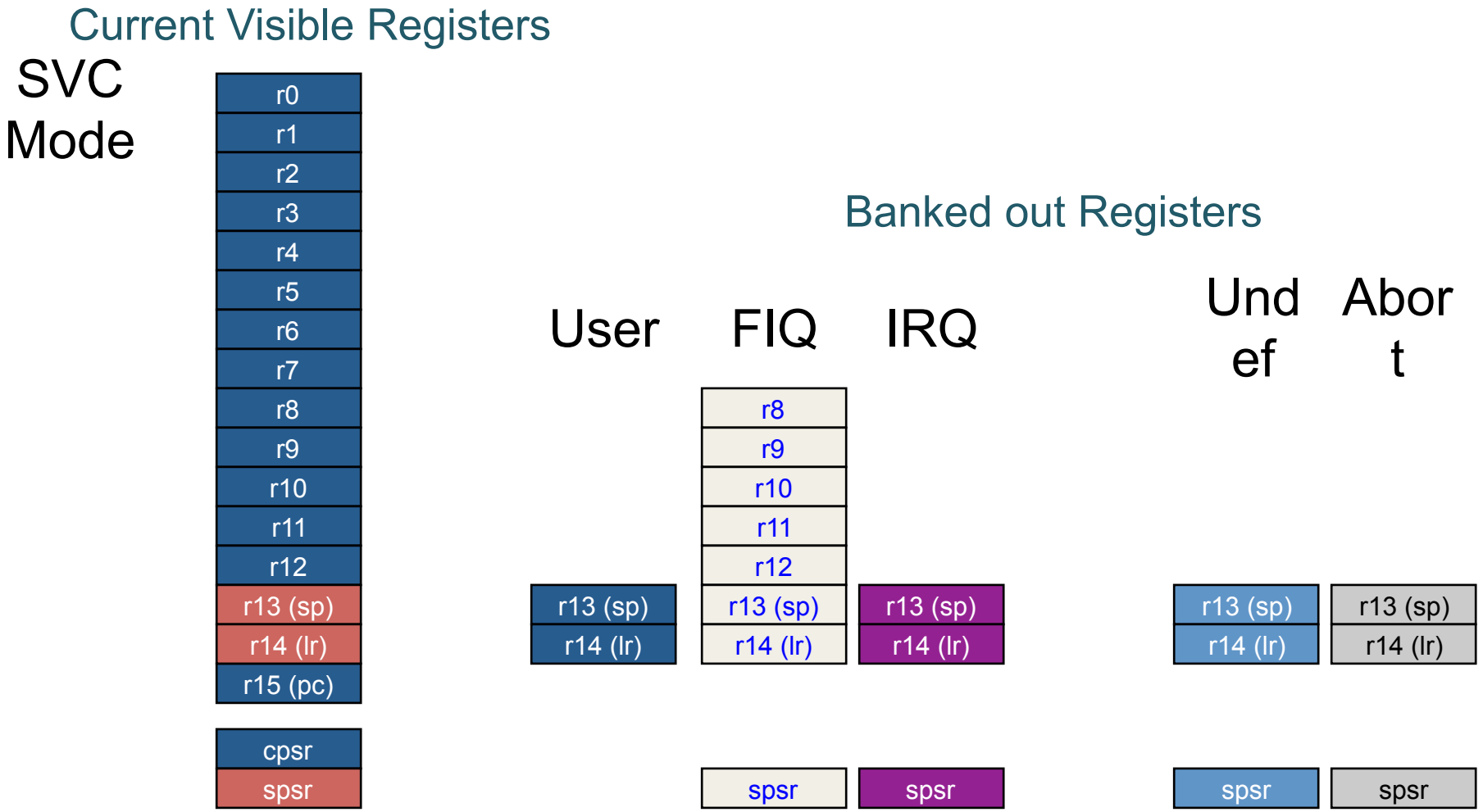
Processor Modes

- **The ARM has six operating modes:**
  - User (unprivileged mode under which most tasks run)
  - FIQ (entered when a high priority (fast) interrupt is raised)
  - IRQ (entered when a low priority (normal) interrupt is raised)
  - Supervisor (entered on reset and when a Software Interrupt instruction is executed)
  - Abort (used to handle memory access violations)
  - Undef (used to handle undefined instructions)
- **ARM Architecture Version 4 adds a seventh mode:**
  - System (privileged mode using the same registers as user mode)

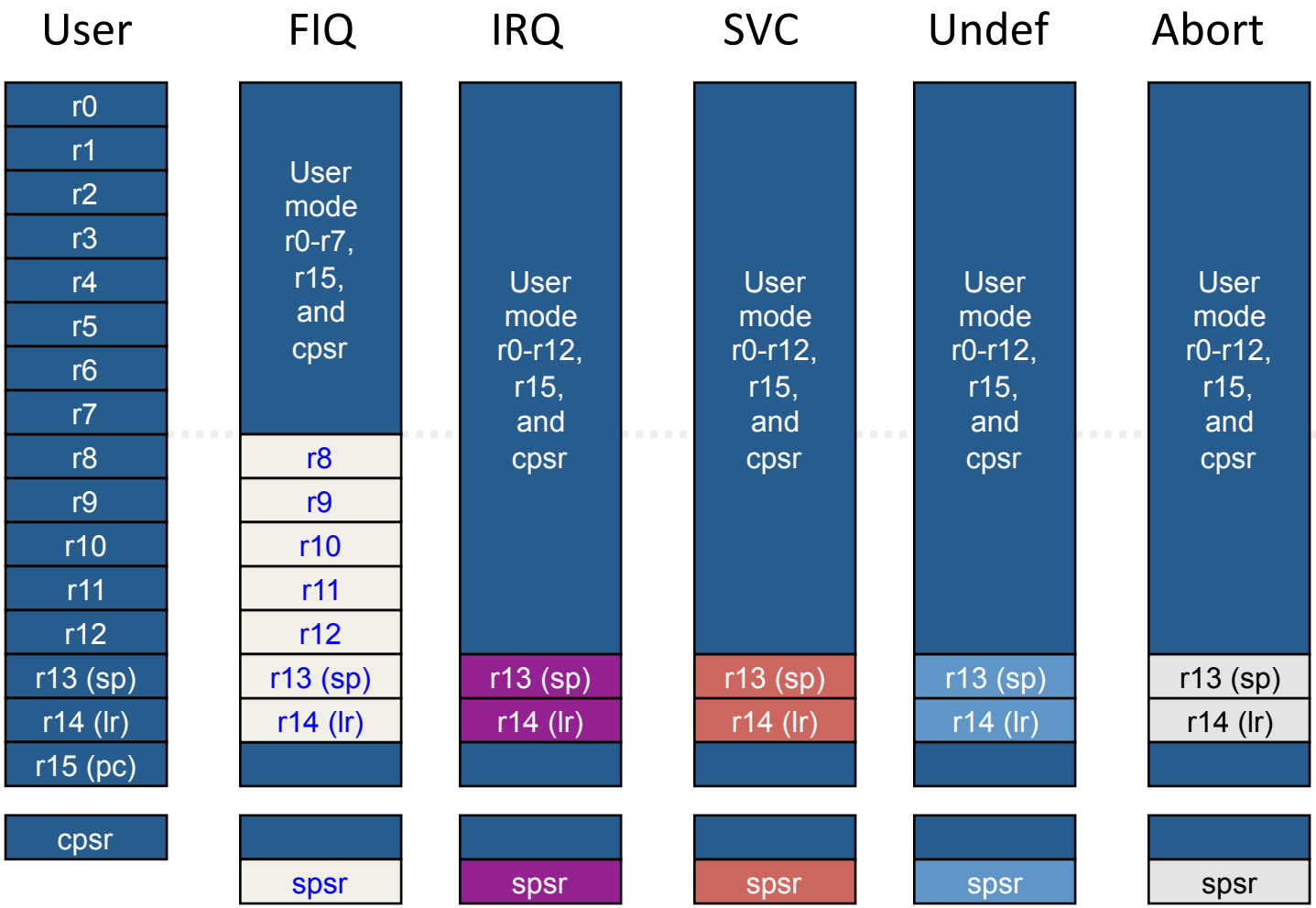
The Registers

- ARM has 37 registers in total, all of which are 32-bits long.
  - 1 dedicated program counter
  - 1 dedicated current program status register
  - 5 dedicated saved program status registers
  - 30 general purpose registers
- However these are arranged into several banks, with the accessible bank being governed by the processor mode. Each mode can access
  - a particular set of r0-r12 registers
  - a particular r13 (the stack pointer) and r14 (link register)
  - r15 (the program counter)
  - cpsr (the current program status register)
- And privileged modes can also access
  - a particular spsr (saved program status register)

The ARM Register Set



Register Organization Summary



Note: System mode uses the User mode register set

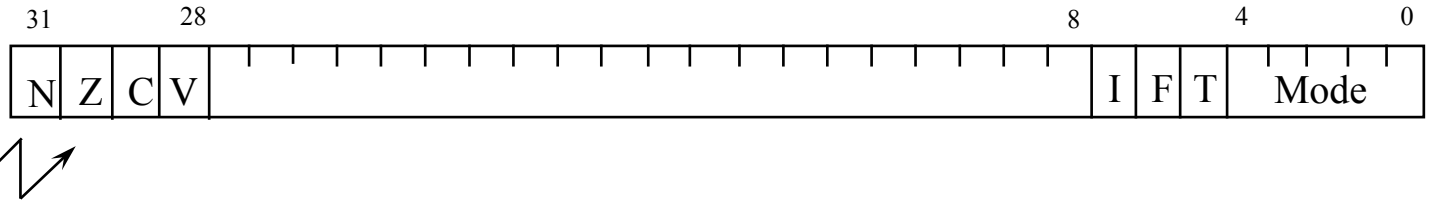
Accessing Registers using ARM Instructions

- No breakdown of currently accessible registers.
  - All instructions can access r0-r14 directly.
  - Most instructions also allow use of the PC.

- Specific instructions to allow access to CPSR and SPSR.

- Note : When in a privileged mode, it is also possible to load-store the (banked out) user mode registers to or from memory.

The Program Status Registers (CPSR and SPSRs)



Copies of the ALU status flags (latched if the instruction has the "S" bit set).

* Condition Code Flags

  N = Negative result from ALU flag.
  Z = Zero result from ALU flag.
  C = ALU operation Carried out
  V = ALU operation oVerflowed

* Mode Bits
  M[4:0] define the processor mode.

* Interrupt Disable bits.
  I = 1, disables the IRQ.
  F = 1, disables the FIQ.

* T Bit (Architecture v4T only)
  T = 0, Processor in ARM state
  T = 1, Processor in Thumb state

Condition Flags

	Logical Instruction	Arithmetic Instruction
Flag		
Negative (N='1')	No meaning	Bit 31 of the result has been set Indicates a negative number in signed operations
Zero (Z='1')	Result is all zeroes	Result of operation was zero
Carry (C='1')	After Shift operation '1' was left in carry flag	Result was greater than 32 bits
oVerflow (V='1')	No meaning	Result was greater than 31 bits Indicates a possible corruption of the sign bit in signed numbers

The Program Counter (R15) and Link Register (R14)

- When the processor is executing in ARM state:
  - All instructions are 32 bits in length
  - All instructions must be word aligned
  - Therefore the PC value is stored in bits [31:2] with bits [1:0] equal to zero (as instruction cannot be halfword or byte aligned).

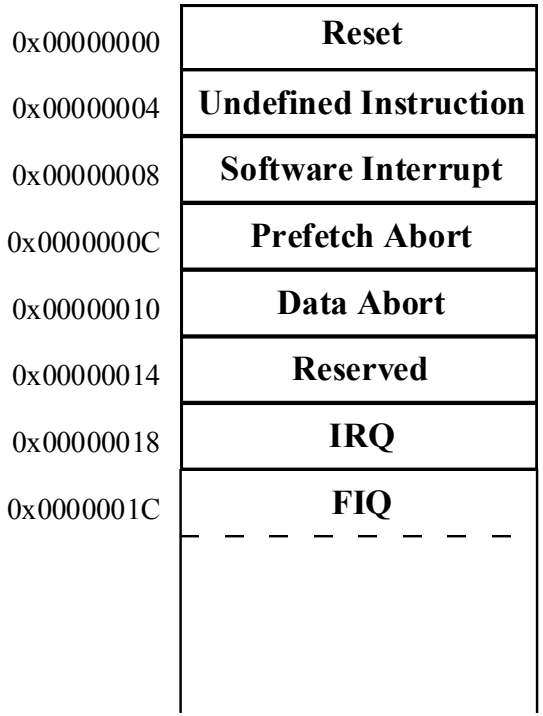
- R14 is used as the subroutine link register (LR) and stores the return address when Branch with Link operations are performed, calculated from the PC.

- Thus to return from a linked branch:

```
MOV r15,r14
   or
MOV pc,lr
```

Exception Handling and the Vector Table

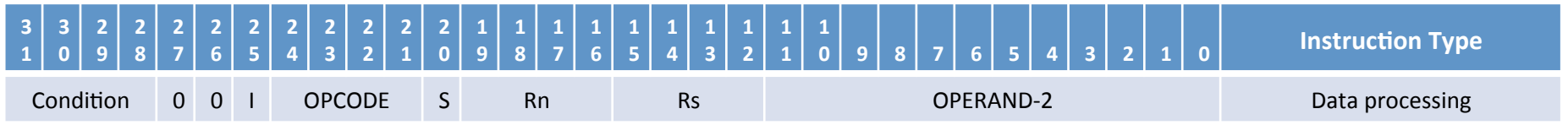
- When an exception occurs, the core:
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - If core implements ARM Architecture 4T and is currently in Thumb state, then
      - ARM state is entered.
    - Mode field bits
    - Interrupt disable flags if appropriate.
  - Maps in appropriate banked registers
  - Stores the "return address" in LR_<mode>
  - Sets PC to vector address
- To return, exception handler needs to:
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>



ARM Instruction Set Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction Type
Condition				0	0	I	OPCODE				S	Rn				Rs				OPERAND-2												Data processing
Condition				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				Multiply
Condition				0	0	0	0	1	U	A	S	Rd HIGH				Rd LOW				Rs				1	0	0	1	Rm				Long Multiply
Condition				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				Swap
Condition				0	1	I	P	U	B	W	L	Rn				Rd				OFFSET												Load/Store - Byte/Word
Condition				1	0	0	P	U	B	W	L	Rn				REGISTER LIST																Load/Store Multiple
Condition				0	0	0	P	U	1	W	L	Rn				Rd				OFFSET 1				1	S	H	1	OFFSET 2				Halfword Transfer Imm Off
Condition				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				Halfword Transfer Reg Off
Condition				1	0	1	L	BRANCH OFFSET																								Branch
Condition				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				Branch Exchange
Condition				1	1	0	P	U	N	W	L	Rn				CRd				CPNum				OFFSET								COPROCESSOR DATA XFER
Condition				1	1	1	0	Op-1				CRn				CRd				CPNum				OP-2			0	CRm				COPROCESSOR DATA OP
Condition								OP-1			L	CRn				Rd				CPNum				OP-2			1	CRm				COPROCESSOR REG XFER
Condition				1	1	1	1	SWI NUMBER																								Software Interrupt

The Condition Field



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction Type
Condition				0	0	I	OPCODE				S	Rn				Rs				OPERAND-2													Data processing

0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)

1110 = AL - always

1111 = NV - reserved.

Conditional Execution

- Most instruction sets only allow branches to be executed conditionally.
- However by reusing the condition evaluation hardware, ARM effectively increases number of instructions.
  - All instructions contain a condition field which determines whether the CPU will execute them.
  - Non-executed instructions consume 1 cycle.
    - Can't collapse the instruction like a NOP. Still have to complete cycle so as to allow fetching and decoding of the following instructions in the pipeline.
- This removes the need for many branches, which stall the pipeline (3 cycles to refill).
  - Allows very dense in-line code, without branches.
  - The TIME penalty of not executing several conditional instructions is frequently less than overhead of the branch or subroutine call that would otherwise be needed.

Using and updating the Condition Field

- To execute an instruction conditionally, simply postfix it with the appropriate condition:
  - For example an add instruction takes the form:
    - ADD r0,r1,r2 ; r0 = r1 + r2 (ADDAL)
  - To execute this only if the zero flag is set:
    - ADDEQ r0,r1,r2 ; If zero flag set then…
      ; … r0 = r1 + r2

- By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an "S".
  - For example to add two numbers and set the condition flags:
    - ADDS r0,r1,r2 ; r0 = r1 + r2 ; …
      and set flags

Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
  - This improves code density *and* performance by reducing the number of forward branch instructions.

```
  CMP    r3,#0
   BEQ    skip
   ADD    r0,r1,r2
skip
```

```
CMP    r3,#0
 ADDNE  r0,r1,r2
```

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". CMP does not need "S".

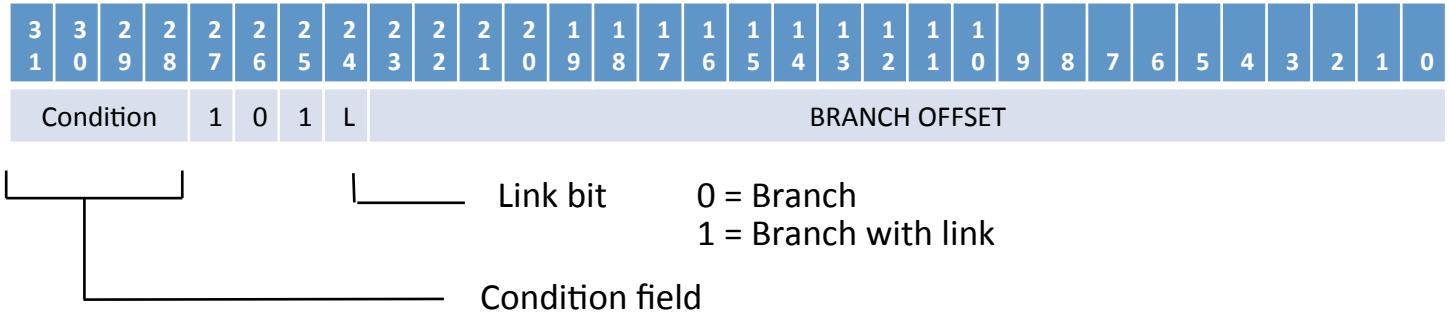
```
loop
   …
   SUBS  r1,r1,#1
   BNE  loop
```

decrement r1 and set flags

if Z flag clear then branch

Branch instructions (1)

- **Branch :** B{<cond>} label
- **Branch with Link :** BL{<cond>} sub_routine_label



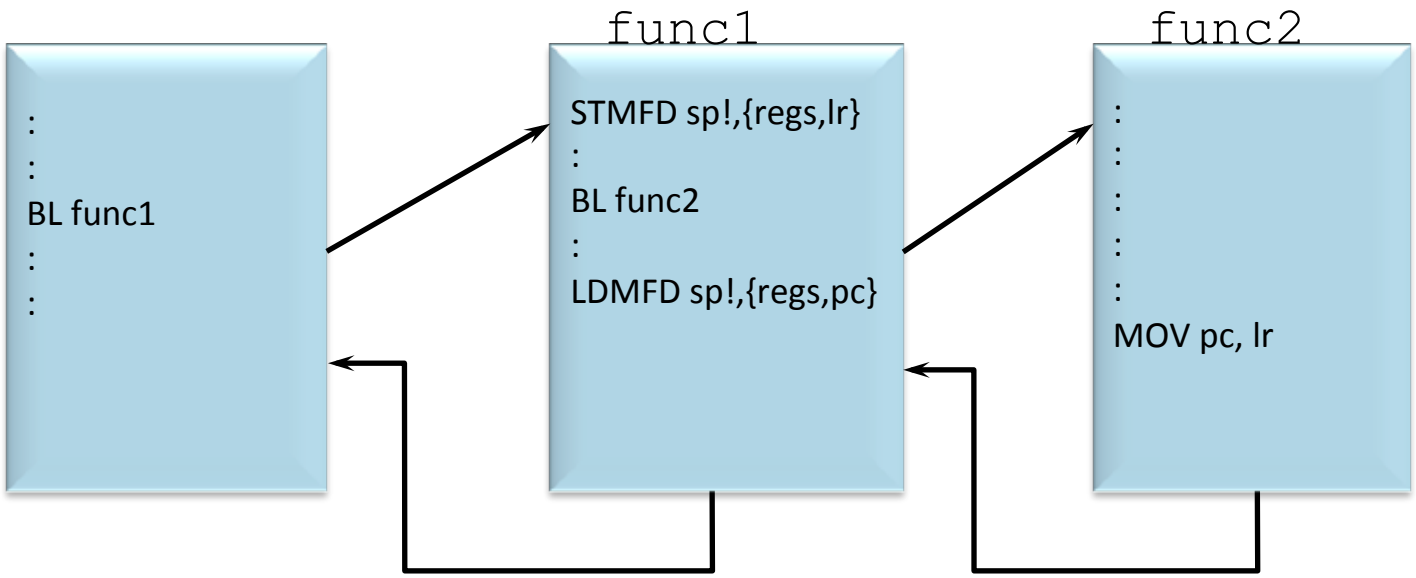
- **The offset for branch instructions is calculated by the assembler:**
  - By taking the difference between the branch instruction and the target address minus 8 (to allow for the pipeline).
  - This gives a 26 bit offset which is right shifted 2 bits (as the bottom two bits are always zero as instructions are word – aligned) and stored into the instruction encoding.
  - This gives a range of ± 32 Mbytes.

Branch instructions (2)

- When executing the instruction, the processor:
  - shifts the offset left two bits, sign extends it to 32 bits, and adds it to PC.
- Execution then continues from the new PC, once the pipeline has been refilled.
- The "Branch with link" instruction implements a subroutine call by writing PC-4 into the LR of the current bank.
  - i.e. the address of the next instruction following the branch with link (allowing for the pipeline).
- To return from subroutine, simply need to restore the PC from the LR:
  - MOV pc, lr
  - Again, pipeline has to refill before execution continues.

Branch instructions (3)

- The "Branch" instruction does not affect LR.
- Note: Architecture 4T offers a further ARM branch instruction, BX
  - See Thumb Instruction Set Module for details.
- BL <subroutine>
  - Stores return address in LR
  - Returning implemented by restoring the PC from LR
  - For non-leaf functions, LR will have to be stacked



Conditional Branches

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Data processing Instructions

- Largest family of ARM instructions, all sharing the same instruction format.
- Contains:
  - Arithmetic operations
  - Comparisons (no results - just set condition codes)
  - Logical operations
  - Data movement between registers
- Remember, this is a load / store architecture
  - These instruction only work on registers, NOT memory.
- They each perform a specific operation on one or two operands.
  - First operand always a register - Rn
  - Second operand sent to the ALU via barrel shifter.
- We will examine the barrel shifter shortly.

Arithmetic Operations

- **Operations are:**
  - ADD operand1 + operand2 ; Add
  - ADC operand1 + operand2 + carry ; Add with carry
  - SUB operand1 - operand2 ; Subtract
  - SBC operand1 - operand2 + carry -1 ; Subtract with carry
  - RSB operand2 - operand1 ; Reverse subtract
  - RSC operand2 - operand1 + carry - 1 ; Reverse subtract with carry
- **Syntax:**
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- **Examples**
  - ADD r0, r1, r2
  - SUBGT r3, r3, #1
  - RSBLES r4, r5, #5

Comparisons

- The only effect of the comparisons is to update the condition flags. Thus no need to set S bit.
- Operations are:
  - CMP operand1 - operand2 ; Compare
  - CMN operand1 + operand2 ; Compare negative
  - TST operand1 AND operand2 ; Test
  - TEQ operand1 EOR operand2 ; Test equivalence
- Syntax:
  - <Operation>{<cond>} Rn, Operand2
- Examples:
  - CMP r0, r1
  - TSTEQ r2, #5

Logical Operations

- **Operations are:**

  AND operand1 AND operand2
  EOR operand1 EOR operand2
  ORR operand1 OR operand2
  ORN operand1 NOR operand2
  BIC operand1 AND NOT operand2 [ie bit clear]

- **Syntax:**
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- **Examples:**

```
AND   r0, r1, r2
BICEQ r2, r3, #7
EORS  r1,r3,r0
```

Data Movement

- **Operations are:**
  MOV operand2
  MVN NOT operand2

  Note that these make no use of operand1.
- **Syntax:**
  - **<Operation>{<cond>}{S} Rd, Operand2**
- **Examples:**

```
MOV     r0, r1
MOVS    r2, #10
MVNEQ   r1,#0
```

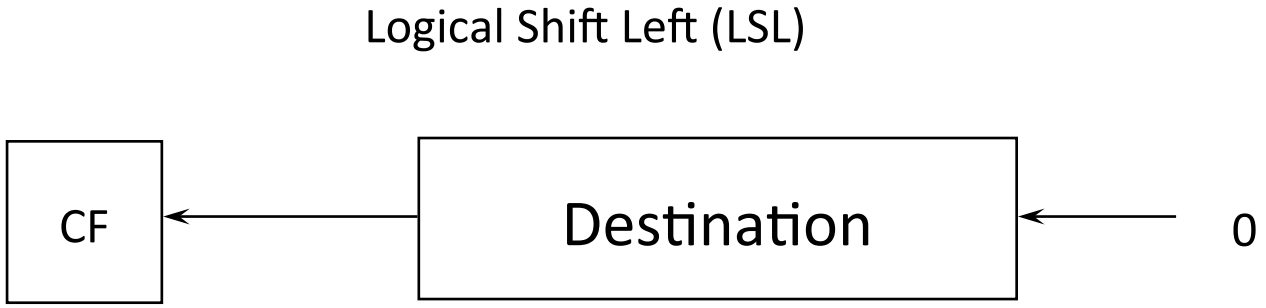
The Barrel Shifter

- The ARM doesn't have actual shift instructions.
- Instead it has a barrel shifter which provides a mechanism to carry out shifts as part of other instructions.
- So what operations does the barrel shifter support?

Barrel Shifter - Left Shift

- **Shifts left by the specified amount (multiplies by powers of two) e.g.**

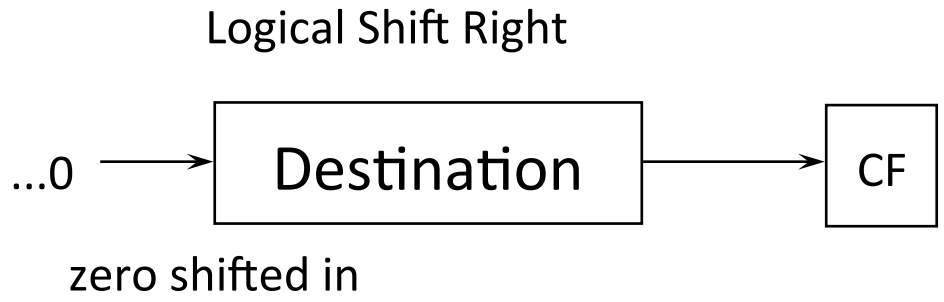
  LSL #5 => multiply by 32



Barrel Shifter - Right Shifts

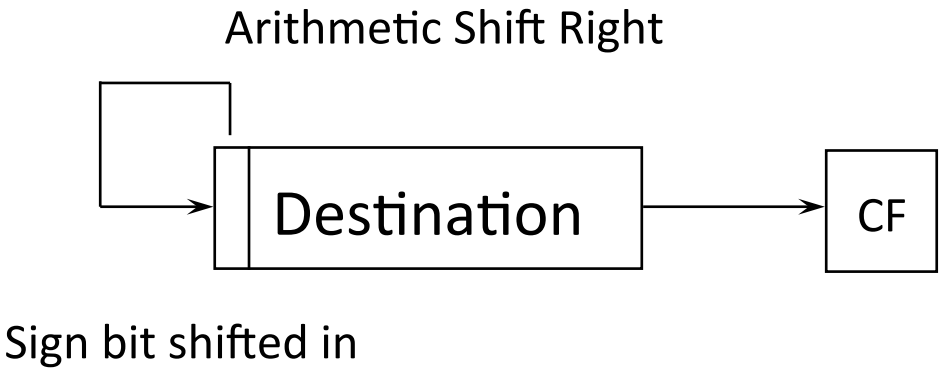
Logical Shift Right (LSR)
Shifts right by the specified amount (divides by powers of two) e.g.

LSR #5 = divide by 32



Arithmetic Shift Right (ASR)
Shifts right (divides by powers of two) and preserves the sign bit, for 2's complement operations. e.g.

ASR #5 = divide by 32



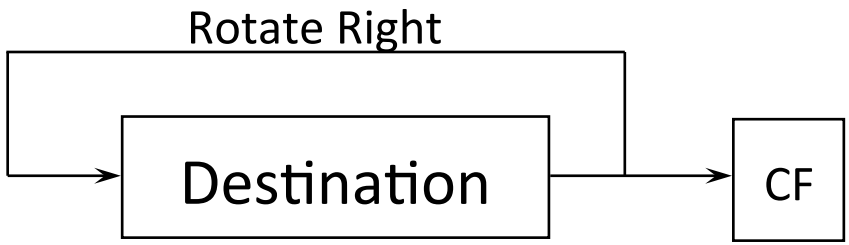
Barrel Shifter - Rotations

Rotate Right (ROR)

Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB.

e.g. ROR #5

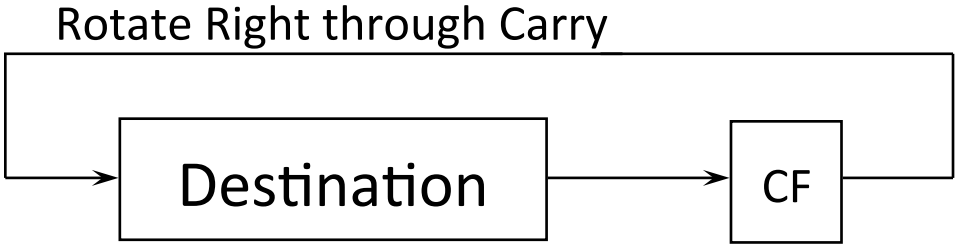
Note the last bit rotated is also used as the Carry Out.



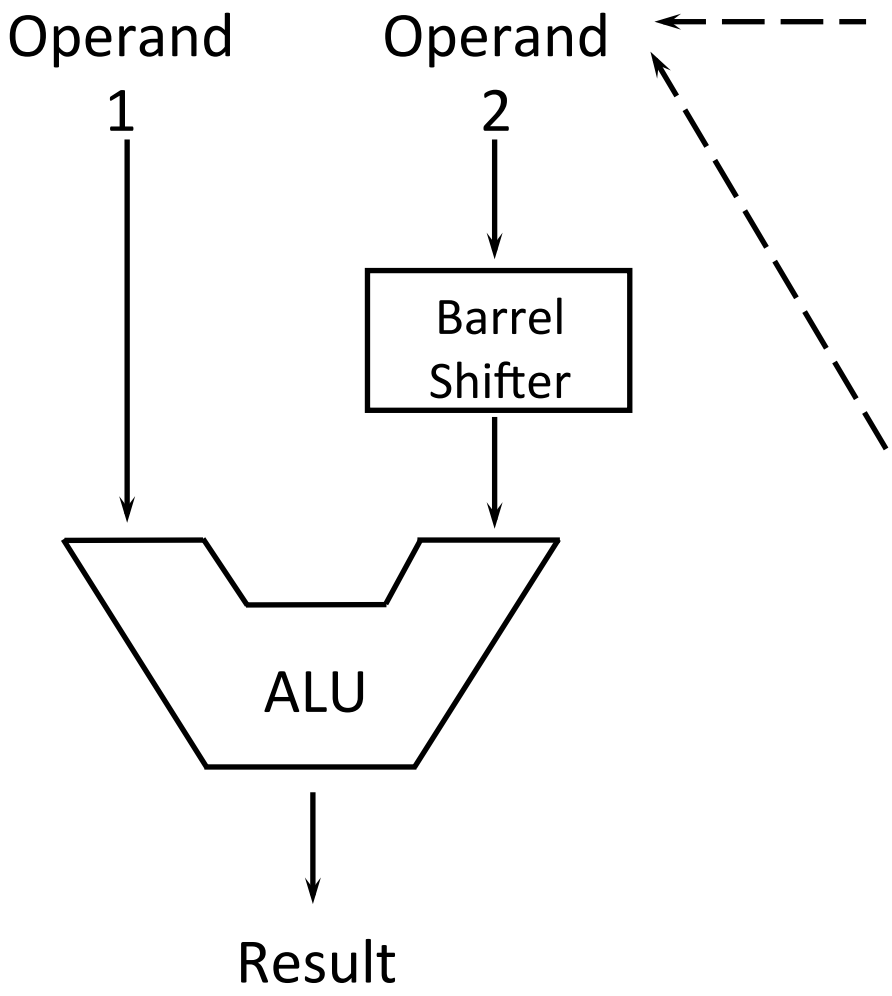
Rotate Right Extended (RRX)

This operation uses the CPSR C flag as a 33rd bit.

Rotates right by 1 bit. Encoded as ROR #0



Using the Barrel Shifter: The Second Operand



- **Register, optionally with shift operation applied.**
- **Shift value can be either be:**
  - **5 bit unsigned integer**
  - **Specified in bottom byte of another register.**

* Immediate value

- 8 bit number
- Can be rotated right through an even number of positions.
- Assembler will calculate rotate for you from ~~constant~~.

Second Operand : Shifted Register

- The amount by which the register is to be shifted is contained in either:
  - the immediate 5-bit field in the instruction
    - NO OVERHEAD
    - Shift is done for free - executes in single cycle.
  - the bottom byte of a register (not PC)
    - Then takes extra cycle to execute
    - ARM doesn't have enough read ports to read 3 registers at once.
    - Then same as on other processors where shift is separate instruction.
- If no shift is specified then a default shift is applied: LSL #0
  - i.e. barrel shifter has no effect on value in register.

Second Operand: Using a Shifted Register

- Using a multiplication instruction to multiply by a constant means first loading the constant into a register and then waiting a number of internal cycles for the instruction to complete.
- A more optimum solution can often be found by using some combination of MOVs, ADDs, SUBs and RSBs with shifts.
  - Multiplications by a constant equal to a ((power of 2) ± 1) can be done in one cycle.

```
MOV R2, R0, LSL #2        ; Shift R0 left by 2, write to R2, (R2=R0x4)
ADD R9, R5, R5, LSL #3    ; R9 = R5 + R5 x 8 or R9 = R5 x 9
RSB R9, R5, R5, LSL #3    ; R9 = R5 x 8 - R5 or R9 = R5 x 7
SUB R10, R9, R8, LSR #4 ; R10 = R9 - R8 / 16
MOV R12, R4, ROR R3     ; R12 = R4 rotated right by value of R3
```

Second Operand: Immediate Value (1)

- There is no single instruction which will load a 32 bit immediate constant into a register without performing a data load from memory.
  - All ARM instructions are 32 bits long
  - ARM instructions do not use the instruction stream as data.
- The data processing instruction format has 12 bits available for operand2
  - If used directly this would only give a range of 4096.
- Instead it is used to store 8 bit constants, giving a range of 0 - 255.
- These 8 bits can then be rotated right through an even number of positions (ie RORs by 0, 2, 4,..30).
  - This gives a much larger range of constants that can be directly loaded, though some constants will still need to be loaded from memory.

Second Operand: Immediate Value (2)

- This gives us:
  - 0 - 255 [0 - 0xff]
  - 256,260,264,..,1020 [0x100-0x3fc, step 4, 0x40-0xff ror 30]
  - 1024,1040,1056,..,4080 [0x400-0xff0, step 16, 0x40-0xff ror 28]
  - 4096,4160, 4224,..,16320 [0x1000-0x3fc0, step 64, 0x40-0xff ror 26]
- These can be loaded using, for example:
  - MOV r0, #0x40, 26 ; => MOV r0, #0x1000 (ie 4096)
- To make this easier, the assembler will convert to this form for us if simply given the required constant:
  - MOV r0, #4096 ; => MOV r0, #0x1000 (ie 0x40 ror 26)
- The bitwise complements can also be formed using MVN:
  - MOV r0, #0xFFFFFFFF ; assembles to MVN r0, #0
- If the required constant cannot be generated, an error will be reported.

Loading full 32 bit constants

- Although the MOV/MVN mechanism will load a large range of constants into a register, sometimes this mechanism will not generate the required constant.
- Therefore, the assembler also provides a method which will load ANY 32 bit constant:
  - LDR rd,=numeric constant
- If the constant can be constructed using either a MOV or MVN then this will be the instruction actually generated.
- Otherwise, the assembler will produce an LDR instruction with a PC-relative address to read the constant from a literal pool.

```
LDR r0,=0x42              ; generates   MOV r0,#0x42
LDR r0,=0x55555555        ; generate    LDR r0,[pc, offset to DCD]
                                        :
                                        :
DCD  0x55555555           ;  Constant in memory
```

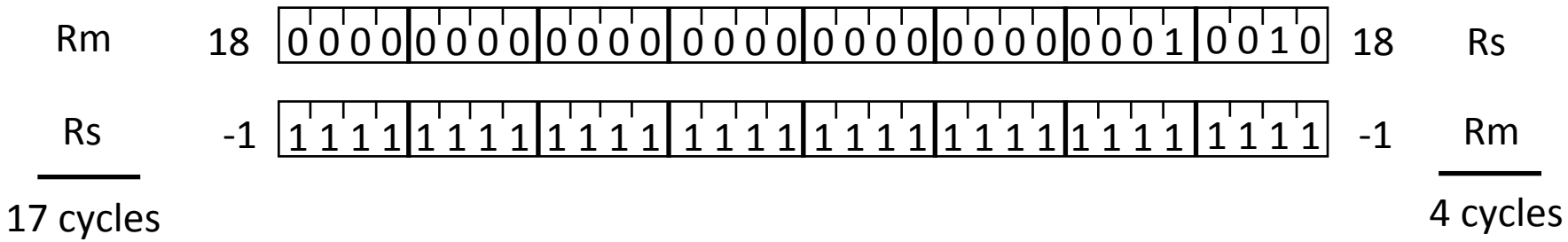
- As this mechanism will always generate the best instruction for a given case, it is the recommended way of loading constants.

Multiplication Instructions

- The Basic ARM provides two multiplication instructions.
- Multiply
  - MUL{<cond>}{S} Rd, Rm, Rs ; Rd = Rm * Rs
- Multiply Accumulate - does addition for free
  - MLA{<cond>}{S} Rd, Rm, Rs,Rn ; Rd = (Rm * Rs) + Rn
- Restrictions on use:
  - Rd and Rm cannot be the same register
    - Can be avoided by swapping Rm and Rs around. This works because multiplication is commutative.
  - Cannot use PC.
- These will be picked up by the assembler if overlooked.
- Operands can be considered signed or unsigned
  - Up to user to interpret correctly.

Multiplication Implementation

- The ARM makes use of Booth's Algorithm to perform integer multiplication.
- On non-M ARMs this operates on 2 bits of Rs at a time.
  - For each pair of bits this takes 1 cycle (plus 1 cycle to start with).
  - However when there are no more 1's left in Rs, the multiplication will early-terminate.
- Example: Multiply 18 and -1 : Rd = Rm * Rs



- Note: Compiler does not use early termination criteria to decide on which order to place operands.

Extended Multiply Instructions

- M variants of ARM cores contain extended multiplication hardware. This provides three enhancements:
  - An 8 bit Booth's Algorithm is used
    - Multiplication is carried out faster (maximum for standard instructions is now 5 cycles).
  - Early termination method improved so that now completes multiplication when all remaining bit sets contain
    - all zeroes (as with non-M ARMs), or
    - all ones.
  - Thus the previous example would early terminate in 2 cycles in both cases.
  - 64 bit results can now be produced from two 32bit operands
    - Higher accuracy.
    - Pair of registers used to store result.

Multiply-Long & Multiply-Accumulate Long

- Instructions are
  - MULL which gives RdHi,RdLo:=Rm*Rs
  - MLAL which gives RdHi,RdLo:=(Rm*Rs)+RdHi,RdLo
- However the full 64 bit of the result now matter (lower precision multiply instructions simply throws top 32bits away)
  - Need to specify whether operands are signed or unsigned
- Therefore syntax of new instructions are:
  - UMULL{<cond>}{S} RdLo,RdHi,Rm,Rs
  - UMLAL{<cond>}{S} RdLo,RdHi,Rm,Rs
  - SMULL{<cond>}{S} RdLo, RdHi, Rm, Rs
  - SMLAL{<cond>}{S} RdLo, RdHi, Rm, Rs
- Not generated by the compiler.
- Warning : Unpredictable on non-M ARMs.

Load / Store Instructions

- **The ARM is a Load / Store Architecture:**
  - Does not support memory to memory data processing operations.
  - Must move data values into registers before using them.
- **This might sound inefficient, but in practice it isn't:**
  - Load data values from memory into registers.
  - Process data in registers using a number of data processing instructions which are not slowed down by memory access.
  - Store results from registers out to memory.
- **The ARM has three sets of instructions which interact with main memory. These are:**
  - Single register data transfer (LDR / STR).
  - Block data transfer (LDM/STM).
  - Single Data Swap (SWP).

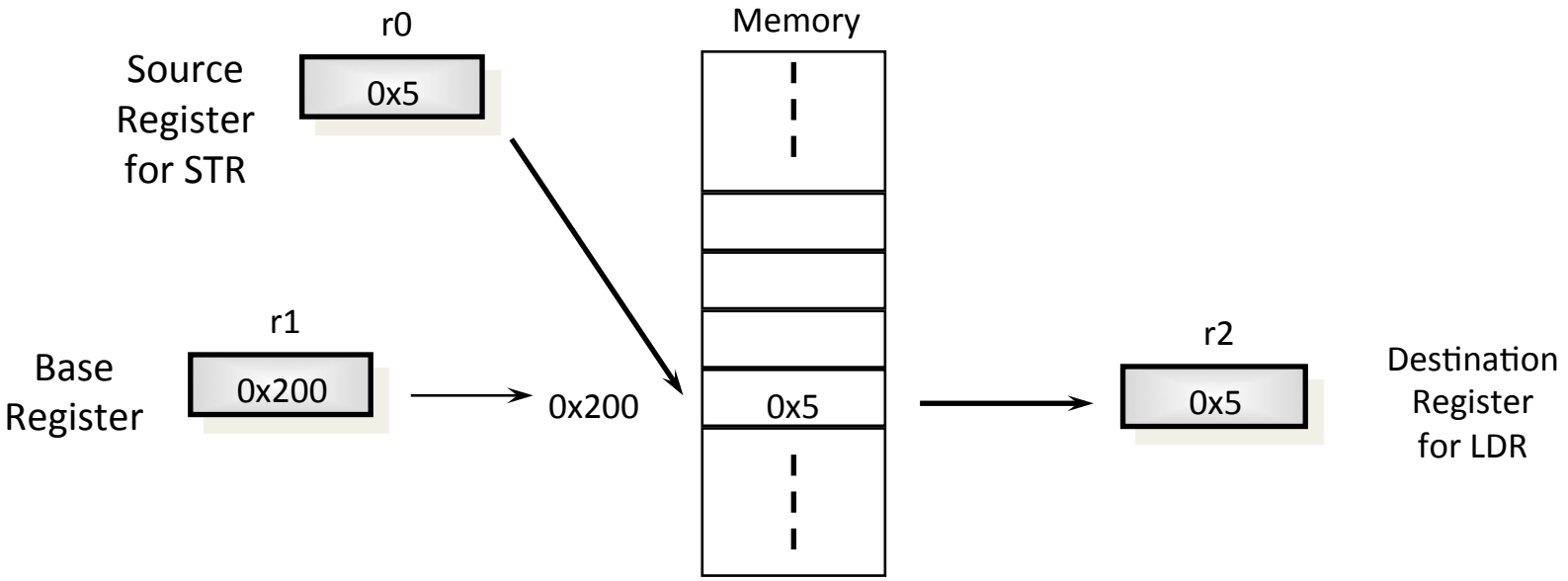
Single register data transfer

- The basic load and store instructions are:
  - Load and Store Word or Byte
    - LDR / STR / LDRB / STRB
- ARM Architecture Version 4 also adds support for Halfwords and signed data.
  - Load and Store Halfword
    - LDRH / STRH
  - Load Signed Byte or Halfword - load value and sign extend it to 32 bits.
    - LDRSB / LDRSH
- All of these instructions can be conditionally executed by inserting the appropriate condition code after STR / LDR.
  - e.g. LDREQB
- Syntax:
  - <LDR|STR>{<cond>}{<size>} Rd, <address>

Load and Store Word or Byte: Base Register

- The memory location to be accessed is held in a base register

```
STR r0, [r1]        ; Store contents of r0 to location pointed to
                    ; by contents of r1.
LDR r2, [r1]        ; Load r2 with contents of memory location
                    ; pointed to by contents of r1.
```

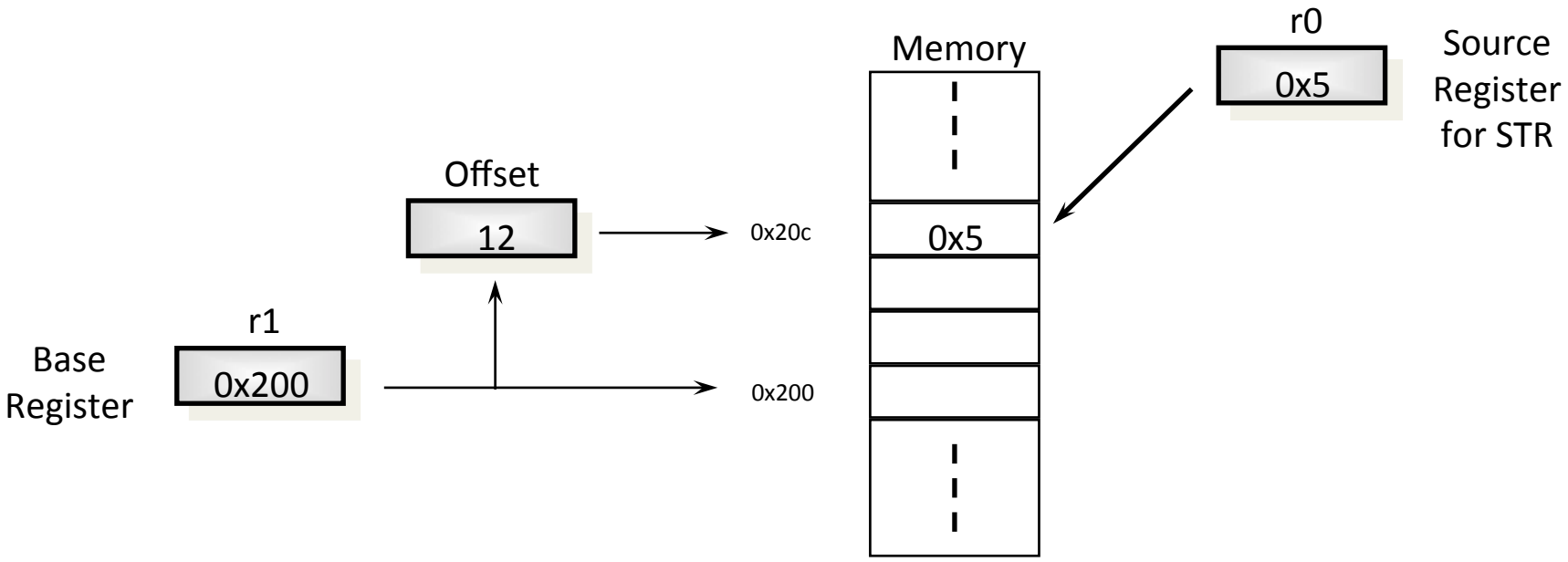


Load/Store Word or Byte: Offsets from the Base Register

- As well as accessing the actual location contained in the base register, these instructions can access a location offset from the base register pointer.
- This offset can be
  - An unsigned 12bit immediate value (ie 0 - 4095 bytes).
  - A register, optionally shifted by an immediate value
- This can be either added or subtracted from the base register:
  - Prefix the offset value or register with '+' (default) or '-'.
- This offset can be applied:
  - before the transfer is made: *Pre-indexed addressing*
    - optionally auto-incrementing the base register, by postfixing the instruction with an '!'.
  - after the transfer is made: *Post-indexed addressing*
    - causing the base register to be auto-incremented.

Load/Store Word or Byte: Pre-indexed Addressing

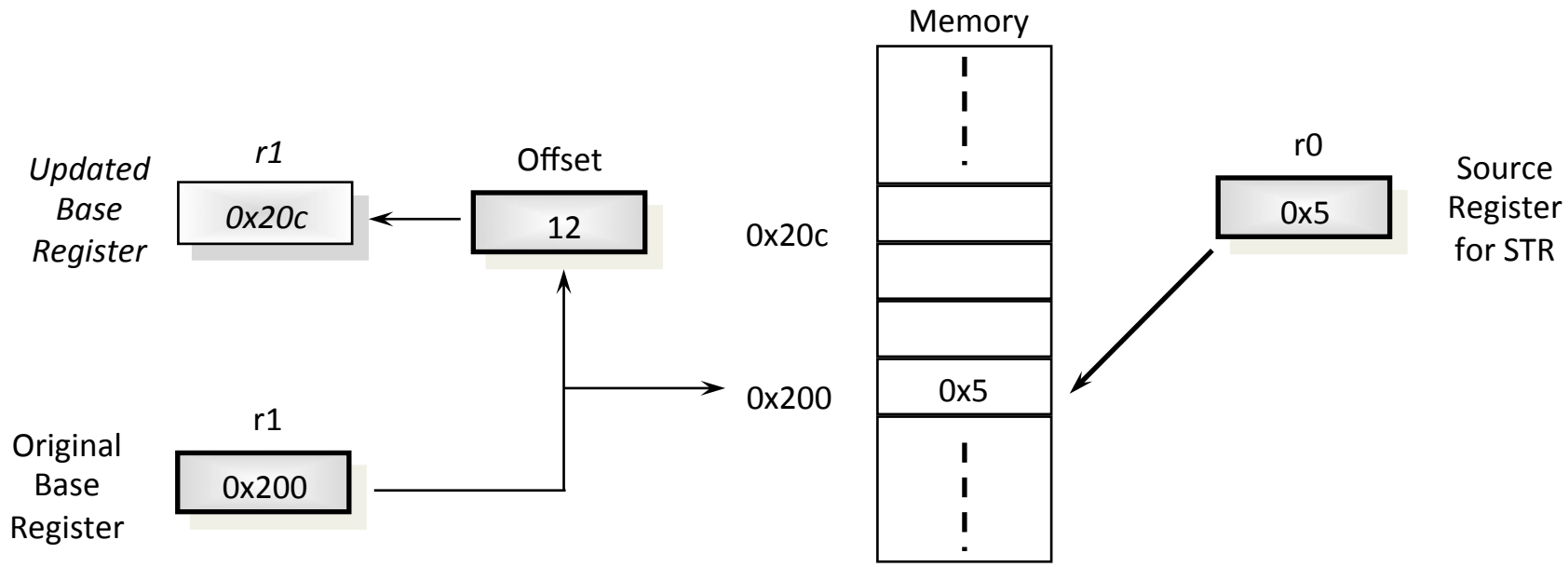
- **Example: STR r0, [r1,#12]**



  - To store to location 0x1f4 instead use: STR r0, [r1,#-12]
  - To auto-increment base pointer to 0x20c use: STR r0, [r1, #12]!
  - If r2 contains 3, access 0x20c by multiplying this by 4:
    - STR r0, [r1, r2, LSL #2]

Load and Store Word or Byte: Post-indexed Addressing

- **Example: STR r0, [r1], #12**



  - To auto-increment the base register to location 0x1f4 instead use:
    - STR r0, [r1], #-12
  - If r2 contains 3, auto-increment base register to 0x20c by multiplying this by 4:
    - STR r0, [r1], r2, LSL #2

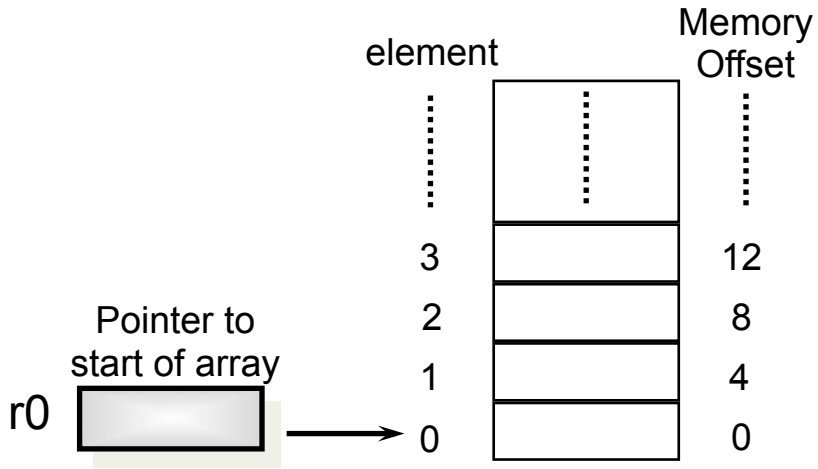
Load and Stores with User Mode Privilege

- When using post-indexed addressing, there is a further form of Load/Store Word/Byte:
  - <LDR|STR>{<cond>}{B}T Rd, <post_indexed_address>

- When used in a privileged mode, this does the load/store with user mode privilege.
  - Normally used by an exception handler that is emulating a memory access instruction that would normally execute in user mode.

Example Usage of Addressing Modes

- Imagine an array, the first element of which is pointed to by the contents of r0.
- If we want to access a particular element, then we can use pre-indexed addressing:
  - r1 is element we want.
  - LDR r2, [r0, r1, LSL #2]



- If we want to step through every element of the array, for instance to produce sum of elements in the array, then we can use post-indexed addressing within a loop:
  - r1 is address of current element (initially equal to r0).
  - LDR r2, [r1], #4
- Use a further register to store the address of final element, so that the loop can be correctly terminated.

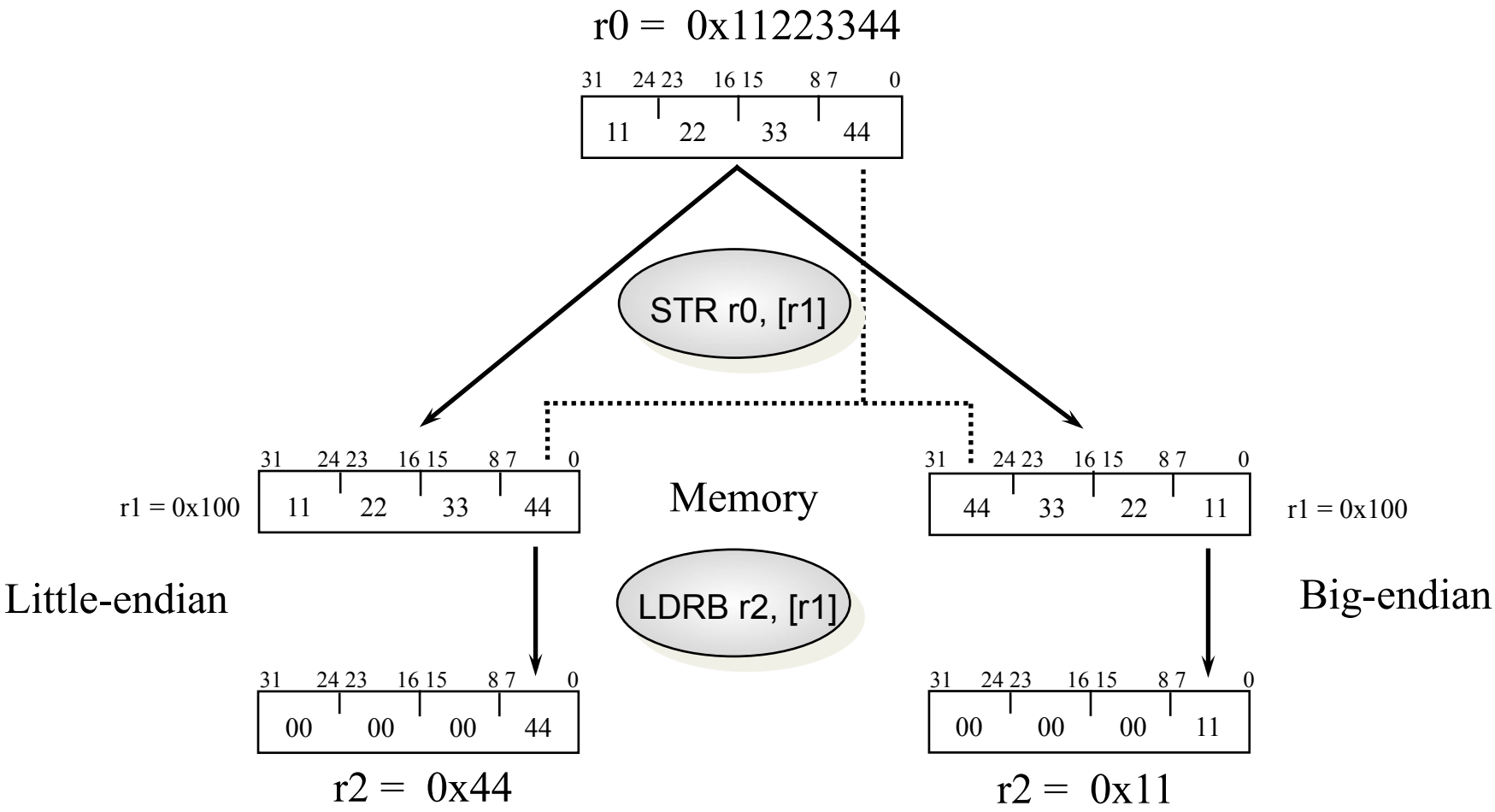
Offsets for Halfword and Signed Halfword / Byte Access

- The Load and Store Halfword and Load Signed Byte or Halfword instructions can make use of pre- and post-indexed addressing in much the same way as the basic load and store instructions.
- However the actual offset formats are more constrained:
  - The immediate value is limited to 8 bits (rather than 12 bits) giving an offset of 0-255 bytes.
  - The register form cannot have a shift applied to it.

Effect of endianess

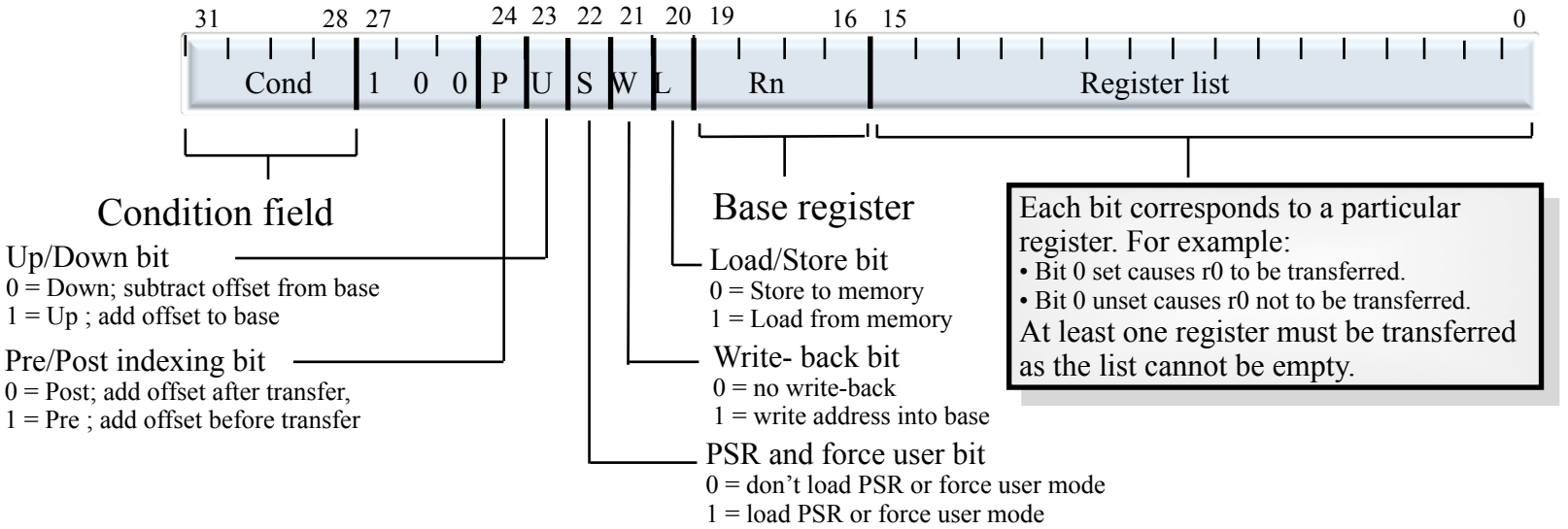
- The ARM can be set up to access its data in either little or big endian format.
- Little endian:
  - Least significant byte of a word is stored in bits 0-7 of an addressed word.
- Big endian:
  - Least significant byte of a word is stored in bits 24-31 of an addressed word.
- This has no real relevance unless data is stored as words and then accessed in smaller sized quantities (halfwords or bytes).
  - Which byte / halfword is accessed will depend on the endianess of the system involved.

YA Endianess Example



Block Data Transfer (1)

- The Load and Store Multiple instructions (LDM / STM) allow betweeen 1 and 16 registers to be transferred to or from memory.
- The transferred registers can be either:
  - Any subset of the current bank of registers (default).
  - Any subset of the user mode bank of registers when in a priviledged mode (postfix instruction with a '^').

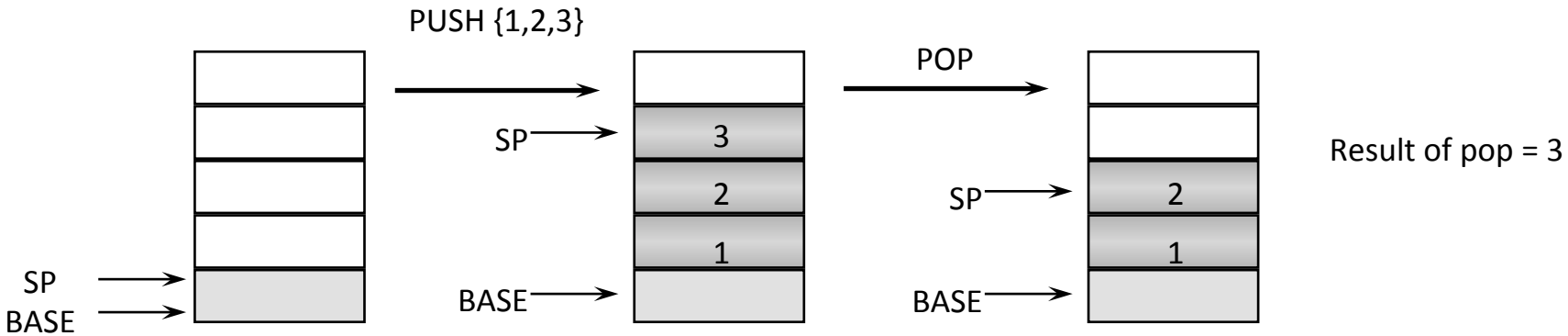


Block Data Transfer (2)

- Base register used to determine where memory access should occur.
  - 4 different addressing modes allow increment and decrement inclusive or exclusive of the base register location.
  - Base register can be optionally updated following the transfer (by appending it with an '!'.
  - Lowest register number is always transferred to/from lowest memory location accessed.
- These instructions are very efficient for
  - Saving and restoring context
    - For this useful to view memory as a stack.
  - Moving large blocks of data around memory
    - For this useful to directly represent functionality of the instructions.

Stacks

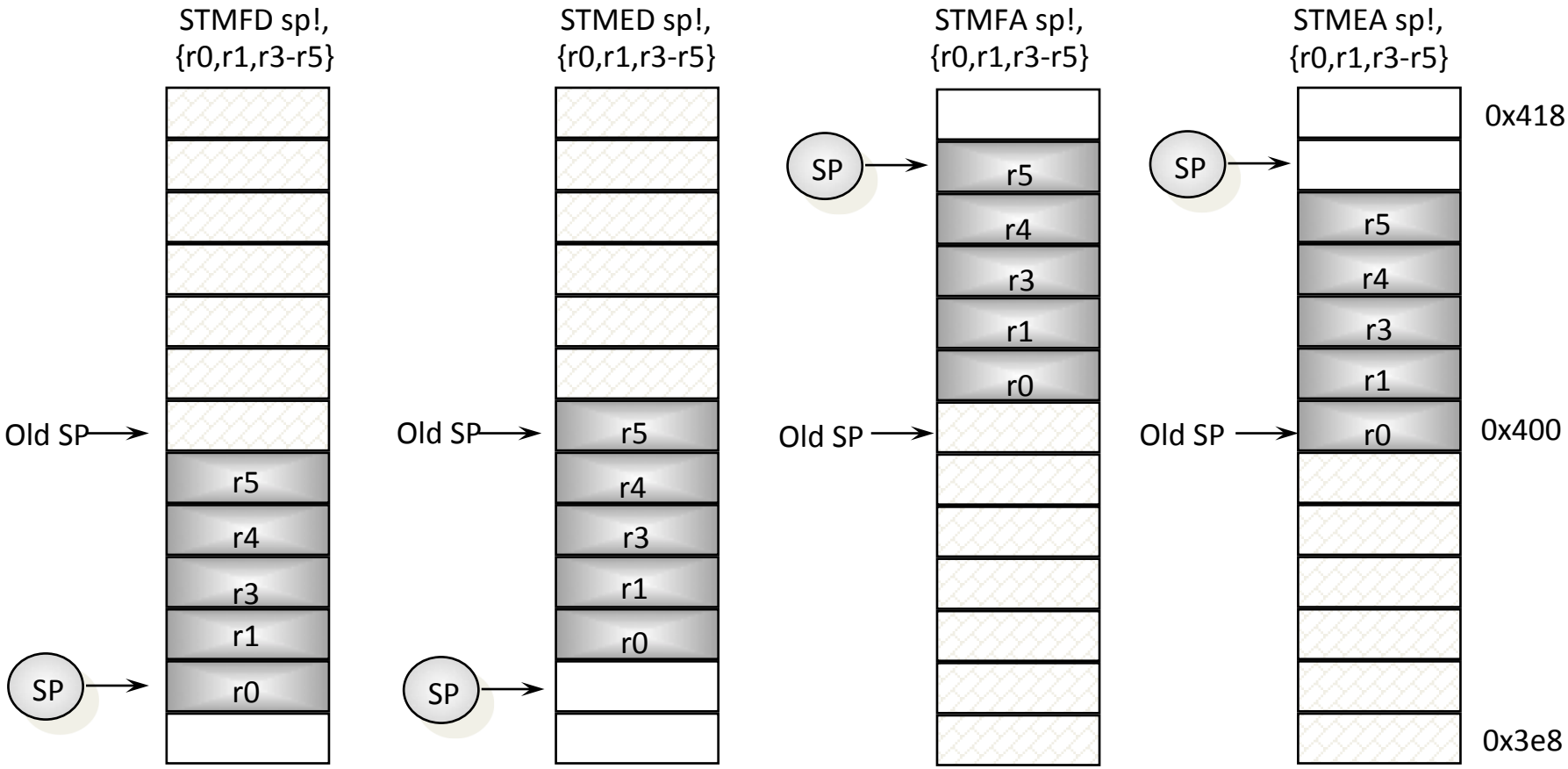
- A stack is an area of memory which grows as new data is "pushed" onto the "top" of it, and shrinks as data is "popped" off the top.
- Two pointers define the current limits of the stack.
  - A base pointer
    - used to point to the "bottom" of the stack (the first location).
  - A stack pointer
    - used to point the current "top" of the stack.



Stack Operation

- Traditionally, a stack grows down in memory, with the last "pushed" value at the lowest address. The ARM also supports ascending stacks, where the stack structure grows up through memory.
- The value of the stack pointer can either:
  - Point to the last occupied address (Full stack)
    - and so needs pre-decrementing (ie before the push)
  - Point to the next occupied address (Empty stack)
    - and so needs post-decrementing (ie after the push)
- The stack type to be used is given by the postfix to the instruction:
  - STMFD / LDMFD : Full Descending stack
  - STMFA / LDMFA : Full Ascending stack.
  - STMED / LDMED : Empty Descending stack
  - STMEA / LDMEA : Empty Ascending stack
- Note: ARM Compiler will always use a Full descending stack.

Stack Examples



Stacks and Subroutines

- One use of stacks is to create temporary register workspace for subroutines. Any registers that are needed can be pushed onto the stack at the start of the subroutine and popped off again at the end so as to restore them before return to the caller :

```
STMFD sp!,{r0-r12, lr}          ; stack all registers
........                        ; and the return address
........
LDMFD sp!,{r0-r12, pc}          ; load all the registers
                                ; and return automatically
```

- See the chapter on the ARM Procedure Call Standard in the SDT Reference Manual for further details of register usage within subroutines.
- If the pop instruction also had the 'S' bit set (using '^') then the transfer of the PC when in a privileged mode would also cause the SPSR to be copied into the CPSR (see exception handling module).

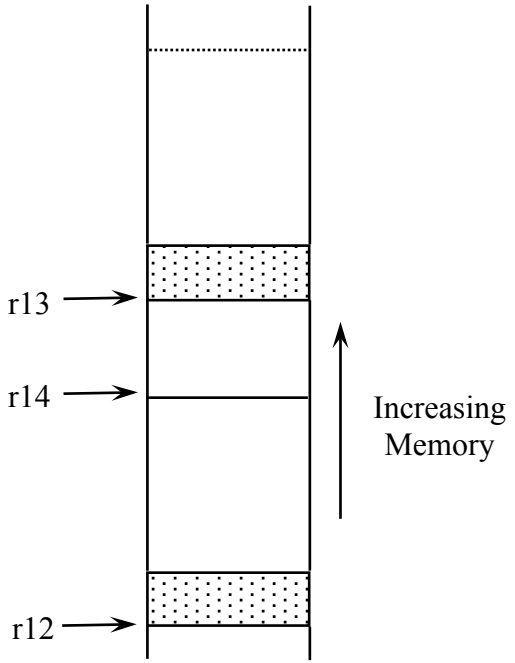
Direct functionality of Block Data Transfer

- When LDM / STM are not being used to implement stacks, it is clearer to specify exactly what functionality of the instruction is:
  - i.e. specify whether to increment / decrement the base pointer, before or after the memory access.
- In order to do this, LDM / STM support a further syntax in addition to the stack one:
  - STMIA / LDMIA : Increment After
  - STMIB / LDMIB : Increment Before
  - STMDA / LDMDA : Decrement After
  - STMDB / LDMDB : Decrement Before

Example: Block Copy

- **Copy a block of memory, which is an exact multiple of 12 words long from the location pointed to by r12 to the location pointed to by r13. r14 points to the end of block to be copied.**

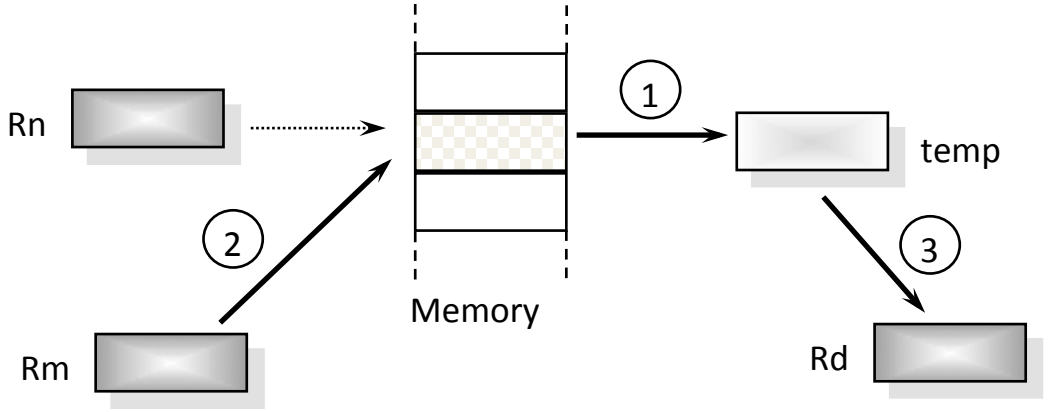
```
; r12 points to the start of the source data
; r14 points to the end of the source data
; r13 points to the start of the destination data
loop    LDMIA   r12!, {r0-r11} ; load 48 bytes
        STMIA   r13!, {r0-r11} ; and store them
        CMP     r12, r14       ; check for the end
        BNE     loop           ; and loop until done
```



  - **This loop transfers 48 bytes in 31 cycles**
  - **Over 50 Mbytes/sec at 33 MHz**

Swap and Swap Byte Instructions

- Atomic operation of a memory read followed by a memory write which moves byte or word quantities between registers and memory.
- Syntax:
  - SWP{<cond>}{B} Rd, Rm, [Rn]



- To implement an actual swap of contents make Rd = Rm.
- The compiler cannot produce this instruction.

Software Interrupt (SWI)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction Type
Condition				1	1	1	1	SWI NUMBER																								Software Interrupt

- In effect, a SWI is a user-defined instruction.
- It causes an exception trap to the SWI hardware vector (thus causing a change to supervisor mode, plus the associated state saving), thus causing the SWI exception handler to be called.
- The handler can then examine the comment field of the instruction to decide what operation has been requested.
- By making use of the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.
- See Exception Handling Module for further details.