

# 32-bit Processor architecture

Parikshit Godbole  
Hardware Technology Development Group  
C-DAC, Pune.

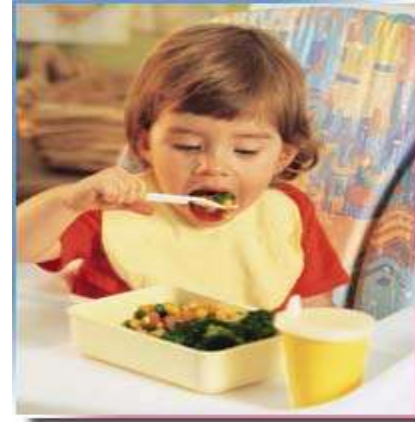
# Identify yourself ?



Spoon  
Feeding



Seeking  
Guidance



Independent



Independent  
and Helping  
others

C-DAC, PUNE

# Outline

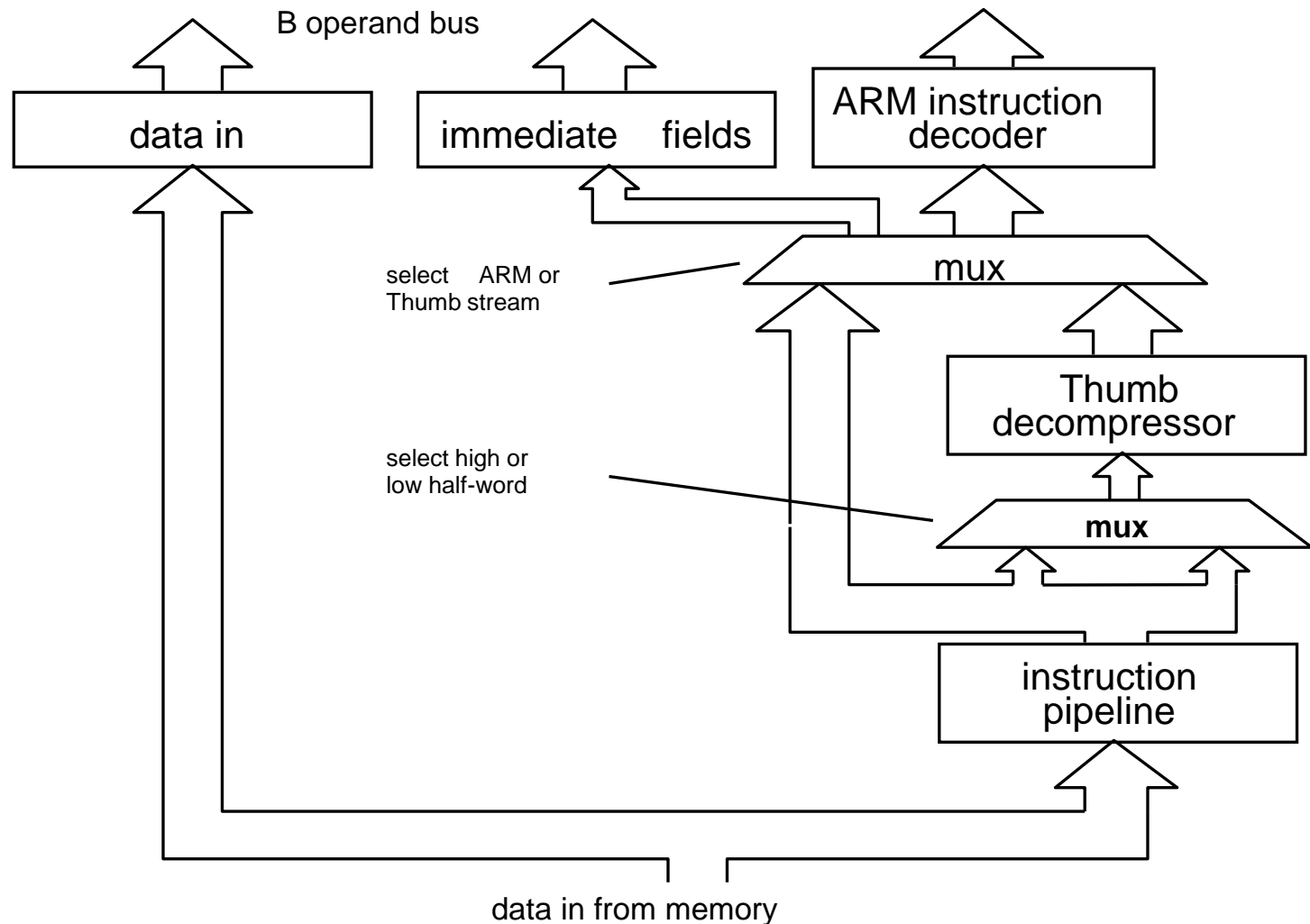
- Thumb hardware implementation
- Thumb instruction set overview
- Entering and exiting from Thumb state
- Guidelines on mixing ARM and Thumb code in same application
  - Thumb properties
  - Design trade-offs
- Enhanced DSP extension, VFP11
- Architectural support for HLL
- Architectural support for OS

# Thumb Hardware implementation

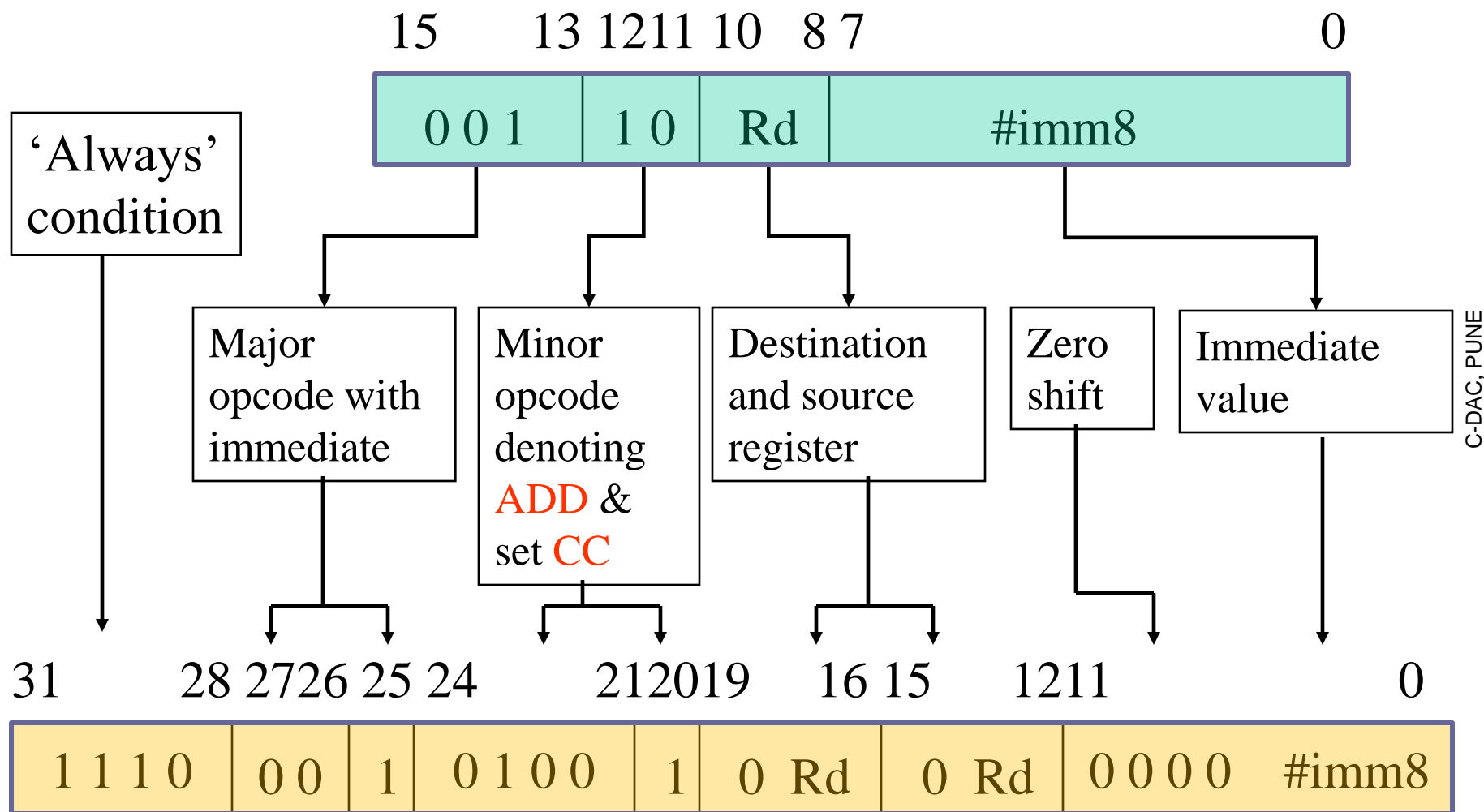
# Thumb implementation

- Thumb instruction decompressor translates Thumb instructions into its equivalent ARM instructions by performing look-up to translate the major and minor opcodes.
- Example shows mapping of a Thumb **'ADD Rd, #imm8'** instruction.
- Thumb 2-address format can always be mapped into the ARM 3-address format by replicating a register specifier.

# Thumb instruction decompressor



# Thumb to ARM instruction mapping



# Overview of Thumb instruction set



# Thumb instruction set overview

- Instructions are **16-bit** long
- Least significant bit of address of instruction is always **“zero”**
- Some instructions use this LSbit to determine whether the code being branched to is Thumb code or ARM code (will cover later)
- All data processing instructions operate on 32-bit values in the registers
- There are **NO** Thumb coprocessor instructions, semaphore instructions, and no Thumb instructions to access the CPSR or SPSR directly

# ARM Vs Thumb capabilities

Capabilities	ARM	THUMB
Conditional execution	Almost all instructions are executed conditionally. Instructions can optionally set condition code flags.	Only conditional branches are allowed. All data processing instructions update flags except when ADD or MOV operates on high registers (e.g. r8 to r12)
Register Access	All instructions can access r0 to r14 with some allowing access to r15 (PC). MRS and MSR instructions are used to move CPSR and SPSR to or from general purpose register where they can be manipulated.	Most instructions can access only r0 to r7, the low registers and update condition code flags, whereas there is limited access to registers r8 to r15, the high registers.
Barrel shifter	Shifting operation can be embedded within the instruction itself.	Shifting operation is an explicit instruction

# Thumb instruction set

- *Branch instructions*
- *Data processing instructions*
- *Single register load and store instructions*
- *Multiple register load and store instructions*
- *Software Interrupt & Breakpoint instructions*

What is **missing** ?

- *Status register access instructions*
- *Semaphore instructions*
- *Coprocessor instructions*

# Branch instructions

- Unconditional Branch up to 2KB
- A conditional Branch up to 256 bytes
- Branch to subroutines up to 4 MB
- Change the processor from ARM state to Thumb state.

e.g.

- B {<cond>} <target address>
- B | BL | BLX <target address>
- BX | BLX <Rm>

# Data processing instructions

- A subset of ARM data-processing instructions.
- All thumb instructions set condition codes.
- Explicit instructions for shifting operands in register.
  - **LSL | LSR | ASR**

# Single register Load/Store instructions

- Eight types of Load/ Store instructions with two basic addressing modes:
  - Register plus register
  - Register plus 5-bit immediate (not available for signed halfword and signed byte loads)

# Single register Load/Store instruction formats

e.g.

- `<opcode1> <Rd>, [<Rn>, #<5_bit_offset>]`
  - LDR | LDRH | LDRB | STR | STRH | STRB
- `<opcode2> <Rd>, [<Rn>, <Rm>]`
  - LDR | LDRH | LDRSH | LDRB | LDRSB | STR | STRH | STRB
- `LDR <Rd>, [PC, #<8_bit_offset>]`
- `<opcode3> <Rd>, [SP, #<8_bit_offset>]`
  - LDR | STR

# Multiple register Load/Store instructions

- Four types of instructions:
  - **LDMIA** and **STMIA** support block copy with fixed 'increment after' addressing mode.
  - **PUSH** and **POP** implement 'fully descending' stack with stack pointer (R13) as base register.
- All instructions update base register after transfer of lower 8 registers.
- **PUSH** can stack return address and **POP** can load PC.



# Use of PUSH and POP

; Call subroutine

**BL** Thumbroutine

; continue

Thumbroutine

**PUSH** {r1, lr} ; enter subroutine

**MOV** r0, #2

**POP** {r1, pc};return from subroutine

# Load/Store multiple instruction formats

- `<opcode1> <Rn>!, <registers>`
  - LDMIA | STMIA
- PUSH | POP {<registers>}

- e.g. **STMIA** r4!, {r1, r2, r3}

What will be value in r4 after this instruction is executed if r4 initially contained 0x8000 ?

# Exception generating instructions

- This causes a processor exception to occur.

e.g.

- SWI <immed\_8>
- BKPT <immed\_8>

# Exploring Branch instructions

Instruction	Description	Effect
B	Branch	pc = label
BL	Branch with Link	pc = label lr = address of next instruction after BL
BX	Branch Exchange	pc = Rm & 0xfffffffffe, T = Rm & 1
BLX	Branch exchange with link	pc = label, T = 1 pc = Rm & 0xfffffffffe, T = Rm & 1 lr = address of next instruction after BLX

# Entering and exiting from Thumb state

# Entering Thumb state

- Executing ARM BX instruction, if bit[0] of address specified by register (Rm) is '1', thumb state is entered.

– BX {<cond>} <Rm>

Effect:  $pc = Rm \& 0xffffffe$ ,  $T = Rm \& 1$

- Setting "T" bit in SPSR and executing an ARM instruction that restores CPSR from SPSR (data processing instruction with "S" bit set and PC as destination, or a LDM with restore CPSR).

# Entering Thumb state

- BLX instruction on architecture version 5 and above and LDR/LDM instructions that load PC can also be used.
  - **BLX <target\_address>** : guaranteed entry to Thumb state
  - **BLX {<cond>} <Rm>** : Thumb entry depends on bit[0] of register Rm  
**Effect:**  $pc = Rm \& 0xffffffe$ ,  $T = Rm \& 1$
  - **LDM {<cond>} <addressing\_mode> <Rn> {!}, <registers>**

# Returning from Thumb state

- If called via BLX in ARM mode (In v5 and above):
  - BX R14
  - PUSH {<registers>, R14} ; subroutine entry
  - POP {<registers>, PC} ; subroutine exit
- BLX with immediate offset (11-bit or 22-bit) always changes to Thumb whereas with register address may change to thumb depending on bit[0].
  - BLX {<cond>} <Rm> : Thumb entry depends on bit[0] of register Rm
- BX with bit[0] of specified register = '0'.
  - BX {<cond>} <Rm>

Effect:  $pc = Rm \& 0xffffffe$ ,  $T = Rm \& 1$



# Guidelines on mixing ARM and Thumb code in same application

# Example

CODE32 (or ARM)

LDR            r0, = thumbroutine+1; Enter thumb state

BLX            r0            ; Jump to thumb code

; continue here

CODE16 (Pre-UAL assembly syntax) | THUMB | THUMBX

thumbroutine

ADD            r1, #1 ; enter subroutine

BX            r14        ;return to ARM code and state

THUMB implies assembly using UAL syntax for Thumb-2

THUMBX implies assembly using UAL syntax for Thumb-2EE

# Remember...

- Assembler will not put ARM-Thumb or Thumb-ARM mode change instructions on its own.
- Use (architecture specific) supported instructions for entering or exiting from Thumb mode.
- Alternately one can design a portable code using instructions supported on all architecture varieties.
- Partition critical tasks from non-critical tasks and decide on strategy, memory requirement, power requirement for ARM and Thumb code.

# Thumb variants and properties

# Thumb-2

- Made its debut in 2003 in ARM1156 core.
- Enhancement to Thumb ISA with
  - New 16-bit instructions for improved program flow
  - New 32-bit instructions derived from ARM equivalent
- Enables use of coprocessor instructions, conditional execution
- Best of both the worlds – Code density and performance
- Bit field manipulating instructions for handling packed structures
- Bit reversal and byte reversals as a single instructions (for DSP e.g. FFT)
- Compare zero and branch

# Thumb-2 processor's view



- hw1 determines the instruction length and functionality.
- If bits[31:16] indicate a 32-bit Thumb instruction, then hw2 is fetched from instruction address + 2.
- This is possible only in Thumb execution state.

# ARM, Thumb, Thumb-2 comparison

## ARM (v6 or earlier)

```
AND r2, r1, #bitmask  
BIC r0, r0, #bitmask << bitpos  
ORR r0, r0, r2, LSL #bitpos
```

## Thumb-2 (ARM or Thumb)

```
BFI r0, r1, #bitpos, #bitwidth
```

# ARM, Thumb, Thumb-2 comparison

## ARM

```
LDREQ r0, [r1]
LDRNE r0, [r2]
ADDEQ r0, r3, r0
ADDNE r0, r4, r0
```

## Thumb

```
BNE L1
LDR r0, [r1]
ADD r0, r3, r0
B L2
L1
LDR r0, [r2]
ADD r0, r4, r0
L2
```

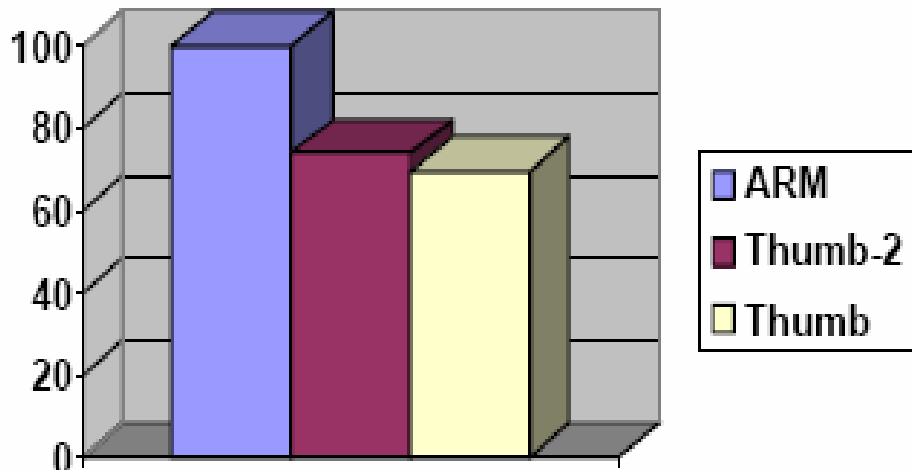
## Thumb-2

```
ITETE EQ
LDR r0, [r1]
LDR r0, [r2]
ADD r0, r3, r0
ADD r0, r4, r0
```

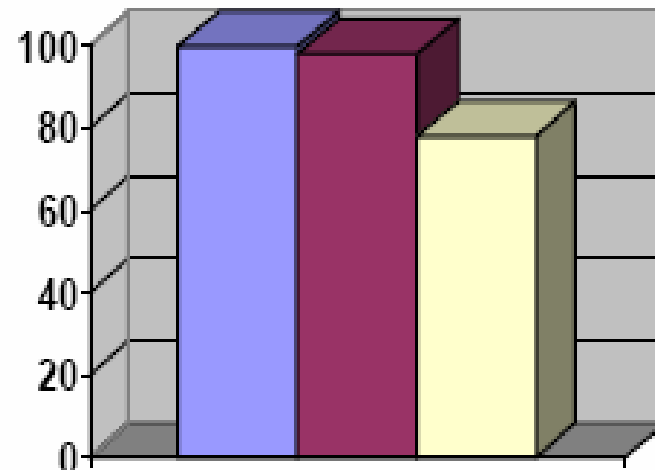


# ARM, Thumb, Thumb-2 performance comparison

Code Size



Performance



# Thumb-2EE

- Extension to Thumb-2 Instruction set particularly suited to code generated at runtime (e.g. by JIT compilation or AOT compilation) in managed **Execution Environments** supporting beyond java, such as Microsoft .NET Compact Framework, Python and others..
- Thumb-2EE include automatic null pointer checks on every load and store instruction, an instruction to perform an array bounds check, and the ability to branch to handlers, which are small sections of frequently called code, commonly used to implement a feature of a high level language, such as allocating memory for a new object.

# Thumb properties

- Thumb code requires **70 %** of the space of ARM code. Thumb-2 takes **74%** equivalent.
- Thumb code uses **40 %** more instructions than ARM code.
- With 32-bit memory, ARM code is **40 %** faster than Thumb code. Thumb-2 code is **38 %** faster than Thumb code.
- With 16-bit memory, Thumb code is **45 %** faster than ARM code.
- Thumb code uses **30 %** less external power than ARM code.

# Design trade - offs

- Parameters include cost, performance, and power.
- A high-end 32-bit ARM system may use Thumb code for certain non-critical routines to save power or memory requirement.
- A low-end 16-bit system may have a small amount of on-chip 32-bit RAM for performance hungry routines in ARM code and off-chip Thumb code for other jobs.

## Design trade – offs contd..

- Higher code density implies less memory space and reduction in cost.
- Power dissipation can be reduced by running ARM code at lower clock speeds.
- Power may also be reduced by having higher code densities since differing memory types have differing requirements.
- Thumb-2 with its 32-bit extensions can achieve higher code densities as well as ARM-like performance.

# Assignment

- **Program #1:** A Palindrome Detector (ARM, Thumb)
- Aim: To write assembly language programs to determine if a given string is a “palindrome”.
- A *palindrome* is a string of characters which reads the same way both backwards and forwards.
- For example, the following strings are palindromes: *a*; *aba*; *cxxc*; *bob*; *madam*. The empty string can also be considered a palindrome. On the other hand, the following strings are not palindromes: *ab*; *xyz*; and *forward*.

# Recap

- Thumb Hardware implementation
- Thumb instruction set overview
- Entering and exiting Thumb state
- Guidelines on mixing ARM and Thumb code in same application
  - Thumb properties
  - Design trade-offs

## Next..

- Enhanced DSP extension
- Vector Floating point architecture (VFP11)
- Architectural support for HLL

# Enhanced DSP extension



# DSP characteristics

- DSP algorithms typically work on 16-bit array with intermediate results being 32-bit wide.
- These numbers are signed fixed point numbers known as Q15 or Q31 with 15 binary places and numeric value ranging from  $-1$  to  $+1 - 2^{-15}$ .
- These numbers are characterized by saturated arithmetic when subject to overflow.
- ARM includes special instructions for saturated arithmetic and efficient load and store instructions for data and coefficients.

# Effects of overflow

- Usable range of 32-bit signed number is  $-2^{31}$  to  $+2^{31}-1$ . If result of some arithmetic operation overflows positively or negatively, result reduced to modulo  $2^{32}$ .
- This could cause unwanted glitches in the output if used in audio applications.
- Saturated arithmetic allows output to be set to maximum of positive or negative extreme in case of overflow, so that it becomes the closest representable number to the correct mathematical result of the operation.

# The Q-flag

- Q-flag in PSR is set when
  - Saturation of addition result in **QADD** or **QDADD**
  - Saturation of subtraction result in **QSUB** or **QDSUB**
  - Saturation of doubling intermediate result in **QDADD** or **QDSUB**
  - Signed overflow during an **SMLA<x><y>** or **SMLAW<y>**
- Q-flag is sticky flag, once set, it remains 1 unless explicitly reset by some instructions.
- Q-flag is not affected by any other arithmetic instruction, such as **ADD**, **SUB** or **MLA**

# DSP instruction types

- Signed Integer multiply and multiply accumulate instructions
  - **SMUL** <x> <y> : 16 X 16 → 32
  - **SMULL** <x> <y> : 32 X 32 → 64
  - **SMULW** <y> : 32 X 16 → 32(48)
  - **SMLAW** <y> : 32 X 16 + 32 → 32
  - **SMLA(D)** <x> <y> : 16 X 16 + 32 → 32
  - **SMLAL** <x><y> : 32 X 32 + 64 → 64
- Instructions **SMUL**, **SMULW**, **SMLAL** does not affect N, Z, C, V, or Q flags. **SMULL**, **SMLAL** affects N,Z.
- Instructions **SMLA**, **SMLAW** does not affect N, Z, C, or V flags.

# DSP instruction types

- Signed Integer multiply and multiply accumulate instructions
  - **SMUAD** <x> <y> : Top & Bot 16 X 16 → 32  
and add results
  - **SMLAD** <x> <y> : Reg + two(16 X 16) → 32
  - **SMLSLD** <y> : Rx & Ry + [(Rz(L16) X  
Rw(L16)) – (Rz(U16) X Rw(U16))] → 32(48)
- Instructions **SMUAD**, **SMLAD**, **SMLSLD** can affect Q flag.

# DSP instruction types contd...

- Saturated addition and subtraction instructions
  - **QADD** : Saturated integer addition
  - **QDADD**: Saturated integer doubling of one operand followed by saturated integer addition with the other operand.
  - **QSUB** : Saturated integer subtraction
  - **QDSUB**: Saturated integer doubling of one operand followed by saturated integer subtraction from the other operand.
- These instructions does not affect N, Z, C, or V flags.

# DSP instruction types contd...

- Two word load and store instructions
  - **LDRD**, **STRD** using register pairs {R0, R1}, {R2, R3} and so on till {R12, R13}
  - Addressing mode same as LDRH/STRH instructions.
- Cache preload instruction
  - **PLD** a hint to the memory system indicating a specified address is likely to occur.
  - Unconditionally executed.
  - Can be useful in hiding delay in case of cache miss.

# DSP instruction types contd...

- Two word coprocessor register transfer instructions
  - **MCRR** Transfers two register values to a coprocessor
  - **MRRC** Transfers coprocessor values to two ARM registers.



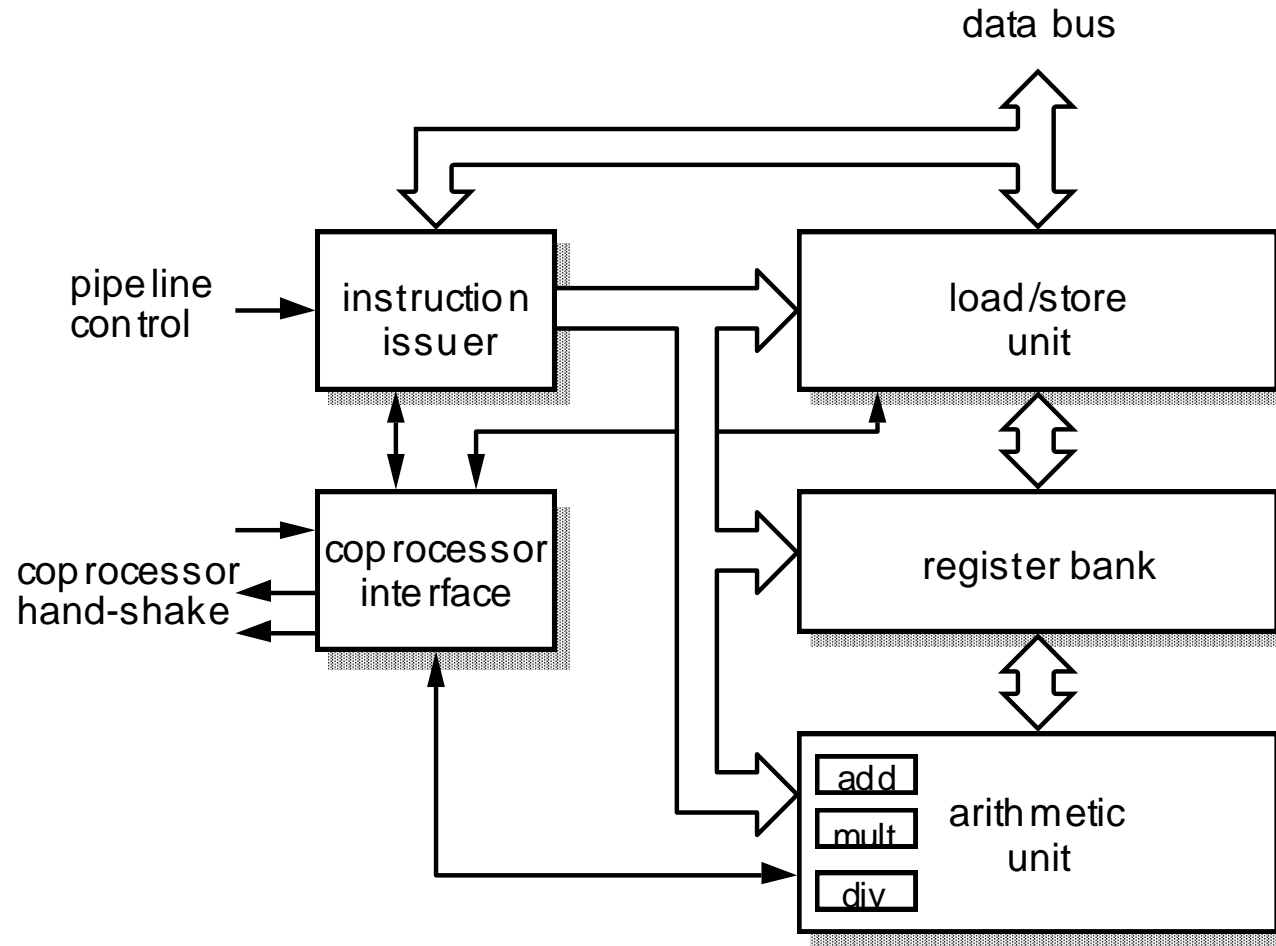
# VFP11

# ARM Floating Point Architecture

- **Features :**

- Solution around CP11 co-proc. Or software emulator.
- Four banks of eight 32-bit registers.
- User-visible Floating Point Status Register, User-invisible Floating point control register.
- Support for single and double precision IEEE 754, 1985 formats.
- Has arithmetic unit which incorporates add, subtract, multiply, divide, square root functions with rounding and normalizing hardware.
- Pipelined architecture with stages of prepare, calculate, align, round.
- Concurrent operation of arithmetic unit and load/store of data for next instruction allows no stalling of pipeline.
- Context switch and corresponding status storing is done only in case subsequent process switched in or else FPA is turned off.

# ARM Floating Point Architecture



# ARCHITECTURAL SUPPORT FOR HIGH-LEVEL LANGUAGES

# Abstraction in software languages

- Difficulties in dealing with machine code.
  - Instruction set : one level up
  - Terms used : instructions, bytes, word, address.
- Need to have standard.
- Complex applications need higher up abstractions for implementations.
- Should the target architecture details be oblivious to high level programmer ?
- Complexity and efficiency of compiler in turn makes programmers job simple.
- CISC Vs RISC Philosophy for compilers and instruction sets.

# Data Types

- Data Characteristics : number,order,grouping of bits.
- Basic Type : 32 bit , Unsigned Integer.
- Signed 32 bit and un/signed byte,16-bit halfword support.
- Un/signed 64 bit support with two 32-bit operations.
- ANSI C basic Data types
  - (Un) Signed Character : 8 bit.
  - (Un) Signed Short Integer : 16 bit.
  - (Un) Signed Integer : 16 bit.
  - (Un) Signed Long Integer : 32 bit.
  - Floating point, Double, Long Double.
  - Enumerated types.
  - Bit fields.

# Arrays, Functions, Structures, Pointers

- Arrays : Base plus scaled index addressing for scan, objects of size  $2^n$  bytes using pointer to the start of the array and loop variable as the index.
- Structures : With base plus immediate offset addressing for accessing objects in it. Note that structure starts at 32-bit word boundary only.
- Pointers: 32-bit unsigned integers and follow all arithmetic operations.
- Subroutine, Function and Procedure: Various branch instructions to facilitate short and long jumps.

# Expressions

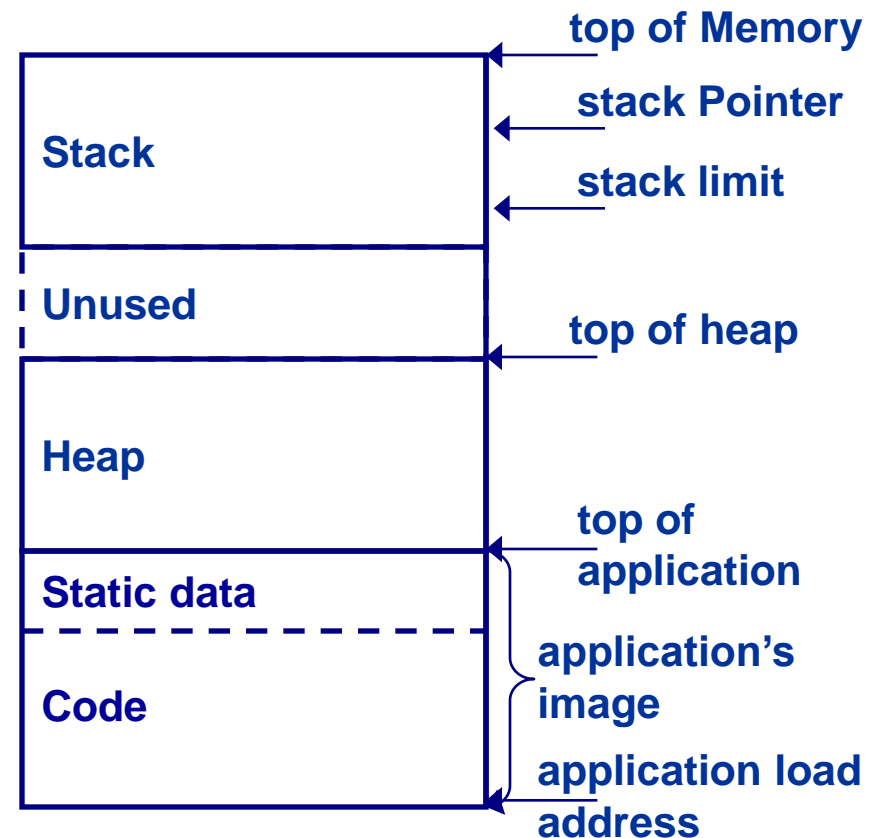
- Except division and remainder, Unsigned arithmetic is implemented directly with arithmetic, bit-wise and shift primitives.
- Key to complex expression evaluation is to store required values in right order and to store frequently used values in register.
  - Trade-off : Maximum number of registers to store required values Vs. available registers for intermediate results.
- Sorting out order to load registers and combining operands to get correct results solving the operator precedence is the requirement of optimum compiler.



# Expressions

- 3-address instruction gives more flexibility over 2 address instruction, more over the scarcity of registers in Thumb mode makes compiler job more difficult.
- Operands are accessed through registers, stack, procedure's literal pool, local or global variable.
- Arithmetic on pointers used depends on size of the data used, if variable is used to increment then it is scaled accordingly.
- Arrays – Similar to pointers with array declaration establishing name, data size and pointer to first element.

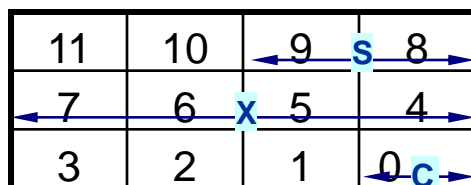
- Memory arranged is linear set of logical addresses, user program expects access to fixed program memory area.
- Two areas grow dynamically, where compiler cannot workout maximum size : stack ,heap.
- The stack : With every non-trivial function new activation frame is created on the stack containing backtrace record and local variables.
- The heap : Area of memory used to satisfy program requests (malloc).
- Stack - recovered automatically  
Heap – to be freed.
- Address space model shows



unused area to the bottom of stack and top of heap, This area is allocated on demand to heap or stack.

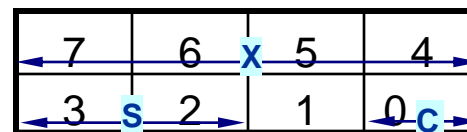
- Once unused area is full application halts

- Non-trivial functions called trivially increase the size of the stack.
- Data Alignment: Bytes at any byte address, half-words at even bytes and words on four byte boundaries. Several data items of different data type will be stored with padding to suit the above alignment.
- e.g. `struct S1 {char c; int x; short s1;}`

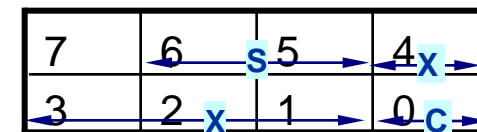


The same structure reorganized gives more efficient data alignment.

`struct S1 {char C; short S; int X;}`



- Packed struct :In case data sharing is required to transfer the data between external system with separate alignment constraint or need to store the data in tightly packed format the structure is packed despite the performance penalty.



# Functions and Procedures

- void function itself is procedure and arguments passed by value does not change the argument value. To change the value, argument should be passed as pointer to the argument only.
- In order to mix routines generated by different compilers and written in assembly, set of rules are defined in ARM Procedure Call Standard.
- AAPCS (Earlier APCS or ATPCS) imposes conventions specific to ARM Limited 'C' compiler related to.

Particular definitions for general purpose registers.

Defining Form of stack used (full ascending/descending).

Format of stack based data structure for back tracing when debugging.

Defines Function argument and result passing mechanism

Supports ARM shared library mechanism

- Register usage

Four argument registers which pass values into function

Five(to seven) regs. Which function must return with unchanged values.

Seven(to five) regs. Which have dedicated role at least for some time.

# Functions and Procedures

- Argument passing: if floating point arguments, those are stored in floating point registers, other arguments stored as list of words in a1 to a4 and rest if any are pushed on to stack in reverse order.
- Result return : Simple through a1, more complex is returned in memory to a location specified by address in a1.
- Function entry and exit

	<b>BL</b>	<b>leaf1</b>	
	....		
<b>leaf1</b>	....		
	<b>MOV</b>	<b>pc, lr</b>	<b>; return</b>

	<b>BL</b>	<b>leaf2</b>	
	....		
<b>leaf2</b>	<b>STMFD</b>	<b>sp!, (regs, lr)</b>	<b>; save registers</b>
	....		
	<b>LDMFD</b>	<b>sp!, (regs, pc)</b>	<b>; restore and return</b>

- Tail continued functions

# Conditional Statements

- if.... Else
  - The conditional execution of instruction allows long nested if..else sequences. e.g. If (a>b) c=a; else c=b;

```
CMP    r0, r1 ; if (a>b)
MOVGT  r2, r0 ; c=a
MOVLE  r2, r1 ; else c=b
```

```
MOV    r2, r0 ; c=a
CMP    r0, r1 ; if (a>b)
MOVLE  r2, r1 ; c=b
```

```
CMP    r0, r1 ; if (a>b)
BLE    ELSE ; Skip clause if false
MOV    r2, r0 ; c=a
B      ENDIF ; Skip else clause
ELSE   MOV    r2, r1 ; else c=b
ENDIF  ..
```

- Switches
  - Can be implemented after conversion to if..else but are rather slower so jump table is used. Jump table stores target instruction address for each possible value. A default address must be kept if the expression value is out of range.

## • Switch Example

```
switch (expression) {
    case constant – expression1 : statement S1
    case constant – expression2 : statement S2
    .....
    case constant – expressionn : statement Sn
    default : statementD
```

## Conditional Statements

```
temp = expression;
if (temp == constant – expression1)
    {statements1}
else ....
else if (temp == constant – expressionn)
    {statementsn}
else (statementsD)
```

## Using Jump table

	<b>ADR</b>	<b>r1, JUMPTABLE</b>	; ro contains value of expression
	<b>CMP</b>	<b>r0, #TABLEMAX</b>	; get base of jump table
	<b>LDRLS</b>	<b>pc, [r1, r0, LSL #2]</b>	; check for overrun
		<b>; statements<sub>D</sub></b>	; ... if OK get pc
	<b>B</b>	<b>EXIT</b>	; ... otherwise default
			; break
<b>L1</b>	<b>....</b>	<b>; statements<sub>1</sub></b>	
	<b>B</b>	<b>EXIT</b>	; break
	<b>....</b>		
<b>LN</b>	<b>....</b>	<b>; statements<sub>N</sub></b>	
<b>EXIT</b>	<b>....</b>		

# Loops

- for loop :
  - The expression can be evaluated with conditional execution, keeping initializations out of the for loop. e.g. for(i = 0; i<10; i++) {a[i] = 0;}

LOOP	MOV	r1, #0	; Value to store in a[i]
	ADR	r2, a[0]	; r2 points to a[0]
	MOV	r0, #0	; i = 0
	CMP	r0, #10	; i <= 10 ?
	BGE	EXIT	; if i >= 10 finish
OR	BEQ	EXIT	; if i = 10 finish
	STR	r1, [r2, r0, LSL #2]	; a[i] = 0
EXIT	ADD	r0, r0, #1	; i++
	B	LOOP	
	....		

5 Instructions for loop including 2 branch instructions



# Loops

LOOP	MOV	r1, #0	; Value to store in a[i]
	ADR	r2, a[0]	; r2 points to a[0]
	MOV	r0, #0	; i = 0
	CMP	r0, #10	; i < 10 ?
	BGE	EXIT	; if i >= 10 finish
	STRLO	r1, [r2, r0, LSL #2]	; a[i] = 0
	ADDLO	r0, r0, #1	; i++
EXIT	BLO	LOOP	
	....		

4 Instructions for loop including only 1 branch instruction

- Optimization can be applied by omitting conditional branch to EXIT and executing STR , ADD and B on opposite condition, thus reducing branch.

# Loops

LOOP	MOV	r1, #0	; Value to store in a[i]
	ADR	r2, a[0]	; r2 points to a[0]
	MOV	r0, #0	; i = 0
	STR	r1, [r2, r0, LSL #2]	; a[i] = 0
	ADD	r0, r0, #1	; i++
	CMP	r0, #10	; i < 10 ?
EXIT	BLO	LOOP	; if i >= 10 finish
	....		

## 4 Instructions for loop including only 1 branch instruction

- In Thumb-2, instruction CB(N)Z can be used to replace “compare and branch” (illustrated above) along with a decrementing loop pointer.
- Further improvements are also possible in this case by moving test to the bottom, as initializations and condition involve constant and code is sure to execute at least once.

# Loops

	<b>MOV</b>	<b>r1, #0</b>	<b>; Value to store in a[i]</b>
	<b>ADR</b>	<b>r2, a[0]</b>	<b>; r2 points to a[0]</b>
	<b>MOV</b>	<b>r0, #10</b>	<b>; i = 10</b>
<b>LOOP</b>	<b>STR</b>	<b>r1, [r2, r0, LSL #2]</b>	<b>; a[i] = 0</b>
	<b>SUBS</b>	<b>r0, #1</b>	<b>; i--</b>
	<b>BNE</b>	<b>LOOP</b>	<b>; if i = 0 finish</b>
<b>EXIT</b>		<b>....</b>	

3 Instructions for loop including only 1 branch instruction

- Using a decrementing loop pointer.

# Loops

- While loop
  - Has simpler structure and used when loop is to be executed variable number of times or in other words not known at the time of compilation

```
LOOP      ....      ; evaluate expression
          BEQ      EXIT
          ....      ; LOOP body
          B        LOOP
EXIT      ....
```

```
B        TEST
LOOP      ....      ; LOOP body
TEST      ....      ; evaluate expression
          BNE      LOOP
EXIT      ....
```

- Do While loop
  - Simplest loop where loop body is executed(at least once) before test.

```
LOOP      ....      ; LOOP body
          ....      ; evaluate expression
          BNE      LOOP
EXIT      ....
```

## Key points

- Use a decrementing loop parameter which saves a compare instruction
- Put as many repeated loop statements as possible to reduce number of branches leading to start of the loop
- Look for optimising loops by replacing “for” loops by “do-while”
- Wherever possible use native data type. Here, 32-bit is native support. This may save instructions to do type casting to lower bit width data type
- Assumptions about type casting (for char, short types) should be done carefully as different compilers have different ways of implementing the same

# Summary

- Enhanced DSP extension
- Vector Floating point architecture (VFP11)
- Architectural support for HLL

## Next..

- Architectural support for OS
  - Protection unit and Memory management unit
  - Context switching
  - Synchronisation
  - I/O access

# Architectural support for Operating systems

# Outline

- OS functions and requirements
- ARM system control co-processor (CP15)
- ARM Protection Unit
- Memory Management Unit
- Synchronization
- Context switching
- Input / Output



# OS functions and requirements

# Background

- Multi user Vs single user systems
  - Time slicing, scheduling of jobs
- Memory management
  - Address translations
  - Providing an environment for each program
- Memory protection
  - Error in one program affecting other programs
- Resource allocation

# Embedded system scenario

- Typically runs fixed set of programs and has no mechanism to introduce new programs.
- Cost of using general purpose OS and scheduling mechanisms and resource requirement are higher.
- Solution is light weight RTOS requiring few kilo bytes of memory.
- Systems employing caches are required to specify which areas in memory are cacheable and so on..

# Co-processors

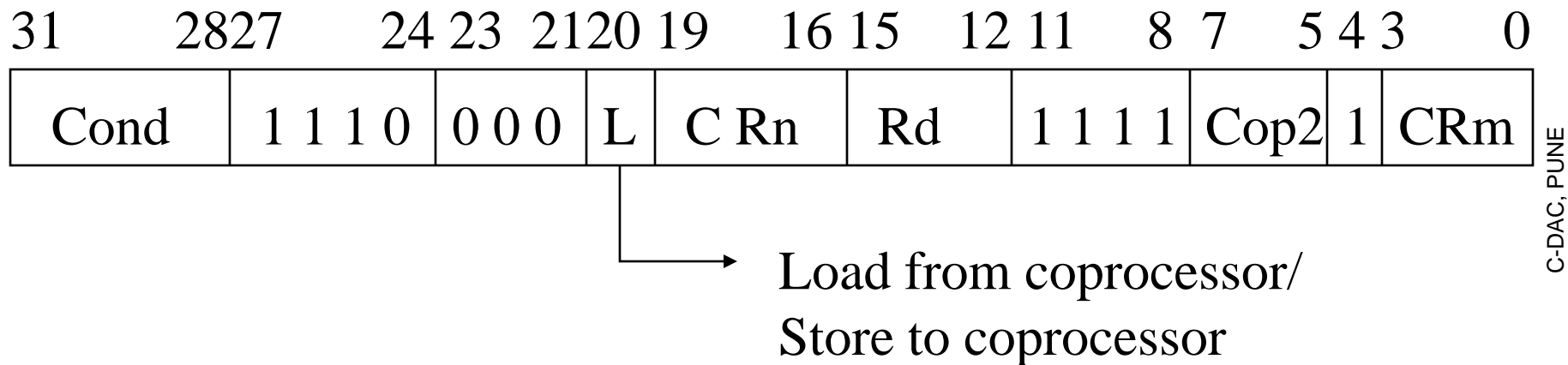
- What are they ?
  - Additional hardware that can be attached to main processor.
  - Have fixed function mostly math processing, memory management
    - e.g. DSP MAC, Floating Point arithmetic, PU
  - Extend the breadth of applications that can be supported on the same processor platform but not as efficient as it would be if these functions were integrated within the processor.

# ARM system control coprocessor

# CP15

- It is an on-chip coprocessor.
- It controls on-chip cache or caches, memory management or protection unit, write buffer, prefetch buffer, branch target cache and system configuration signals.
- Control is affected by read/write instructions on **CP15** registers.
- MRC, MCR are the only instructions defined for **CP15**.
- **Must be executed in supervisor mode**. Any attempt to run these instructions in user mode will cause undefined instruction trap to be generated.

# CP15 instruction format



# CP15 registers

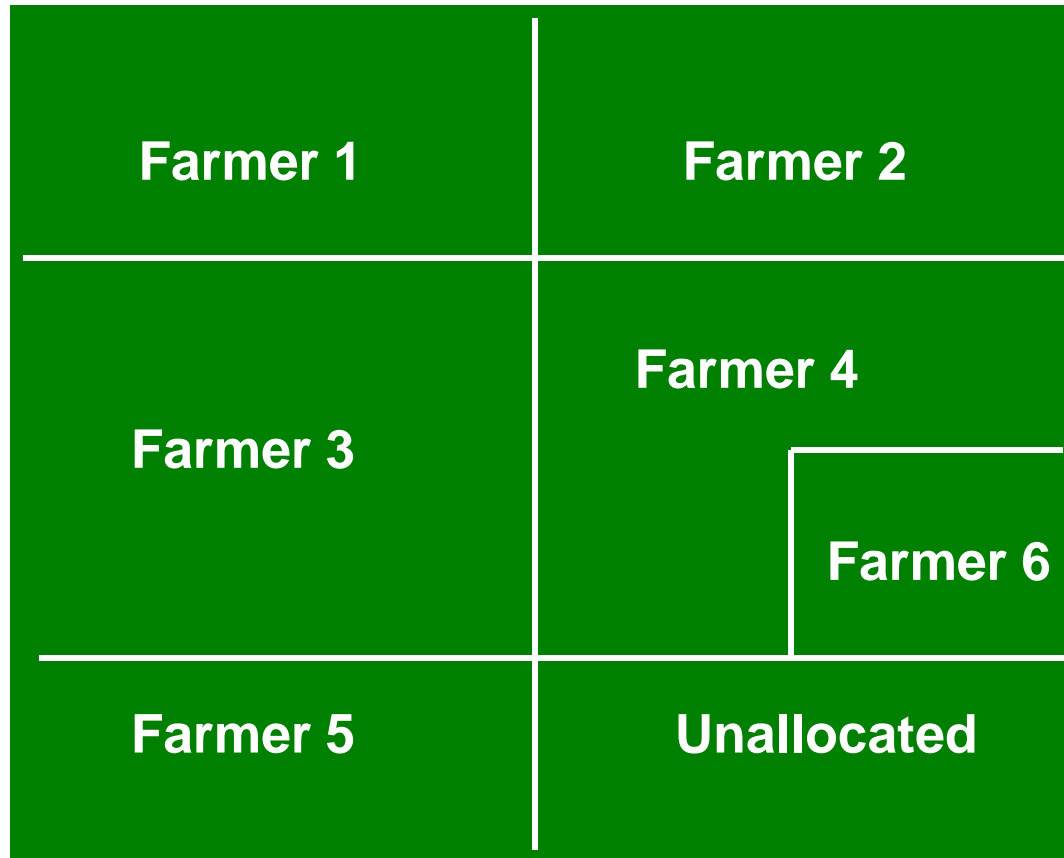
- Employs sixteen, 32-bit registers accessed by specifying **CRn**.
- **Cop2** and **CRm** are used additionally which allows more than 16 physical coprocessor registers to be accessed.
- **Rd** is source/destination ARM register. Specifying PC (**R15**) as **Rd** causes instruction to be **UNPREDICTABLE**.

31	2827	24	23	2120	19	16	15	12	11	8	7	5	4	3	0
Cond	1 1 1 0	0 0 0	L	C Rn	Rd	1 1 1 1	Cop2	1	CRm						



# ARM protection unit

# Concept



**Landlord has  
full access  
to all fields**

# ARM Protection Unit

- Simpler version of memory management architecture as compared to MMU.
- Defines up-to eight (or sixteen) protection regions.
- Protection regions can overlap.
- **No** Virtual to physical address mapping support.
- Does not use translation table.

# Register structure

Register	Purpose
0	ID Register
1	Configuration
2	Cache Control
3	Write Buffer Control
5	Access Permissions
6	Region Base and Size
7	Cache Operations
9	Cache Lock Down
15	Test
4, 8, 10-14	UNUSED

# Example register 1 fields

313029282726252423

14131211    8 7 6 4 3 2 1 0

<b>i</b>	<b>A</b>	<b>n</b>	<b>f</b>	<b>B</b>	<b>n</b>	<b>k</b>	<b>F</b>	<b>L</b>	<b>c</b>	<b>k</b>	<b>S</b>	0	0	0	0	0	0	0	0	<b>V</b>	<b>I</b>	0	0	0	<b>B</b>	0	0	<b>W</b>	<b>C</b>	0	<b>M</b>
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	---	---	---	---	---	---	---	---	----------	----------	---	---	---	----------	---	---	----------	----------	---	----------

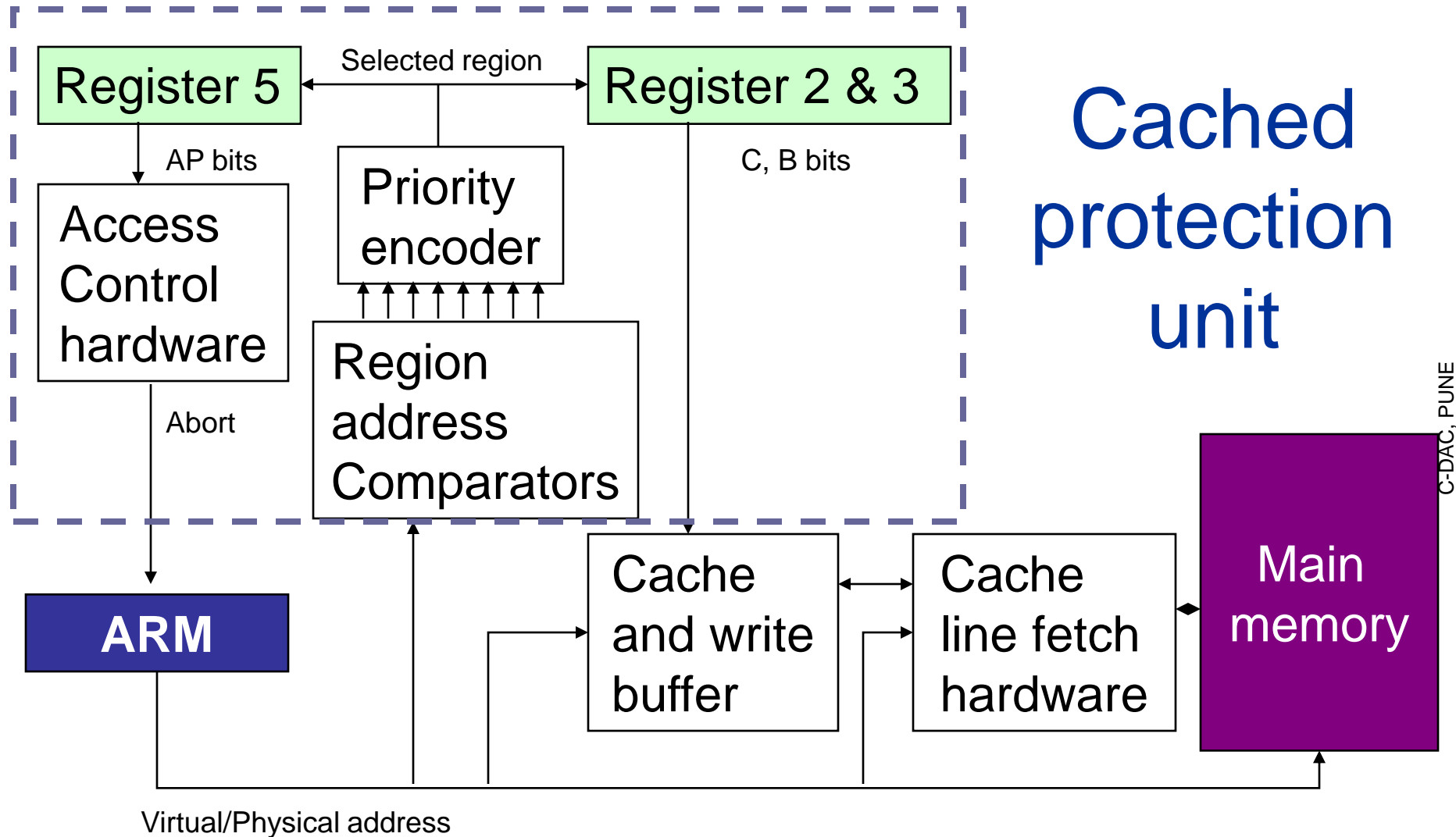
- Register 1 is a configuration register
- **M** enables PU, **C** enables data or unified cache, **W** enables write buffer, **B** switches from little endian to big endian byte ordering, **I** enables instruction cache (if it is separate from data cache), **V** causes exception vectors to move to near the top of address space, **S**, **Lck**, **F**, and **Bnk** control cache on ARM740T and **nf** and **iA** control clock mechanisms on ARM940T.

# Registers used

- Register 6 defines size and starting addresses of the eight units.
  - Size range: 4KB to 4GB
  - Starting address of a region must be multiple of its size.
    - e.g. 4KB region (size 0x1000) can start at address 0x12345000 but 8KB (size 0x2000) cannot.
- Register 2 defines cacheable (C) bits
- Register 3 defines bufferable (B) bits
- Register 5 defines access permissions (AP)

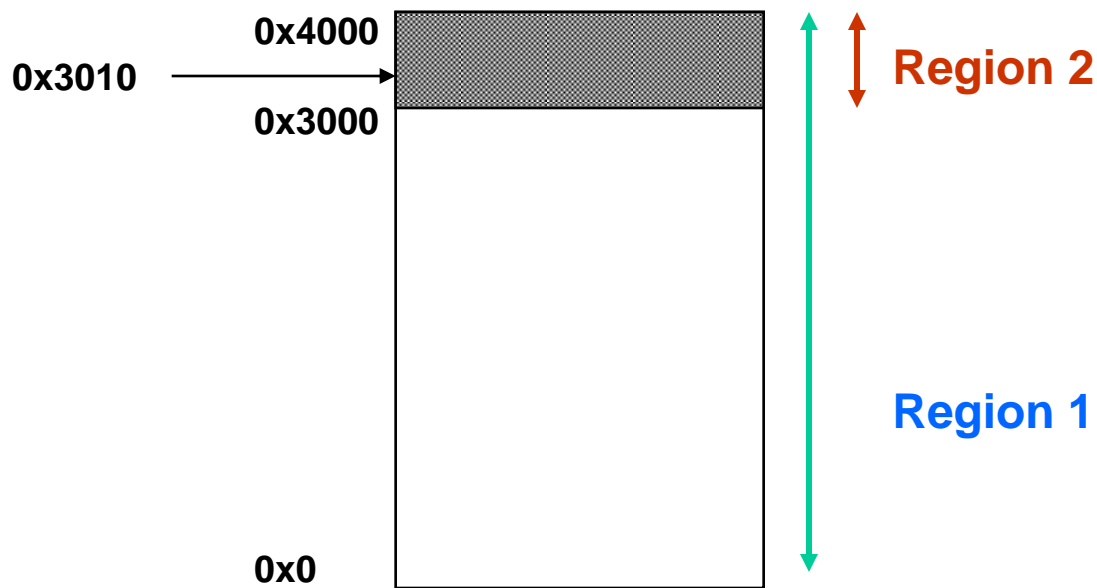
# Memory accesses

- Memory accesses aborted if
  - Address does not lie in any protection region
  - If AP bits doesn't permit access
- Register 1 bit[0] controls enabling and disabling the protection unit.
- System may employ separate instruction and data access protection units using **Cop2** field.
- Code that enables protection unit must lie within a valid (instruction) protection region.





# Using overlapping regions



- **Region 1:** 16 KB with privileged mode access only
- **Region 2:** 4 KB, with privileged mode full access, User mode read only
- Background regions (Not falling in any defined areas)

# Access Permission settings

AP	Privileged permissions	User permissions
0b00	No access	No access
0b01	Read/Write	No access
0b10	Read/Write	Read only
0b11	Read/Write	Read/Write

# Memory management unit

# Memory Management Unit

- Provides full virtual memory architecture.
- Features:
  - High level system control using translation tables for virtual-to-physical addresses.
  - Cacheability and bufferability
  - Domain based control
  - Status information about memory aborts

# MMU registers

Register	Purpose
0	ID Register
1	Control
2	Translation Table Base
3	Domain Access Control
5	Fault Status
6	Fault Address
7	Cache Operations
8	TLB Operations
9	Read Buffer Operations
10	TLB lockdown
13	Process ID Mapping
14	Debug Support
15	Test & Clock Control
4, 11-12	UNUSED

# MMU registers

- MMU is controlled by **CP15** registers 2, 3, 4 , 5, 6, 8 and 10 and some bits of register 1.
- Register 1: control M, A, S, R bits
  - **M**: MMU enable/disable
  - **A**: Alignment check enable/disable
  - **S, R**: Used for access permissions
- Register 2: Translation table base
  - Physical address of the currently active first level translation table.

# MMU registers contd...

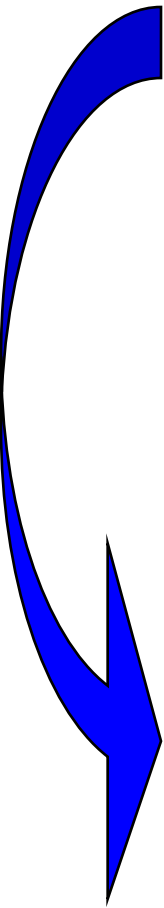
- Register 3: Domain access control
  - Controls domain accesses up to 16 entries.
- Register 4: reserved
- Register 5: Fault status
  - Returns value of the fault status register i.e. fault domain and type of access.
- Register 6: Fault address
  - Reads value of fault address register (FAR), a virtual address of data access.

# MMU registers contd...

- Register 8: TLB functions
  - Invalidate instruction/data or both TLB single or entire entries.



# MMU Access categories

- 
- Sections – 1MB blocks of memory
  - Page-mapped access
    - Tiny pages: 1 KB
    - Small pages: 4 KB
    - Large pages: 64 KB
  - Two levels of translation tables are maintained in the memory
    - First level table holds both sections and pointers to second level tables
    - Second level table hold large and small page translations and in one case tiny pages.

# Domains

- It is a collection of sections, large pages and small pages.
- ARM architecture supports 16 domains.
- Two kinds of domain accesses
  - Clients
    - Are users of domain (execute programs, access data) and are guarded by AP.
  - Managers
    - Control the behaviour of the domain (domain access and the current sections and pages in the domain) and are not guarded by AP.

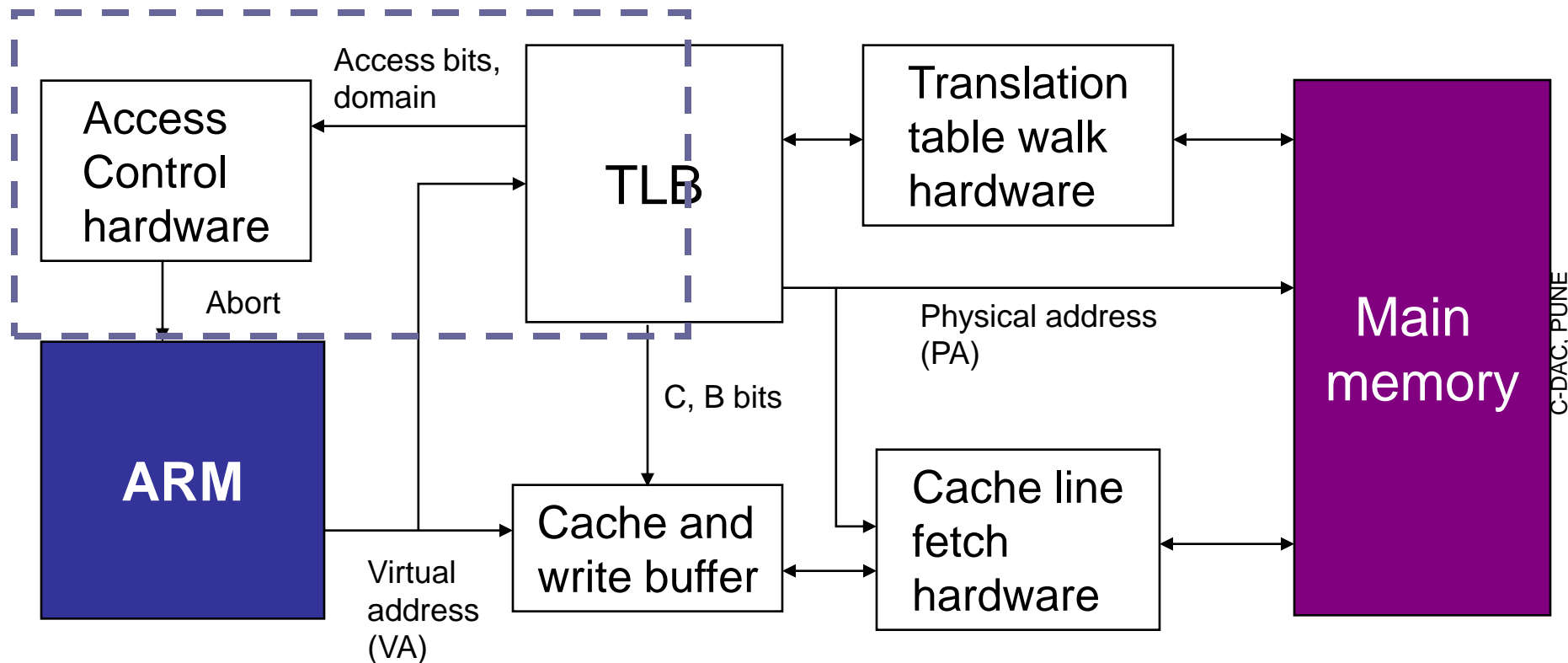
# Domain access values

Value	Privileged permissions	Description
0b00	No access	Any access generates a domain fault
0b01	Client	Accesses checked against AP
0b10	Reserved	UNPREDICTABLE if used
0b11	Manager	Accesses are not checked against AP.

# Access Permission settings

AP	S	R	Privileged permissions	User permissions
0b00	0	0	No Access	No Access
0b00	1	0	Read only	No Access
0b00	0	1	Read only	Read only
0b00	1	1	UNPREDICTABLE	UNPREDICTABLE
0b01	X	X	Read/Write	No Access
0b10	X	X	Read/Write	Read only
0b11	X	X	Read/Write	Read/Write

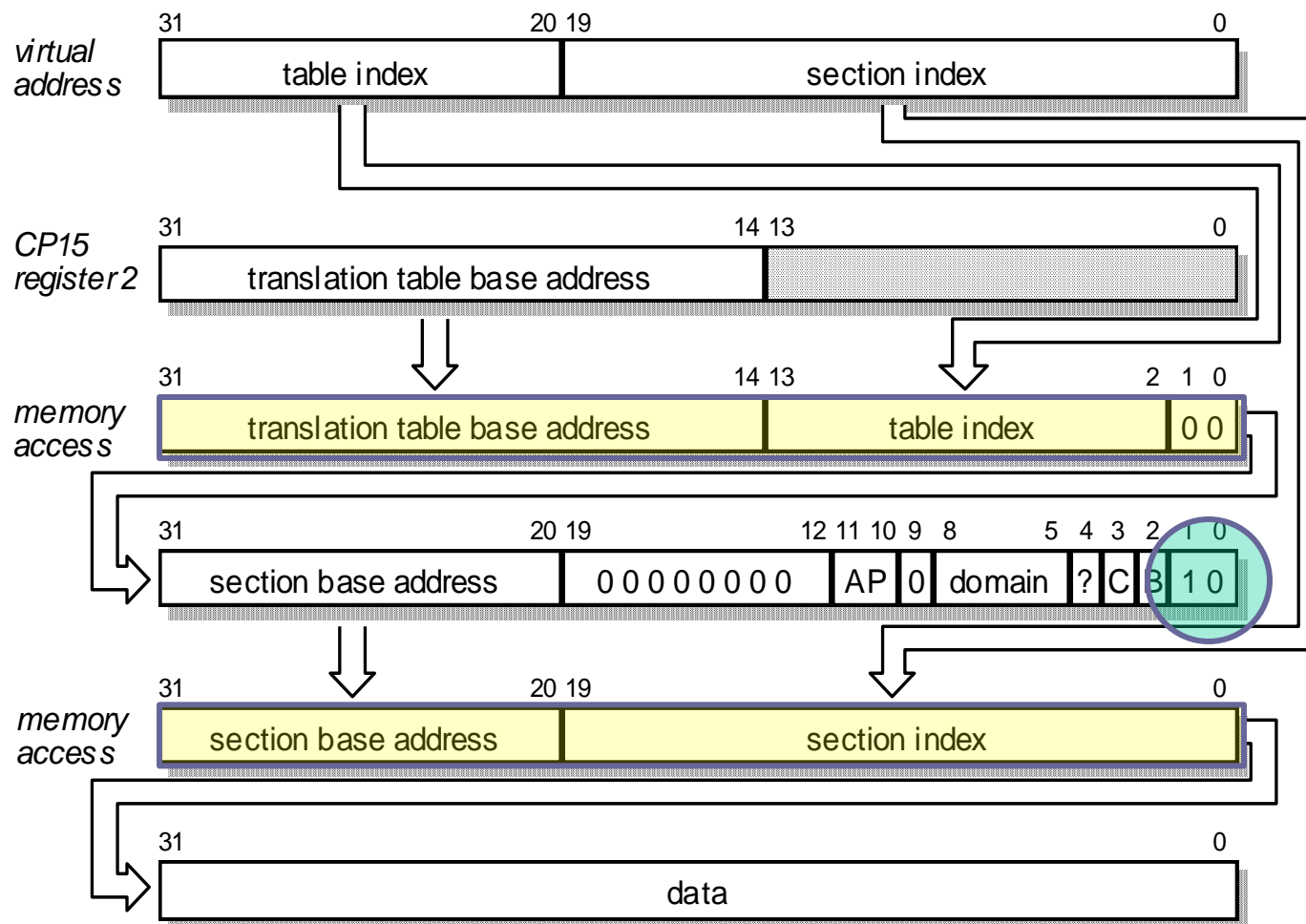
# Cached MMU memory system



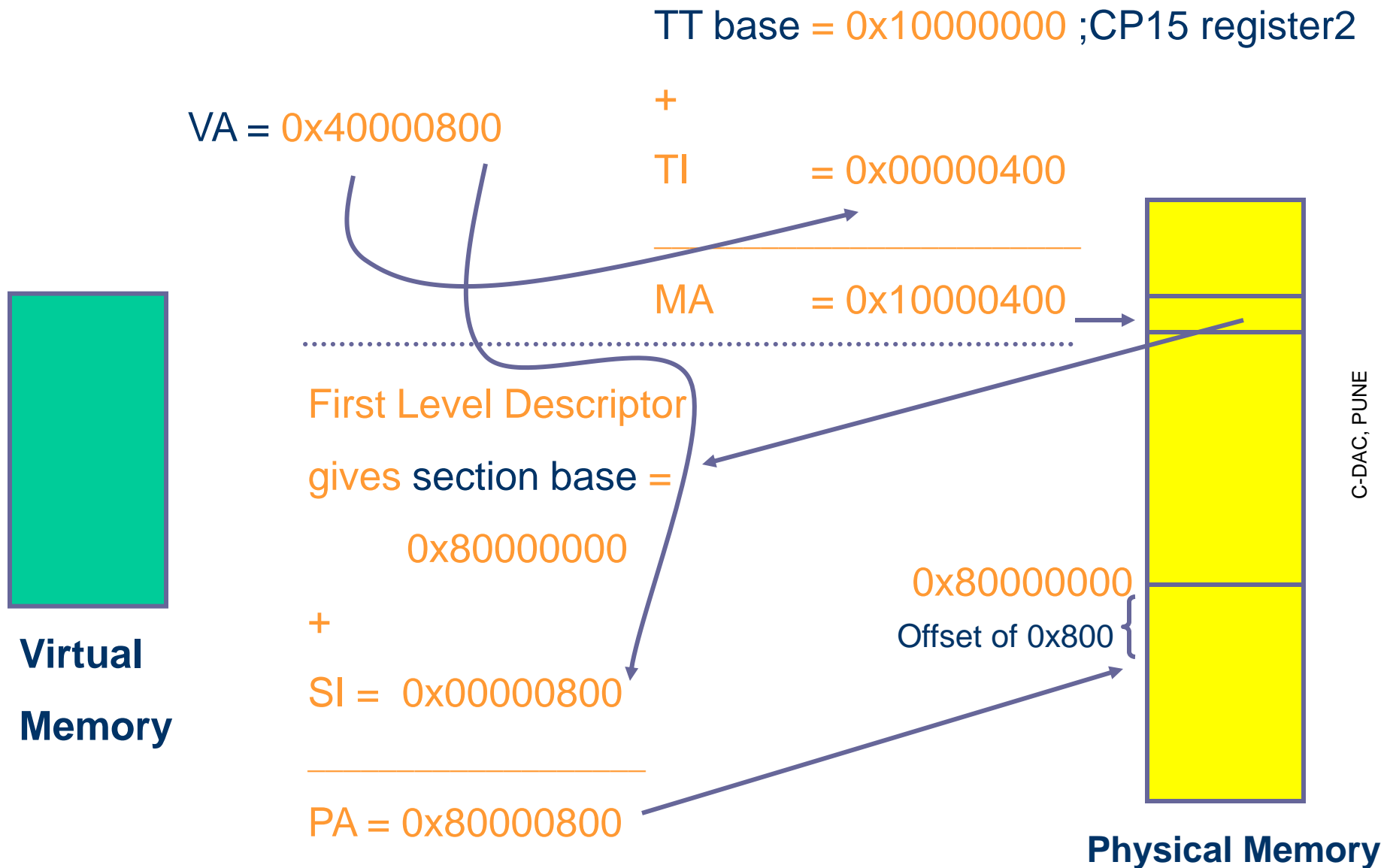
# Memory access sequence

- ARM generates virtual address. How come virtual ??
- MMU looks in TLB (Instruction or data TLB)
- If address is not found, walk hardware is invoked to retrieve the translation and AP information from translation table held in main memory.
- This information is stored in TLB.
- C (cacheable), B (bufferable) bits are used to control the cache and write buffer.
- AP bits and domain control actual access. If access is not permitted MMU signals abort.
- PA is used as address for main memory access for non-cached memory accesses or as the address for line fetch in the event of cache miss for cached memory accesses.

# Section translation sequence

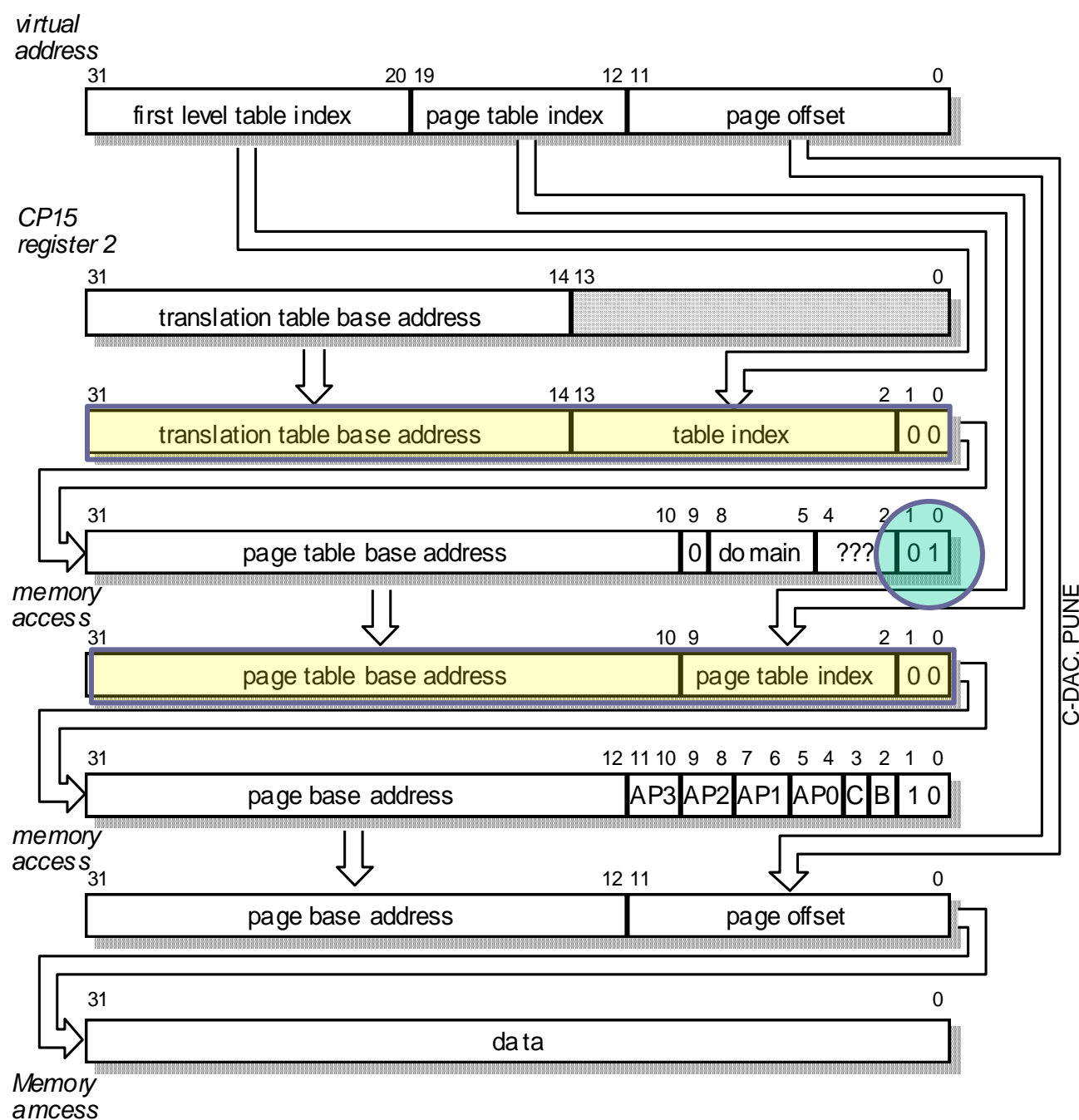


# Section translation sequence





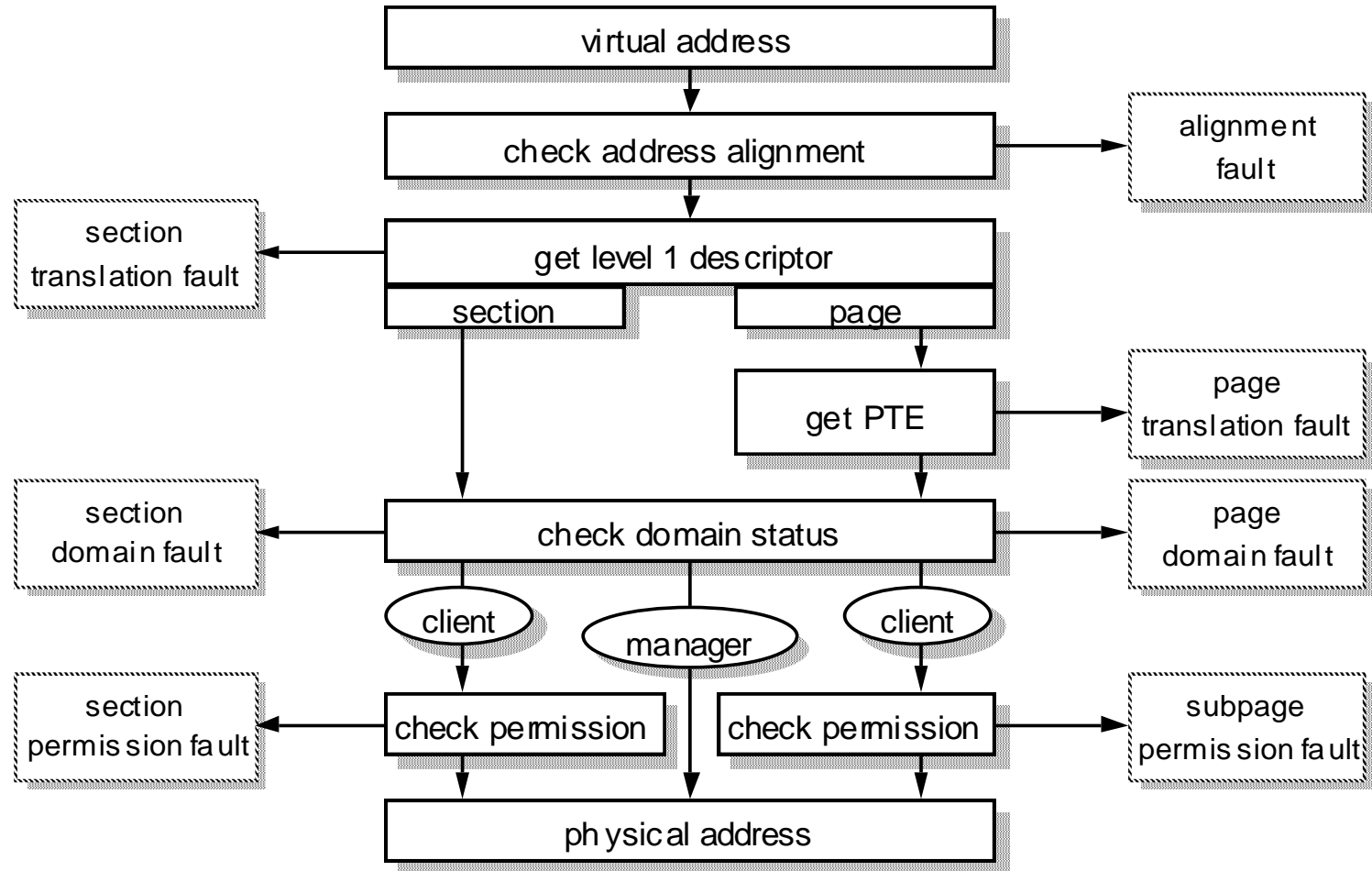
# Small page translation sequence



# Aborts

- MMU faults
  - Alignment fault
  - Translation fault
  - Domain fault
  - Permission fault
- External aborts
  - Reads
  - Buffered writes
  - First level descriptor fetch
  - Second level descriptor fetch

# Faults



# Process synchronization

# Synchronization

- It is a basic mechanism between different processes accessing a shared data structure to ensure correct behaviour.
- Achieved by having mutually exclusive access.

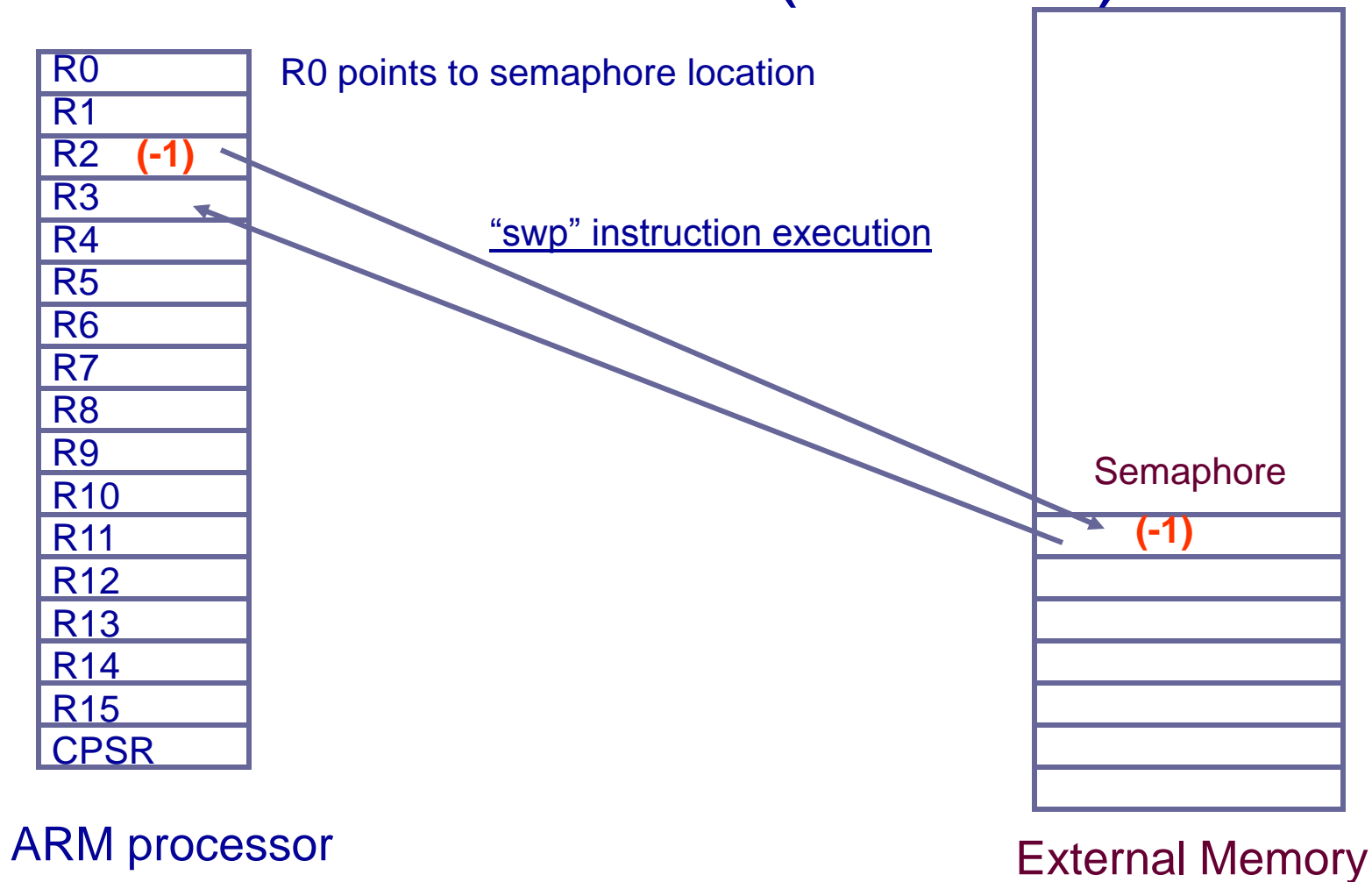
How ?

- Checking access status from a memory location
- Architectural support in the form of **SWP** which is an atomic operation. Atomic means it involves a locked load and store transaction which cannot be interrupted.

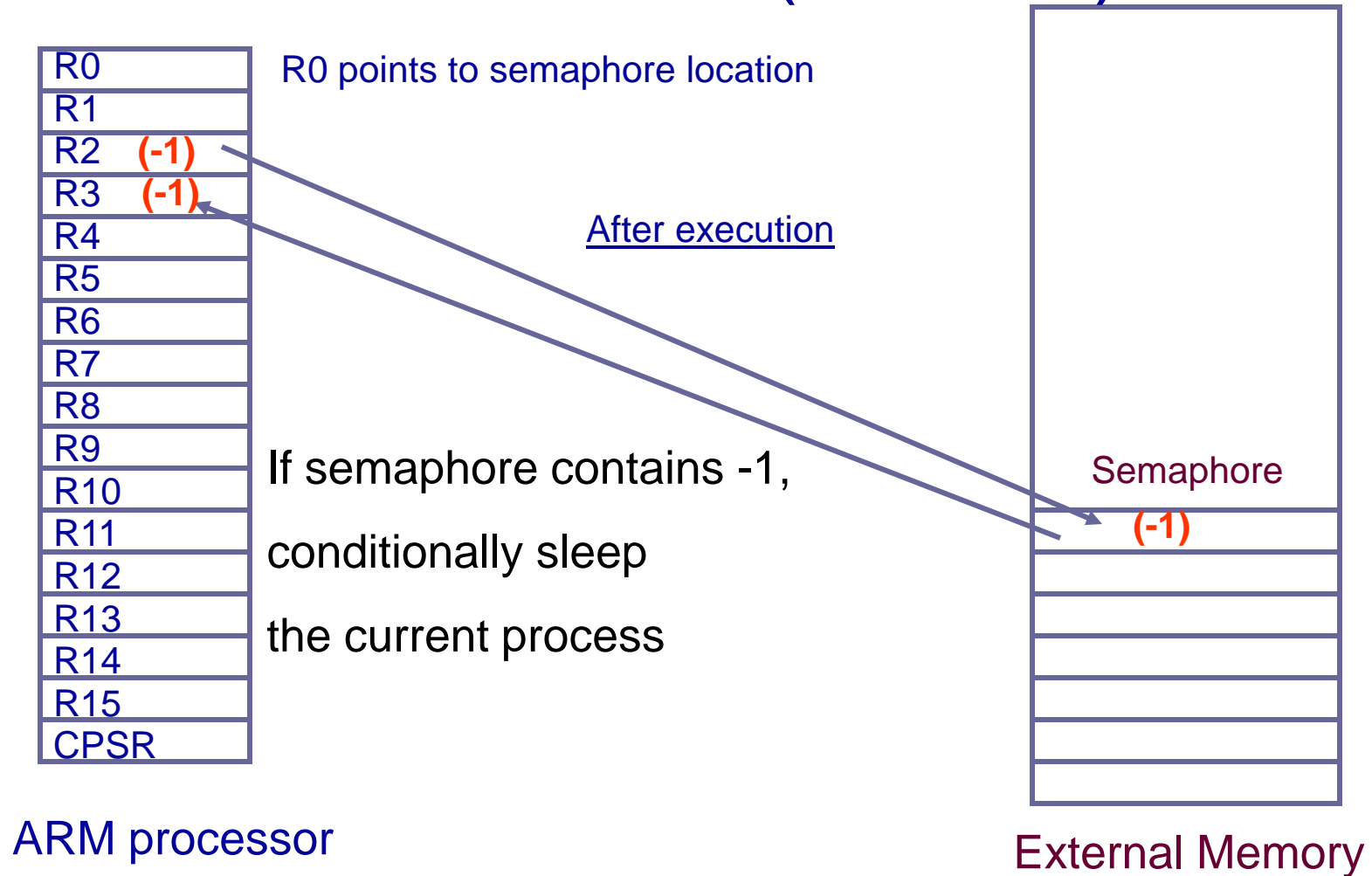
# example

- Before swap ( $0x9000 = 0x12345678$ )
  - $r0 = 0x00000000$
  - $r1 = 0x11112222$
  - $r2 = 0x00009000$
  - **SWP  $r0, r1, [r2]$**
- After swap ( $0x9000 = 0x11112222$ )
  - $r0 = 0x12345678$
  - $r1 = 0x11112222$
  - $r2 = 0x00009000$
- What if we execute **SWP  $r0, r0, [r2]$**  ?

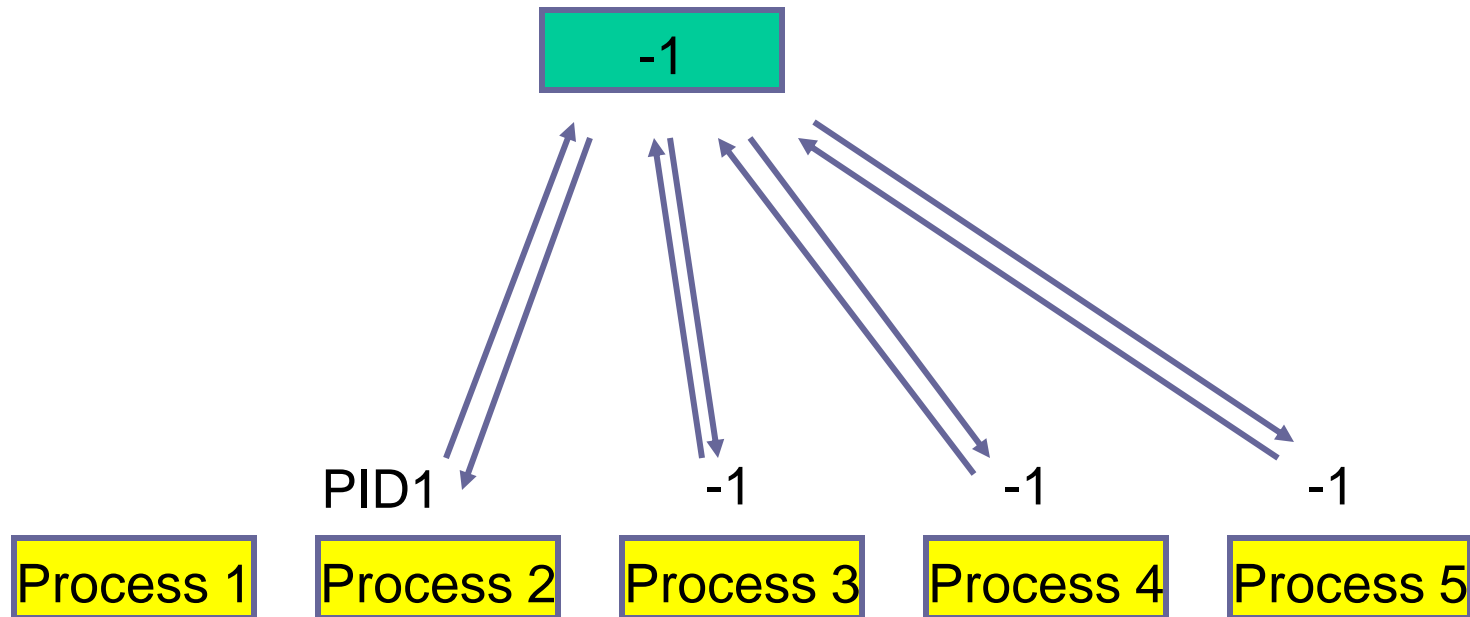
# Look for lock (Case 1)



# Look for lock (Case 1)





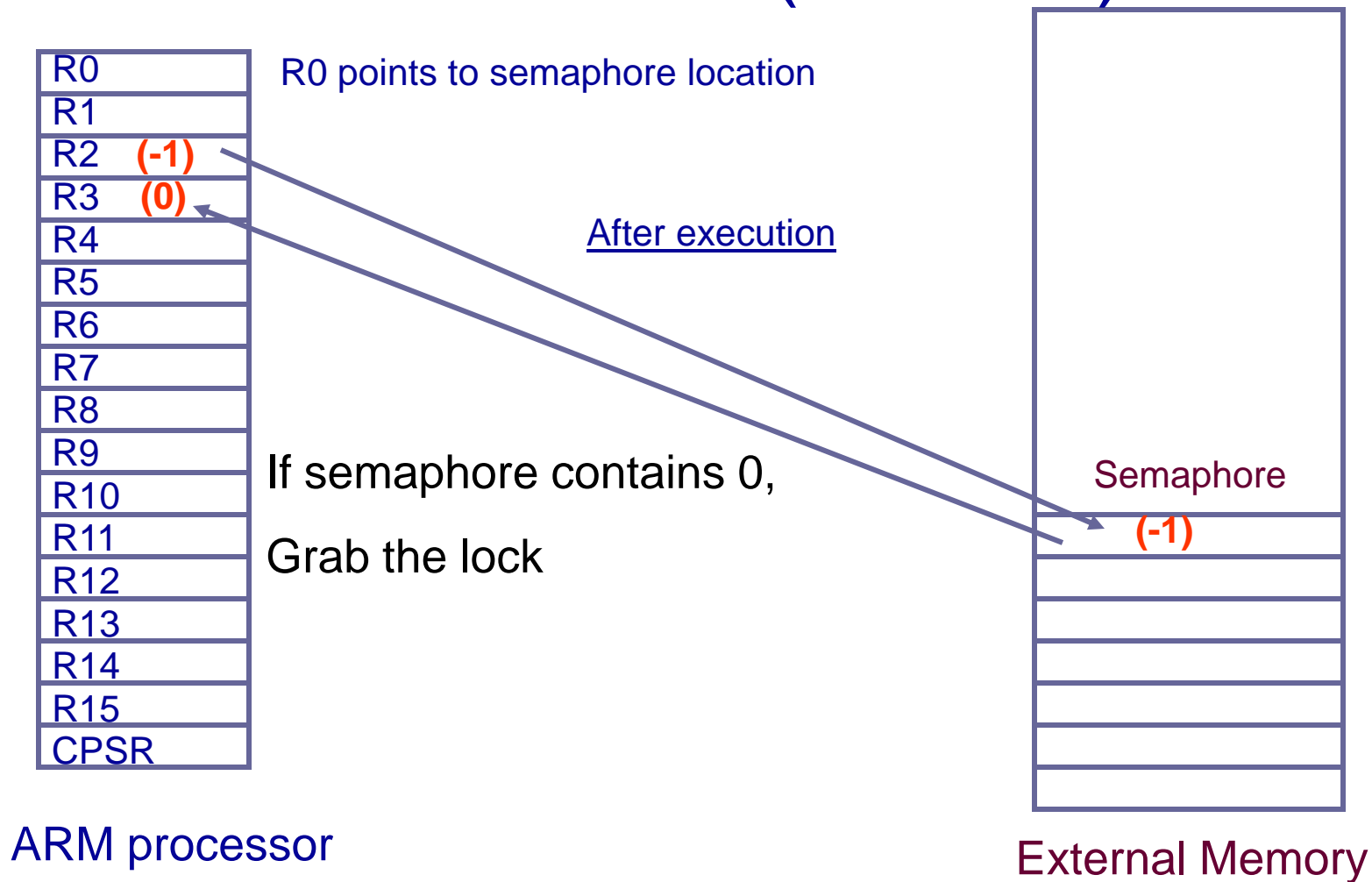


Process 1 has the lock, value = PID1

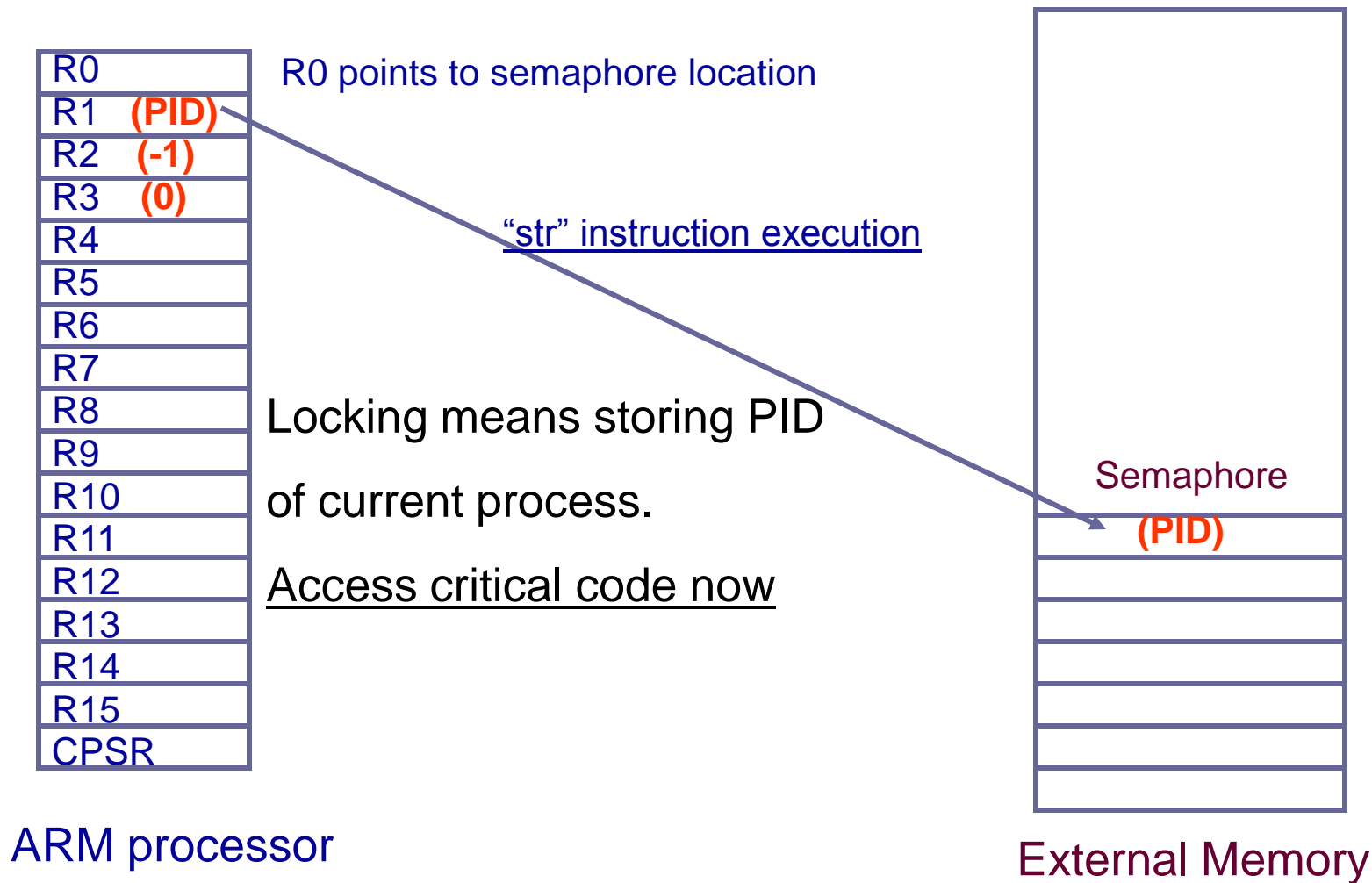
## C-DAC, PUNE



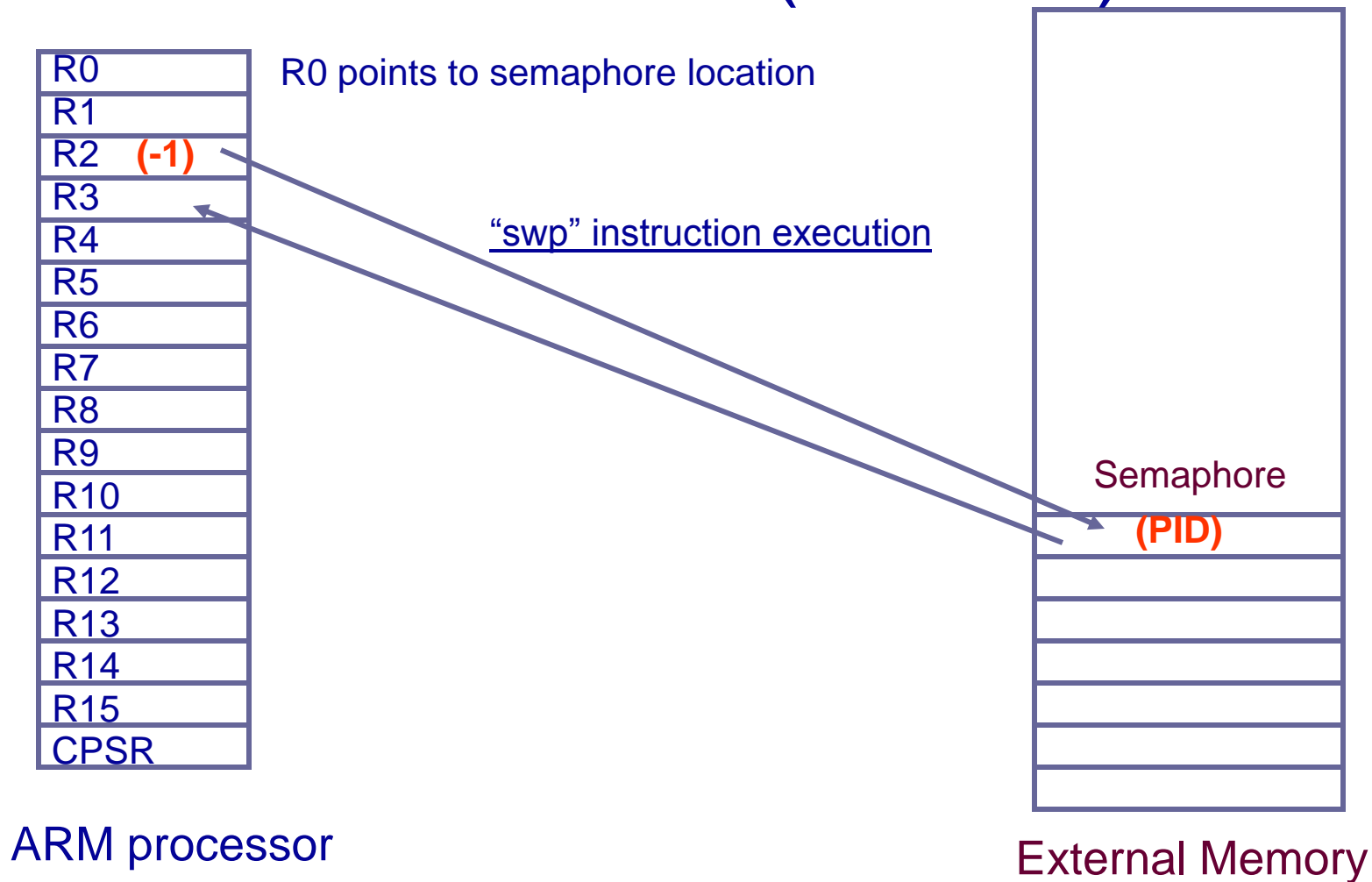
# Look for lock (Case 2)



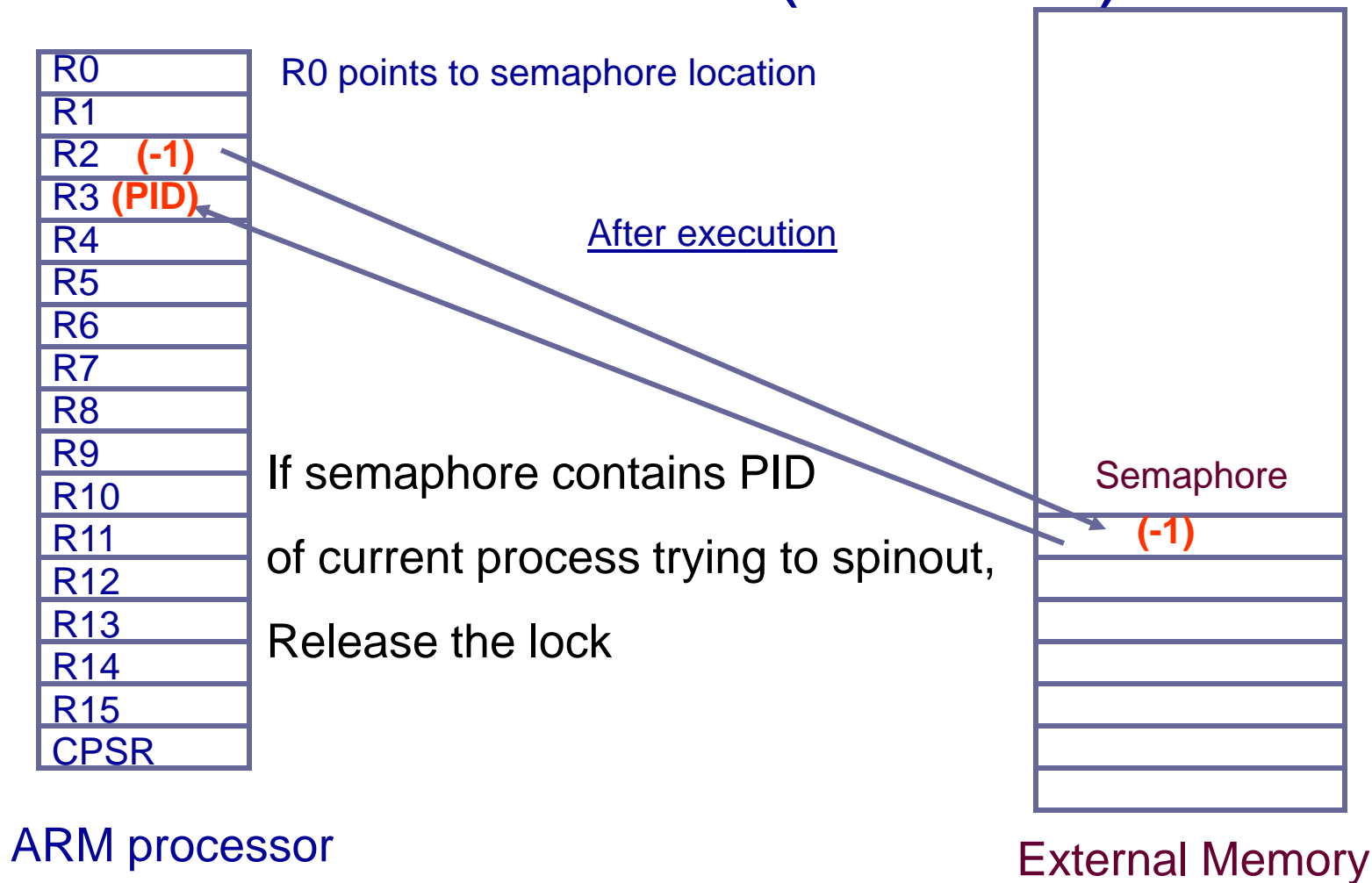
# Grab the lock



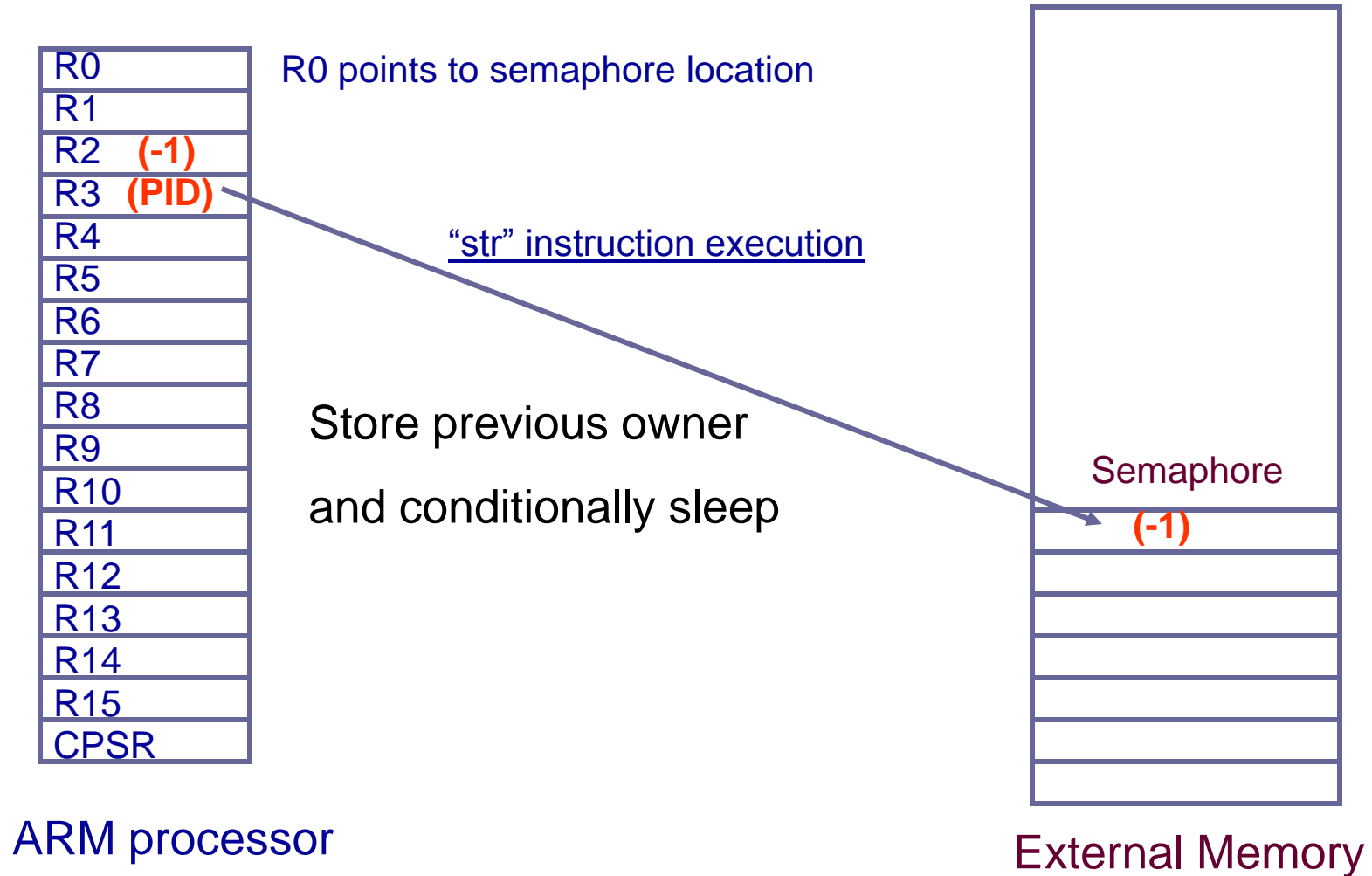
# Look for lock (Case 3)

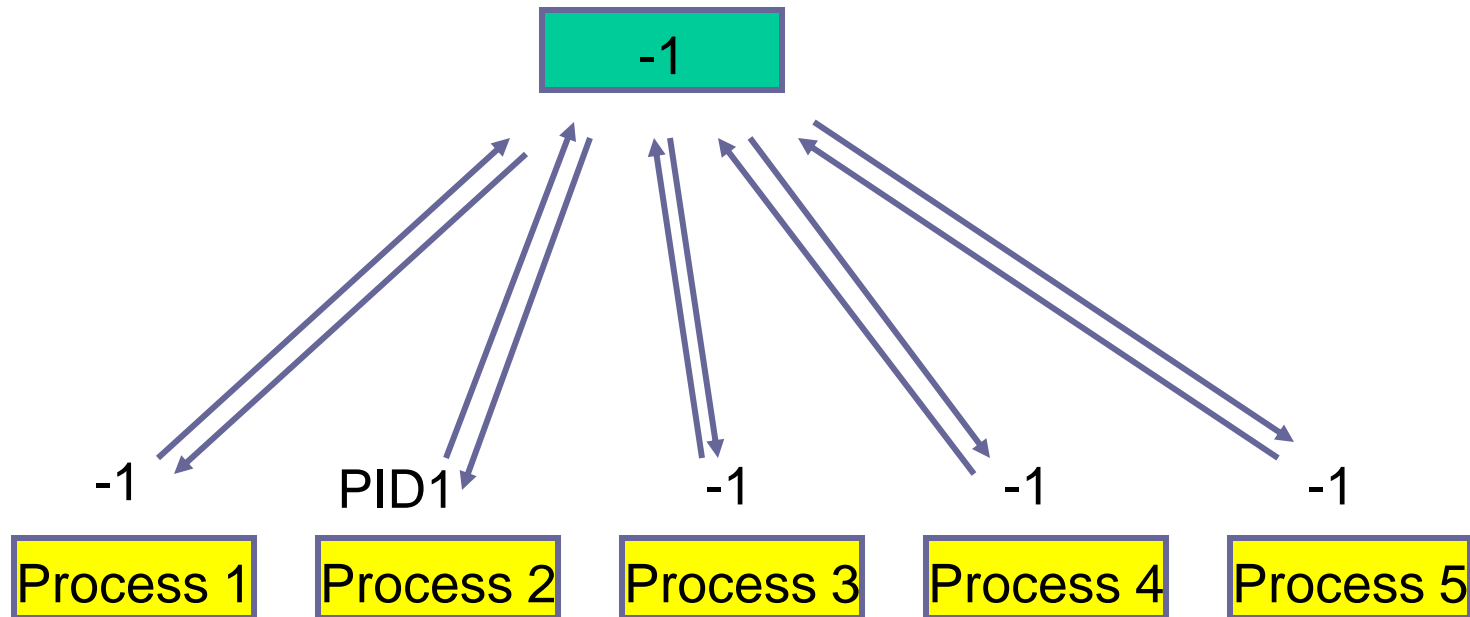


# Look for lock (Case 3)



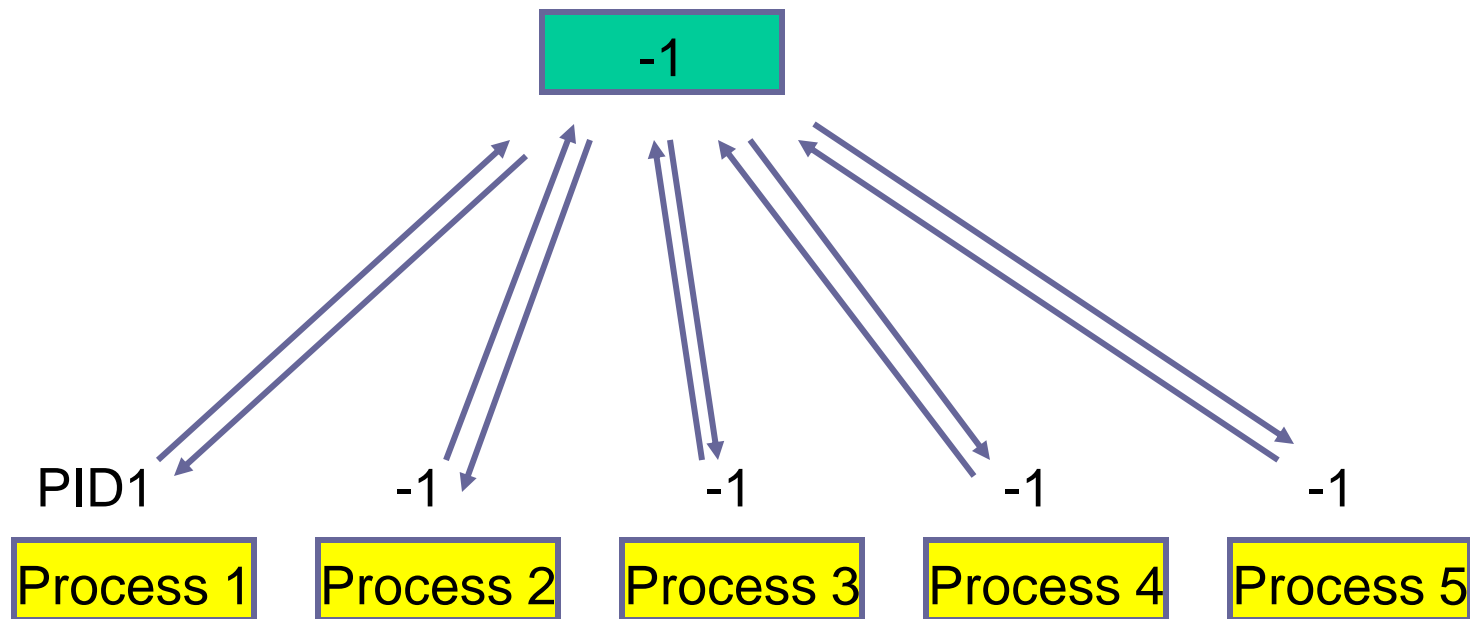
# Release the lock





Process 1 trying to spinout i.e. leave critical code & release the lock  
All other processes trying to grab the lock or spin in.





Process 1 checks the lock value by “swp” instruction

If PID matches, then it will release the lock by setting it to ‘0’

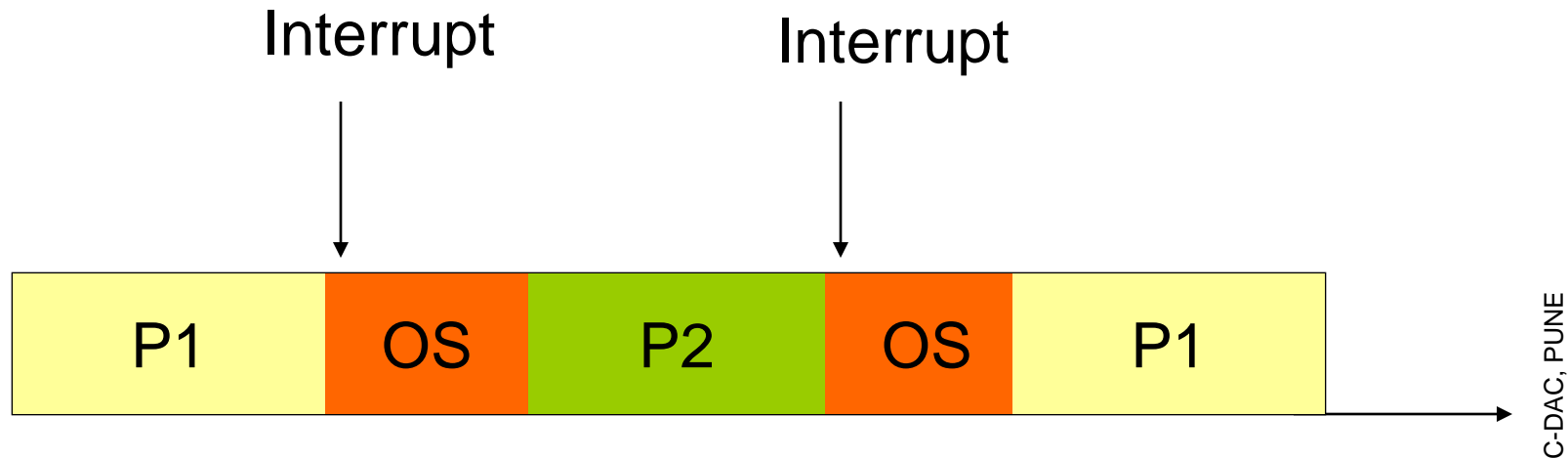
Else it is an error since only process 1 is spinning out and all other processes are spinning in.

# Context switching

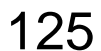
# Context switching

- A process runs in a **context**, which is all the system state that must be established for the process to run correctly.  
**e.g.** processor registers, translation tables in memory, data values of process in memory.
- Processes being switched will store their state in memory as process control block (PCB) and the new process's state will be copied to processor registers and other areas.
- Architectural support is provided by **LDM** and **STM** instructions with various addressing modes. These instructions can load or store all the registers corresponding a particular context.

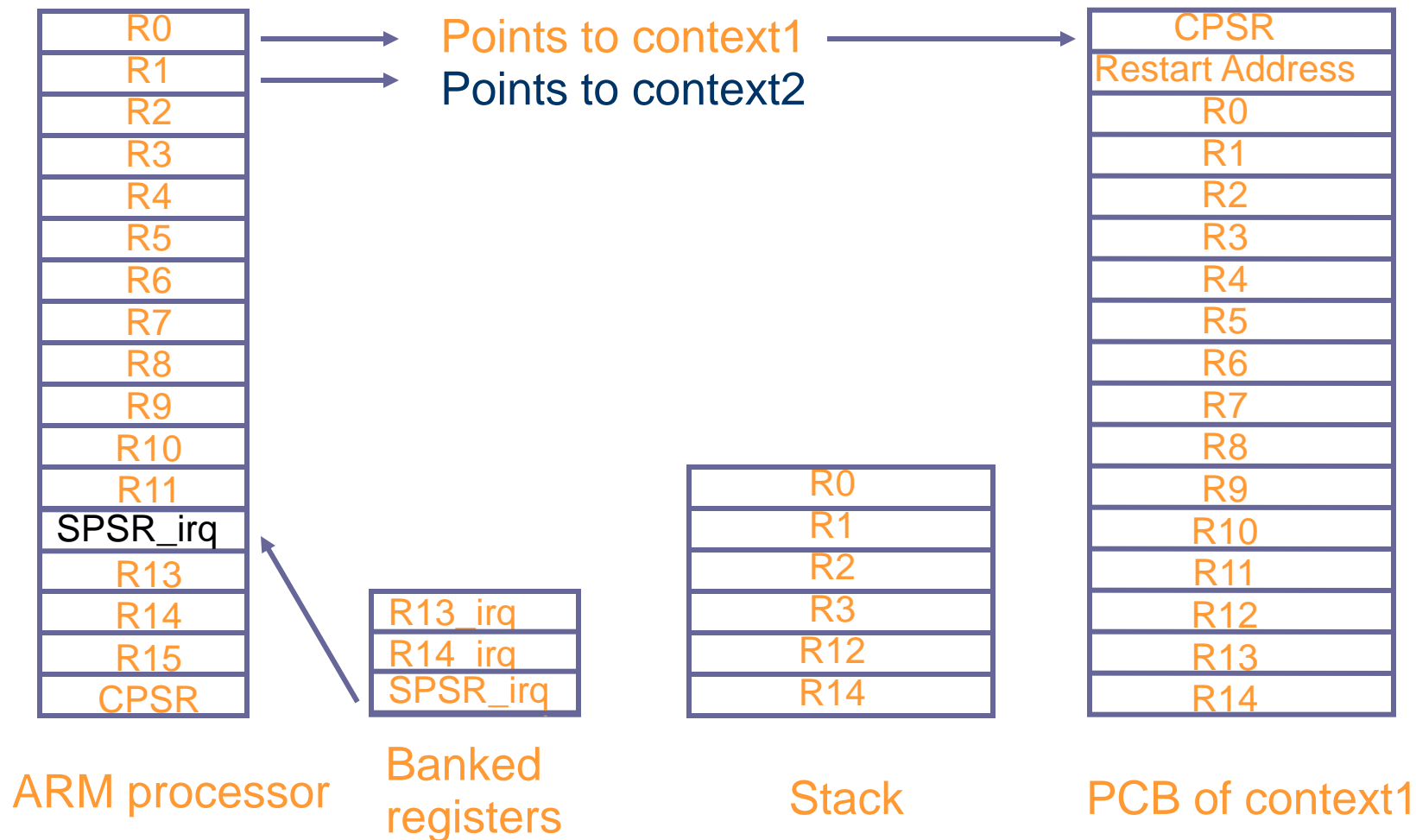
# Context switching process



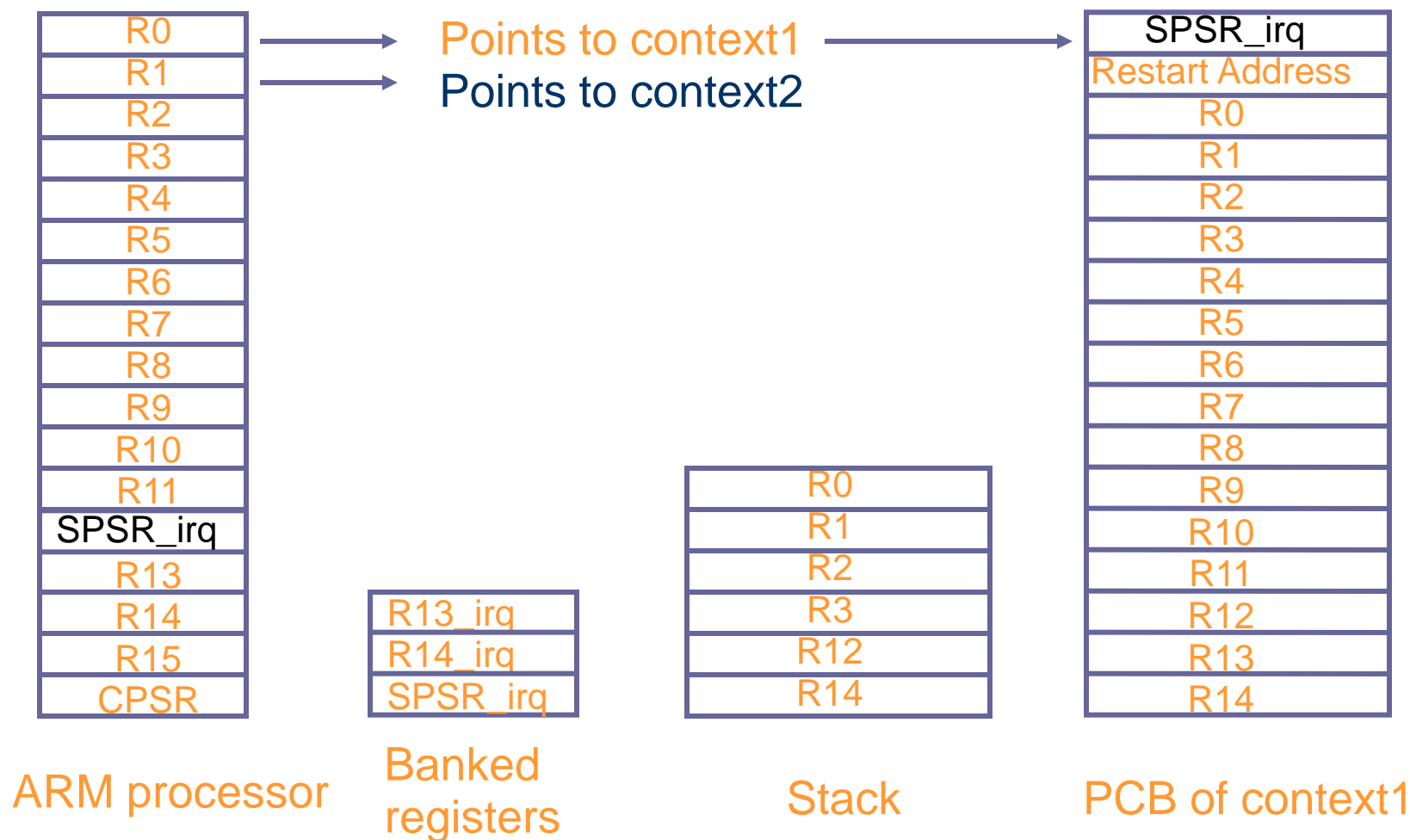
## C-DAC, PUNE



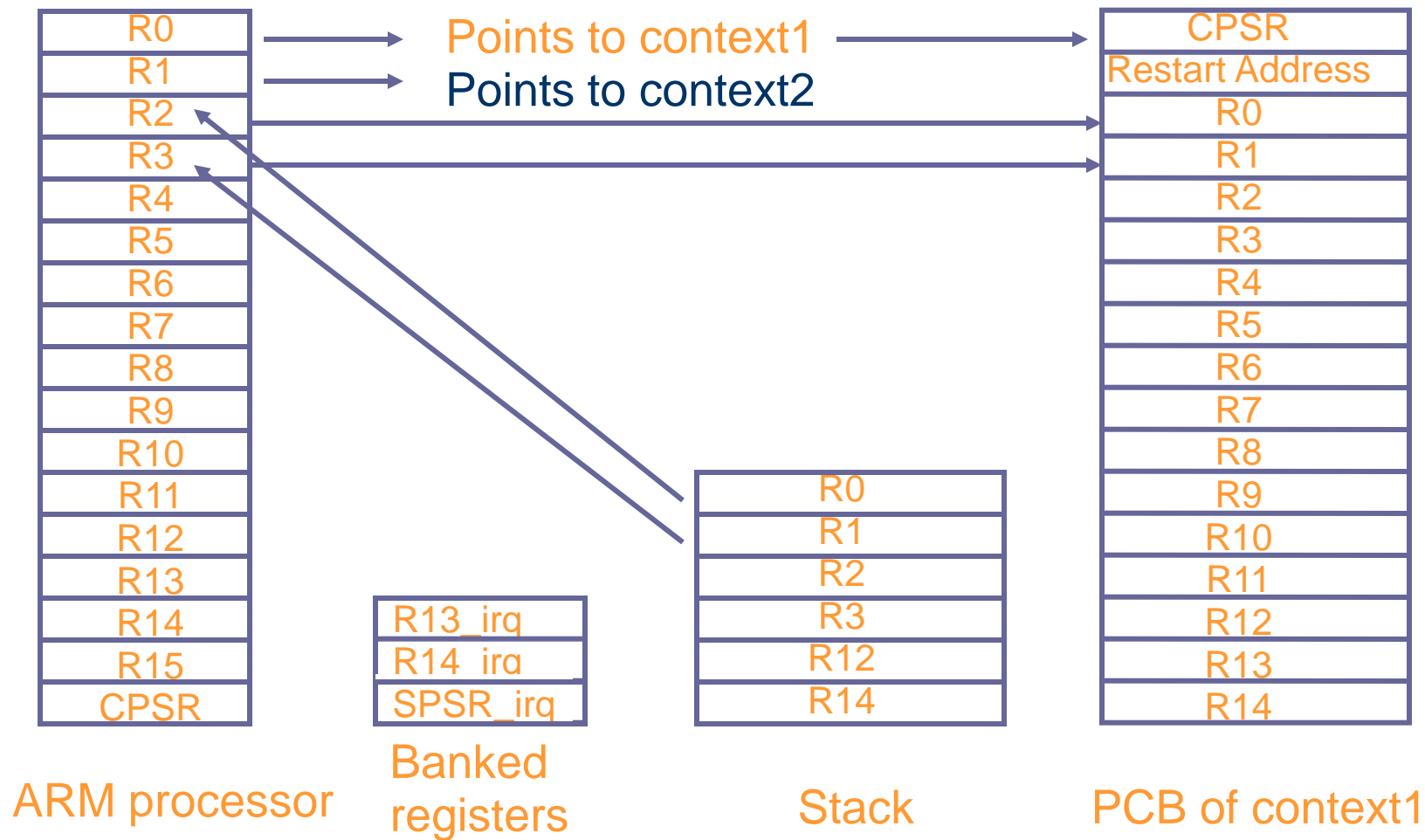
# Save CPSR of interrupted process



# Save CPSR of interrupted process

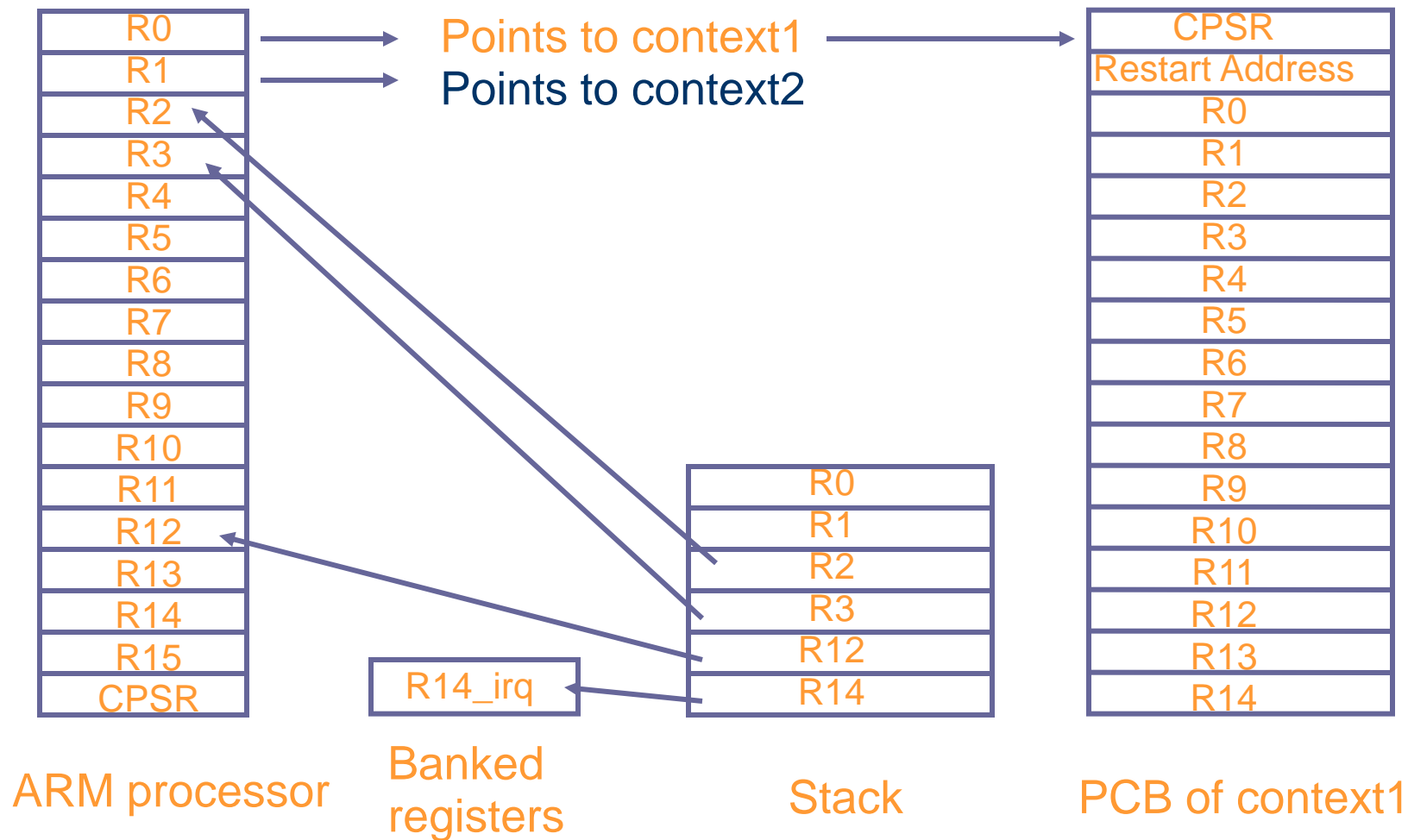


# Save R0, R1 from stack to PCB

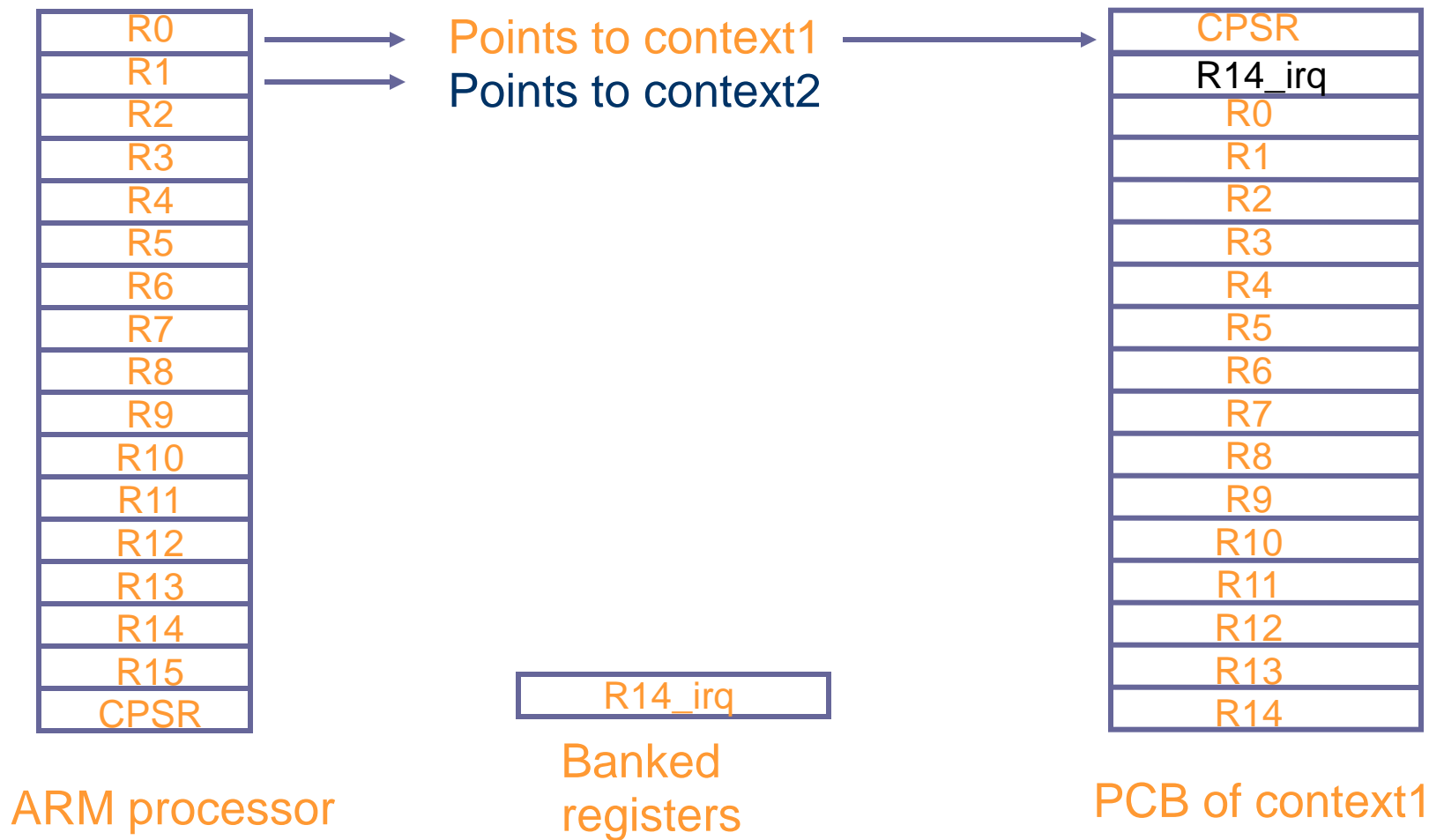




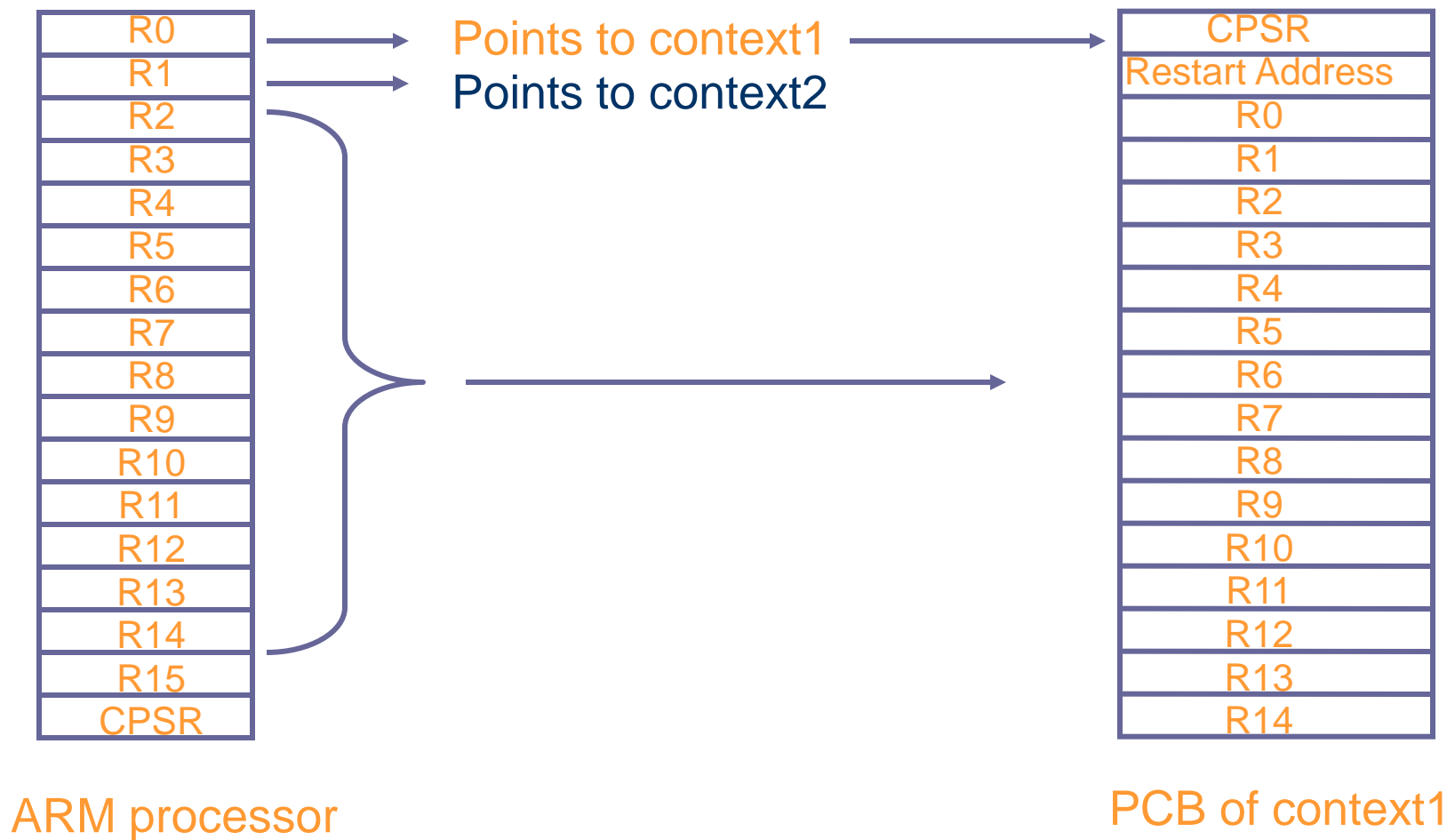
# Save stack to registers



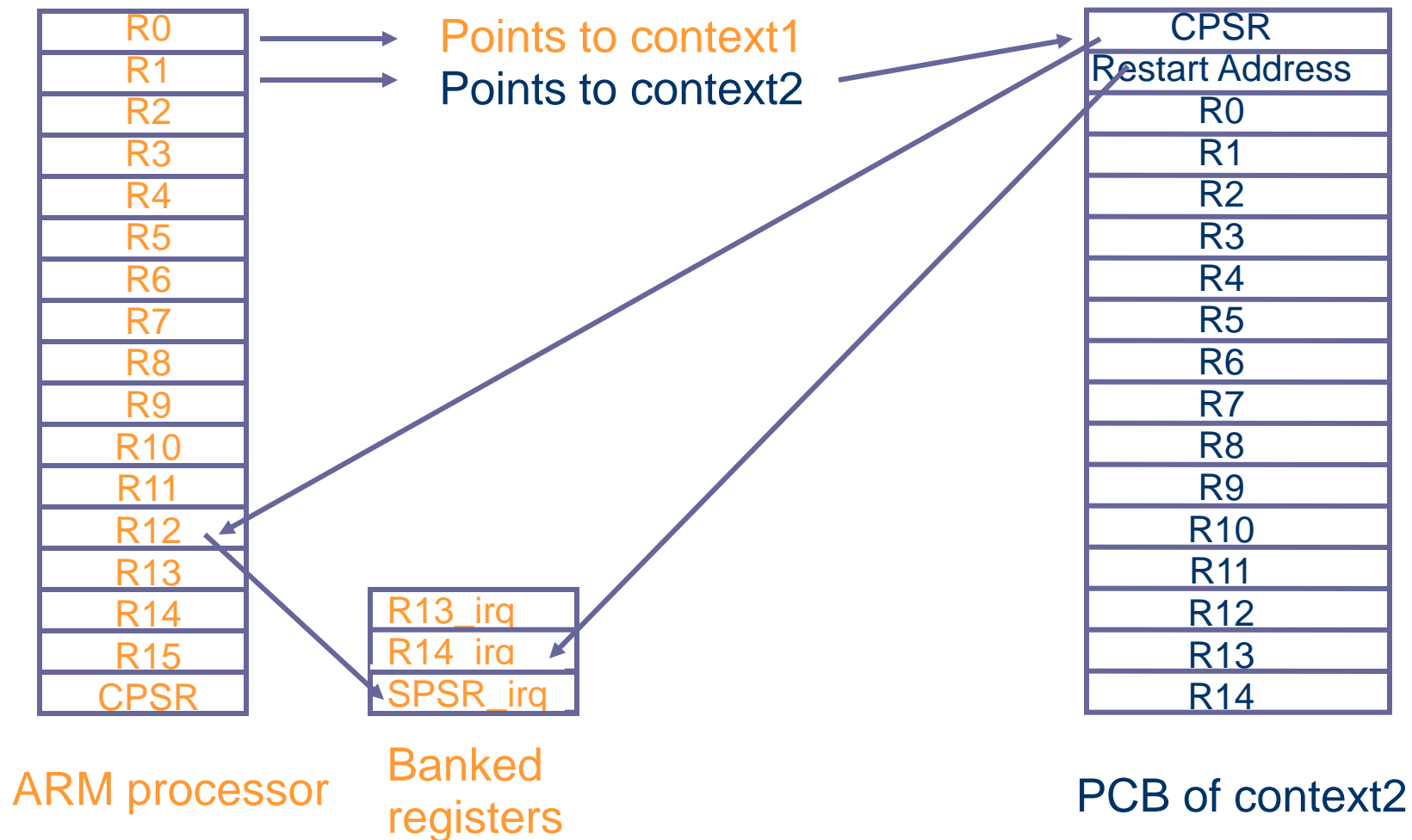
# Save restart address to PCB



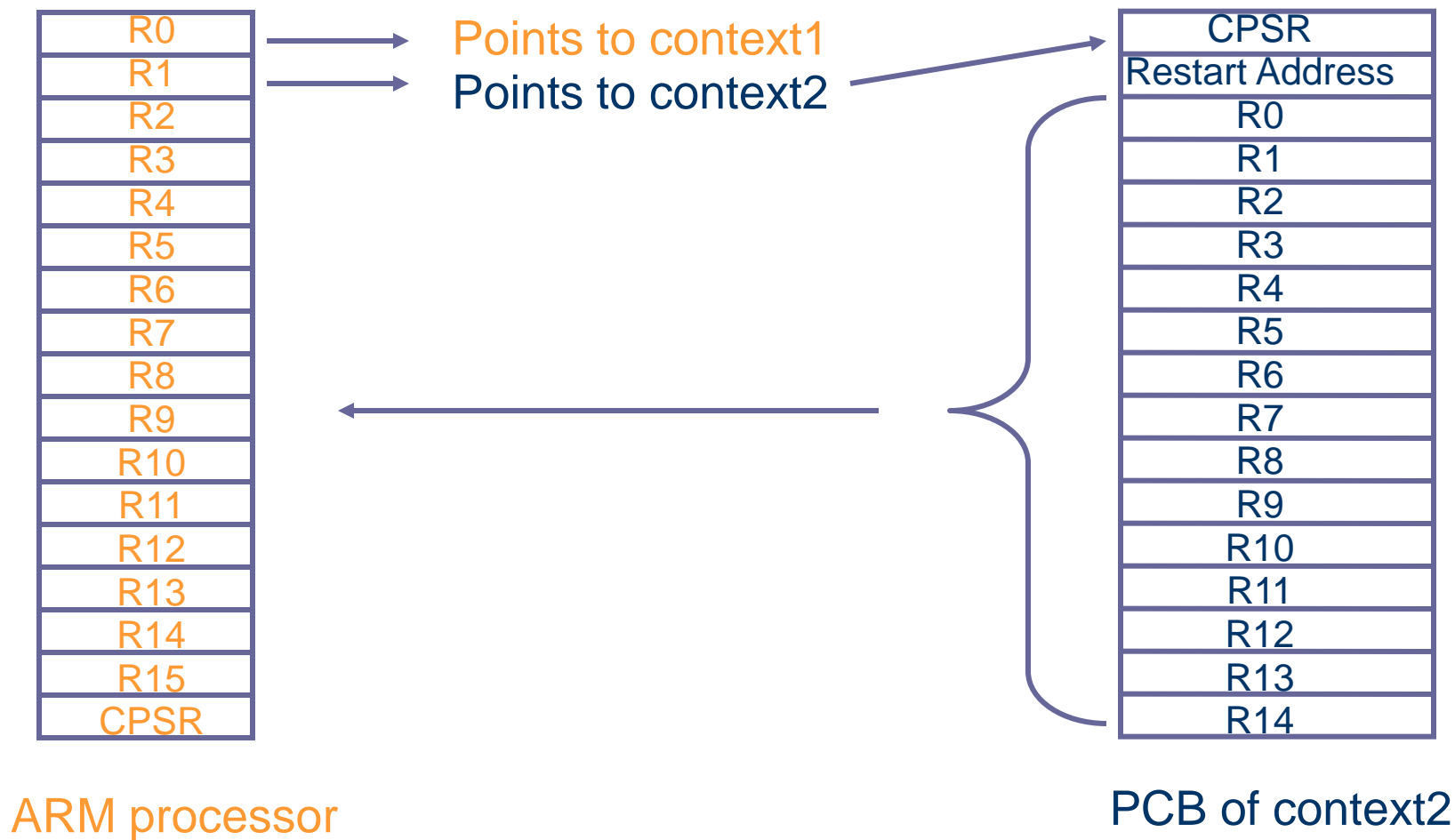
# Save R2-R14 registers to PCB



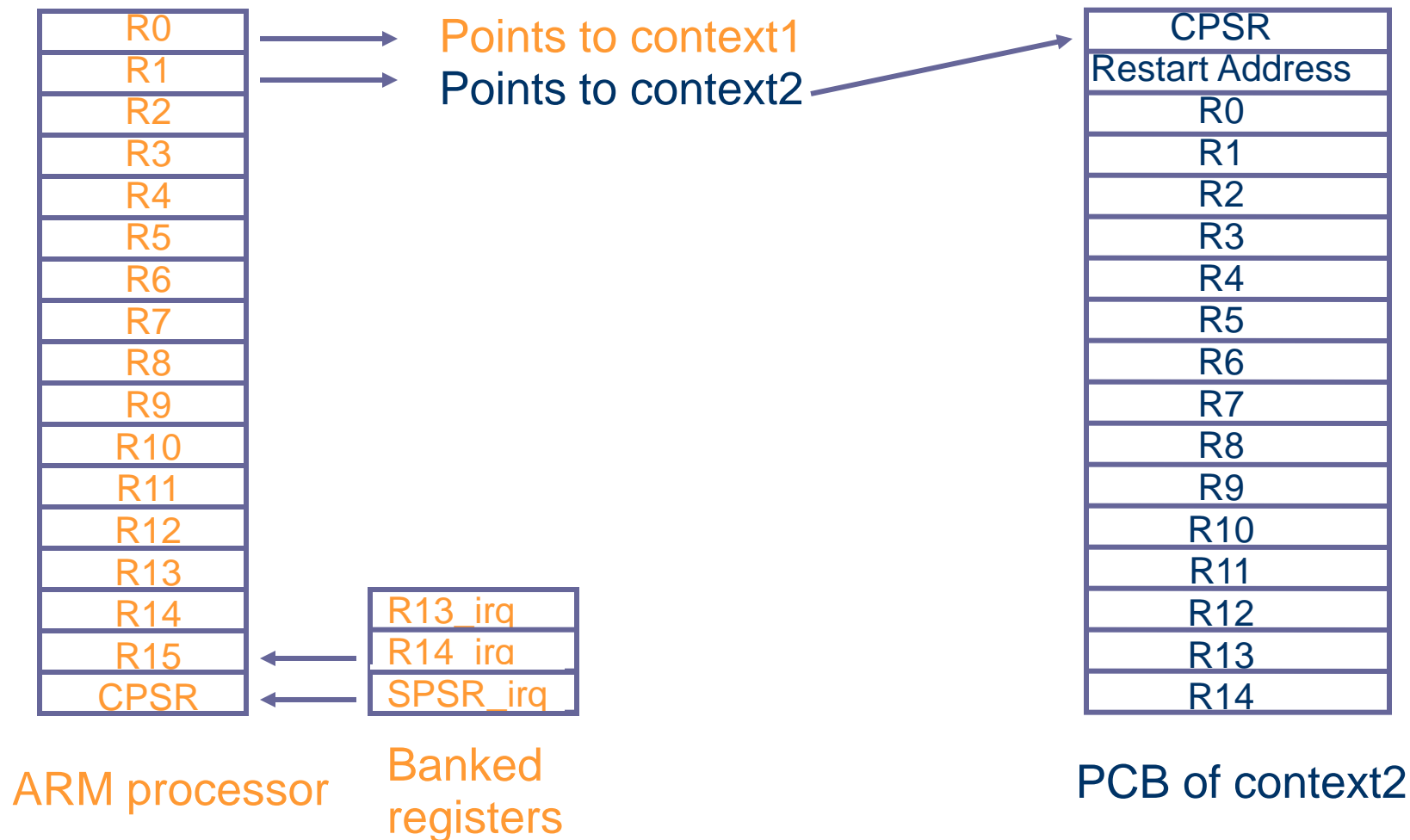
## Load CPSR and return address of context 2 to registers



# Load R0-R14 registers



# Restore context 2



# Input / Output

# Input / Output

- I/O functions can be implemented using memory mapped I/O addressing and interrupt inputs.
- Memory mapped I/O locations behave differently as opposed to normal memory locations. They are “read sensitive”.

e.g. serial port communication.

In this case, user mode programs may not be allowed to access serial port directly, but by using **SWI** instruction to make a OS call or by using C library functions written to use these calls.



# Input / Output

- **DMA:** Useful for supporting high data bandwidth I/O functions. It will typically interrupt processor only if an error occurs or when buffer becomes full. It will however occupy some memory bandwidth.
- **FIQ:** It has more banked registers for saving in overhead incurred in register save and restore. A system employing no DMA support but one bandwidth sensitive I/O data traffic can use FIQ and all other sources can use IRQ line.

# Operating system issues

- I/O functions are typically handled by OS calls.
- OS has to manage scheduling in case some I/O activity is time consuming or waiting for some resources to get free. This can typically be done by returning to other processes until the required I/O device signals an interrupt that it is ready.
- OS also has to be careful, when modifying translation tables and must not make data buffers inaccessible to itself.

# Summary

- OS functions and requirements
- ARM system control co-processor (CP15)
- ARM Protection Unit
- Memory Management Unit
- Synchronization
- Context switching
- Input / Output

Thank you