# DESN2000: Engineering Design & Professional Practice (EE&T)



Week 10

Exceptions and interrupts



David Tsai

School of Electrical Engineering & Telecommunications

Graduate School of Biomedical Engineering

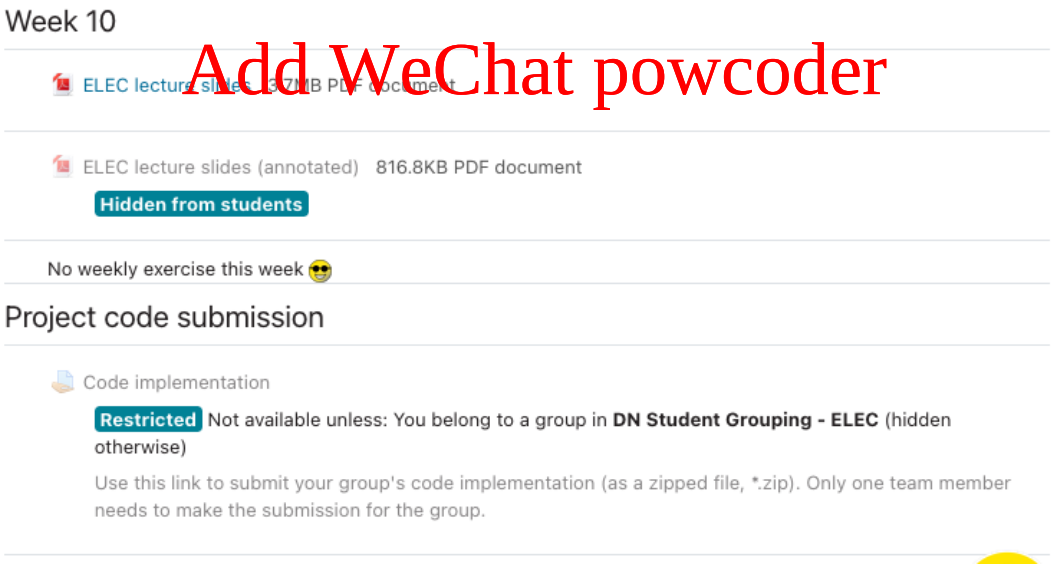
d.tsai@unsw.edu.au

Biomedical Microsystems Lab

# Reminders

- Design Presentation (pitch) due on Friday 8 PM
  - Remember: present evidence for all claimed features, either during the presentation or in the Code Implementation.
- Code Implementation due on Friday 11:59 PM
  - Submit via Moodle, as *.zip file.
  - Only one member of team needs to submit.

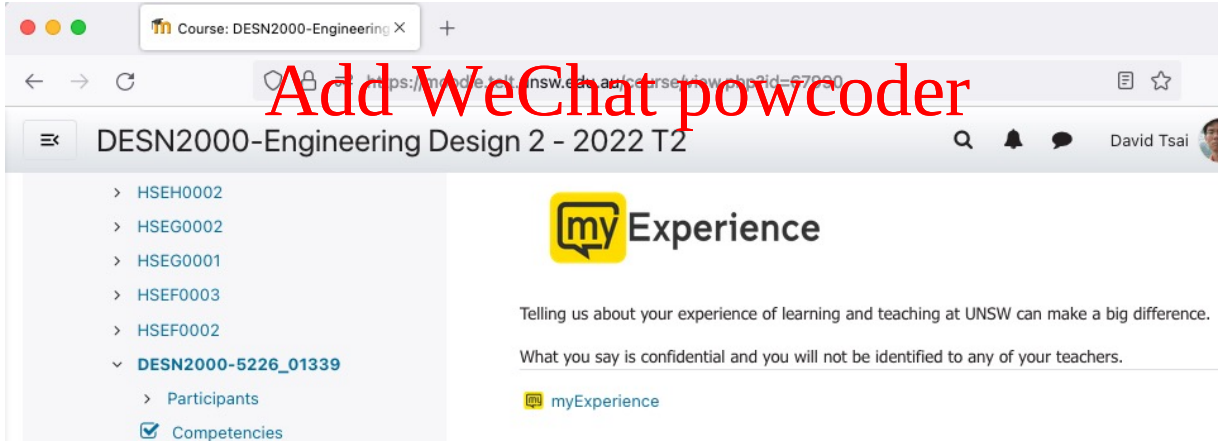




# myExperience

- Available now in Moodle.
- Please fill the survey. We do read these to improve future DESN2000 offerings.





# This week

- Exceptions
- Interrupts and errors
- ARM processor modes
- Configuring the CPSR and SPSR
- Vector table & exception handler

# Exceptions

- Events that break the normal execution flow
  - Benign Moving mouse, pressing keyboard, …
  - Catastrophic Bus error, memory access error, …
- Categorized into two classes:
  1. Interrupts
  2. Error conditions

# Interrupts

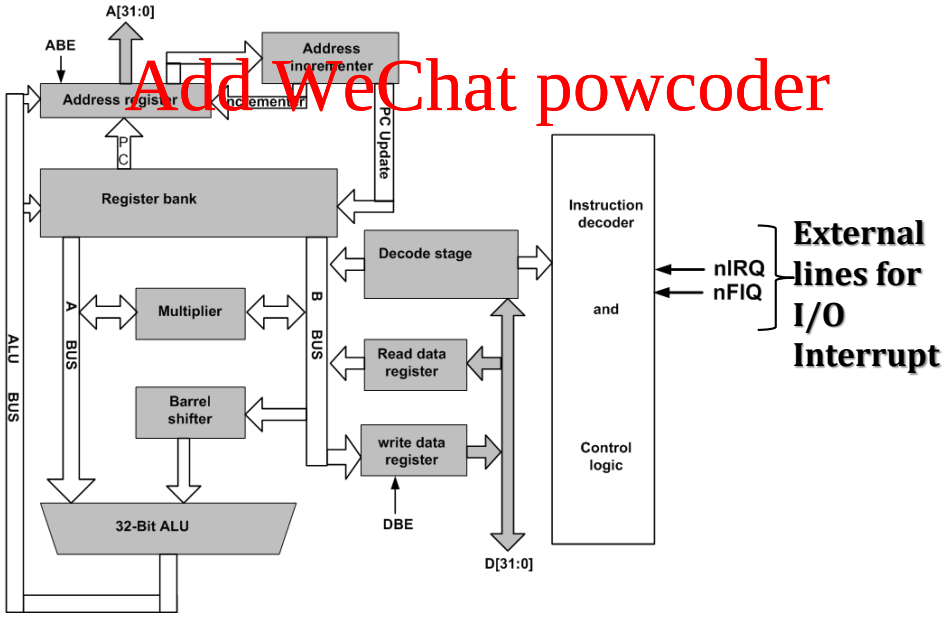
- **Hardware interrupts**
  Exceptions raised *asynchronously* by I/O devices so that they can be served by the CPU.
- **Software interrupt**
  Exception raised within the application (i.e. by the CPU). It is a user defined *synchronous* exception.

# Hardware interrupts

Why?

- An *efficient* way for **I/O devices** to get the CPU's attention.
- ARM has two hardware interrupt lines:
  1. IRQ general use
  2. FIQ high-priority and fast interrupts.
- I/O interrupts are asynchronous (can arrive any time, not sync'ed to the CPU's clock edges).
  - Can occur in the middle of an executing instruction.





# Error conditions

- These **arise within the software** (assembly program).
- Can occur for many reasons, and indeed, could occur more often than hardware interrupts.
- These include:
  1. **Undefined instruction**
     - Floating point instructions emulated in software.
  2. **Data abort**
     - Grabbing *data* in memory that do not physically exist (paging, swap file / virtual memory).
     - Attempts to write *data* in read-only memory region.
  3. **Prefetch abort**
     - Attempts to read an *instruction* from memory and something goes wrong.



**Why the distinction between *Data abort* and *Prefetch abort*?**

# Exceptions in ARM

- ARM processors support these exceptions:
  - **Reset** Processor reset pin is asserted (signalling power-up).
  - **Undef. instruction** The currently executing instruction is not recognized.
  - **Software Interrupt** A user-defined synchronous interrupt instruction.
  - **Prefetch Abort** Attempts to execute an instruction that was not fetched due to illegal address.
  - **Data Abort** Transfer instruction trying to load / store at an illegal address.
  - **IRQ** Interrupt Request pin is asserted.
  - **FIQ** Fast Interrupt Request pin is asserted.

# ARM modes of operation

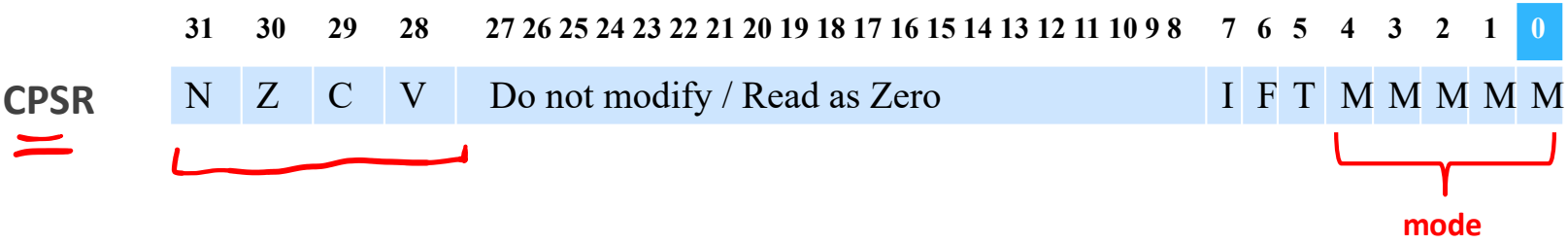
Non-privileged mode

| | | |
|---|---|---|
| 10000 | User | Normal user code |
| 10001 | FIQ | Handling fast interrupts |
| 10010 | IRQ | Handling standard interrupts |
| 10011 | SVC | Handling software interrupts (SWIs) |
| 10111 | Abort | Handling memory faults |
| 11011 | Undef | Handling undefined instruction |
| 11111 | System | Running privileged operating system tasks |



Privileged mode

mode



# User vs privileged modes

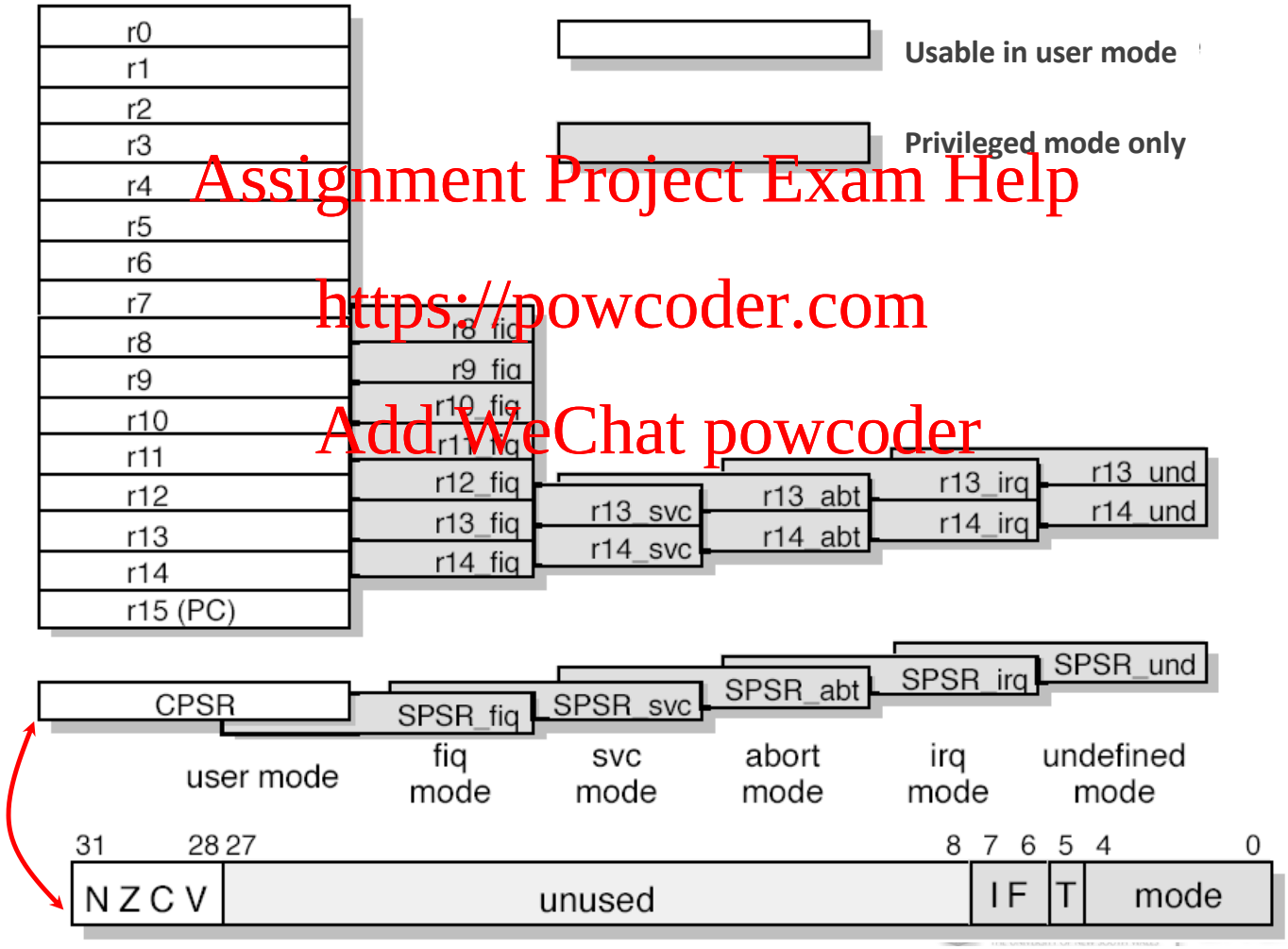
- User applications run in User Mode.
- Privileged modes used for
  - Servicing hardware / software interrupts.
  - Handling undefined instructions.
  - Running privileged tasks.



- Privileged modes ⇒ User mode: always OK.
- User mode ⇒ Privileged modes: only via controlled mechanisms:
  - Hardware interrupts
  - Software interrupts
  - Error conditions

# ARM modes and registers

- **Except for System mode**, every ARM mode has a set of dedicated registers that "swaps in" and replaces the normal regs upon switching into the privileged mode.

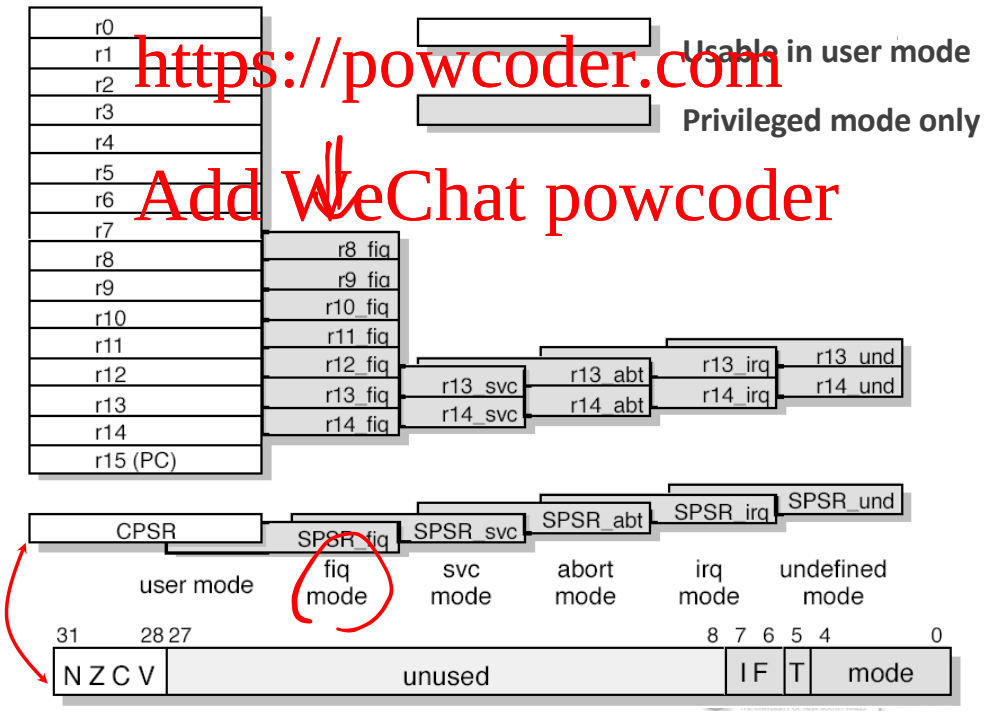


# ARM modes and registers

- The registers tell you a lot about the design intents of each mode.
  - Why does FIQ mode have more dedicated registers than the other modes?
  - Why doesn't System mode have any dedicated registers?

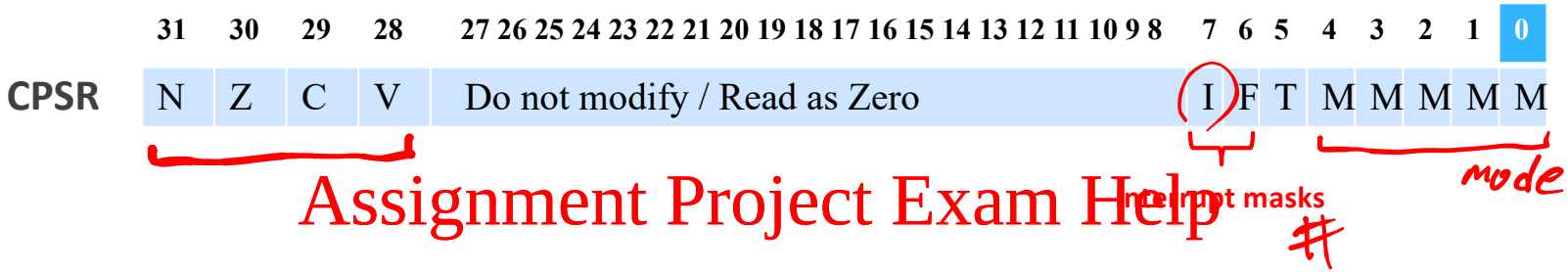
- **FIQ should be fast (Gbit Ethernet, HDDs, etc). ARM swap in/out (using dedicated circuits) lots of extra regs for you to use.**
- **System mode is intended to be used by OS tasks needing privileged accesses but doesn't need extra registers.**





# ARM hardware interrupts

- The CPSR affects how the processor handles hardware interrupts.



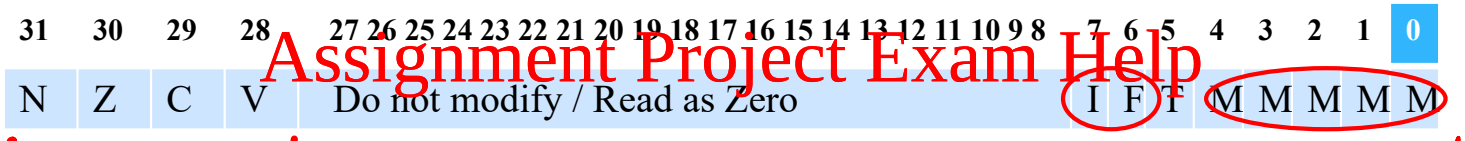



- Setting the mask bit high disables the corresponding interrupt:

| Mask bit | Purpose |
|---|---|
| F | Fast interrupt (FIQ) mask bit |
| I | Normal interrupt (IRQ) mask bit |

# Configuring the C/SPSR

- Modify the CPSR (current program status register) to change:
  - Conditional flags
  - Disable / enable interrupts (IRQ, FIQ)
  - Mode






- Up to two PSRs are visible at any time:
  - CPSR
  - SPSR (*saved* program status register, the previous mode's PSR).
- User and System modes **do not** have SPSR.

- In User mode: only the condition flags can be edited, all other bits are protected.
- In the privileged modes: the entire CPSR can be edited.

# Configuring the C/SPSR

- Copying a general register into a PSR:

```
MSR CPSR, R0          ; R0 into CPSR
MSR SPSR, R0          ; R0 into SPSR
MSR CPSR_f, R0        ; flag bits of R0 into CPSR
MSR CPSR_f, #1<<28    ; flag bits (immediate) into CPSR
```

Weird _f notation: will discuss shortly

- Copying the PSR into a general register:

```
MRS R0, CPSR          ; CPSR into R0
MRS R0, SPSR          ; SPSR into R0
```



- Can only edit the SPSR of the current mode.
  - Example: if in IRQ mode, you cannot edit the SPSR of FIQ.
  - Switch to FIQ mode first, if you want to change FIQ's SPSR (SPSR_fiq).

# Configuring the C/SPSR

- Two ways to modify the CPSR / SPSR.

- Method 1: read-write-modify strategy.
  1. Transfer PSR to general register using MRS
  2. Modify relevant bits
  3. Transfer updated value to PSR register using MSR
- Example: changing to Supervisor mode



```
MRS a1, CPSR
BIC a1, a1, #0x1F   ; clear mode bits
ORR a1, a1, #0x13   ; set SVC mode
MSR CPSR, a1
```

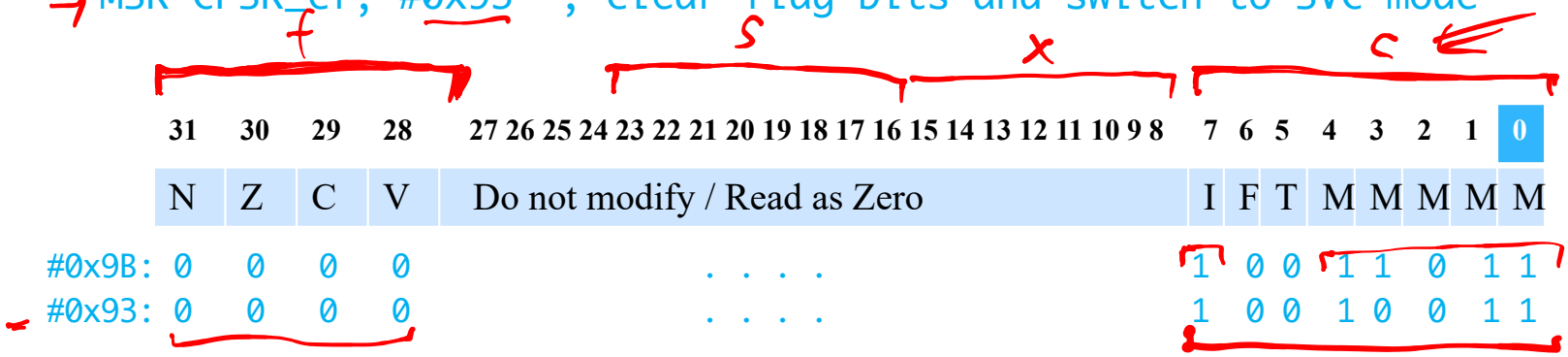
| | | |
|---|---|---|
| 10000 | User | Normal user code |
| 10001 | FIQ | Handling fast interrupts |
| 10010 | IRQ | Handling standard interrupts |
| 10011 | SVC | Handling software interrupts (SWIs) |
| 10111 | Abort | Handling memory faults |
| 11011 | Undef | Handling undefined instruction |
| 11111 | System | Running privileged operating system tasks |

# Configuring the C/SPSR

- Method 2: using the bit-field flags.
  - The CPSR / SPSR register is divided into 4 sections:
    - **c** control bits, [7-0]: including I/F disable, Thumb mode, processor mode.
    - **x** extension bits, [15-8]
    - **s** status bits, [23-16]
    - **f** flag bits, [31-24]: including NZCV flags
  - Combine flags for compound effects.
  - Example:



```
MSR CPSR_c, #0x9B   ; switch to Undef mode, mask IRQ
MSR CPSR_c, #0x93   ; switch to SVC mode, mask IRQ
MSR CPSR_f, #0x93   ; clear flag bits, ctrl bits ignored
MSR CPSR_cf, #0x93  ; clear flag bits and switch to SVC mode
```



# Configuring the C/SPSR

- MRS moves the CPSR or the SPSR of the current mode into a general-purpose register:

```
MRS{cond} Rd, CPSR
MRS{cond} Rd, SPSR
```

- Binary format:


  - For CPSR, R = 0
  - For SPSR, R = 1

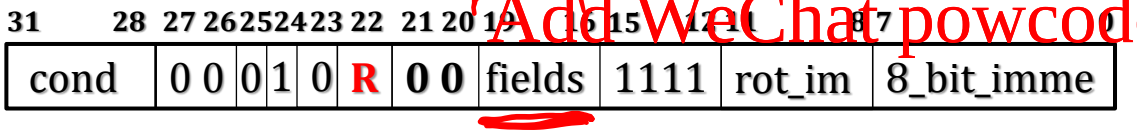
# Configuring the C/SPSR

- MSR moves a general purpose register value into the CPSR / SPSR of the current mode.
- Useful for changing condition flags, interrupt masks or processor mode.
- Syntax:

```
MSR{cond} CPSR<_fields>, #32bit_immediate
MSR{cond} CPSR<_fields>, Rm
MSR{cond} SPSR<_fields>, #32bit_immediate
MSR{cond} SPSR<_fields>, Rm
```

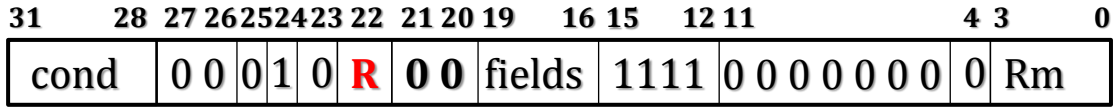


- Binary format (immediate):



# Configuring the C/SPSR

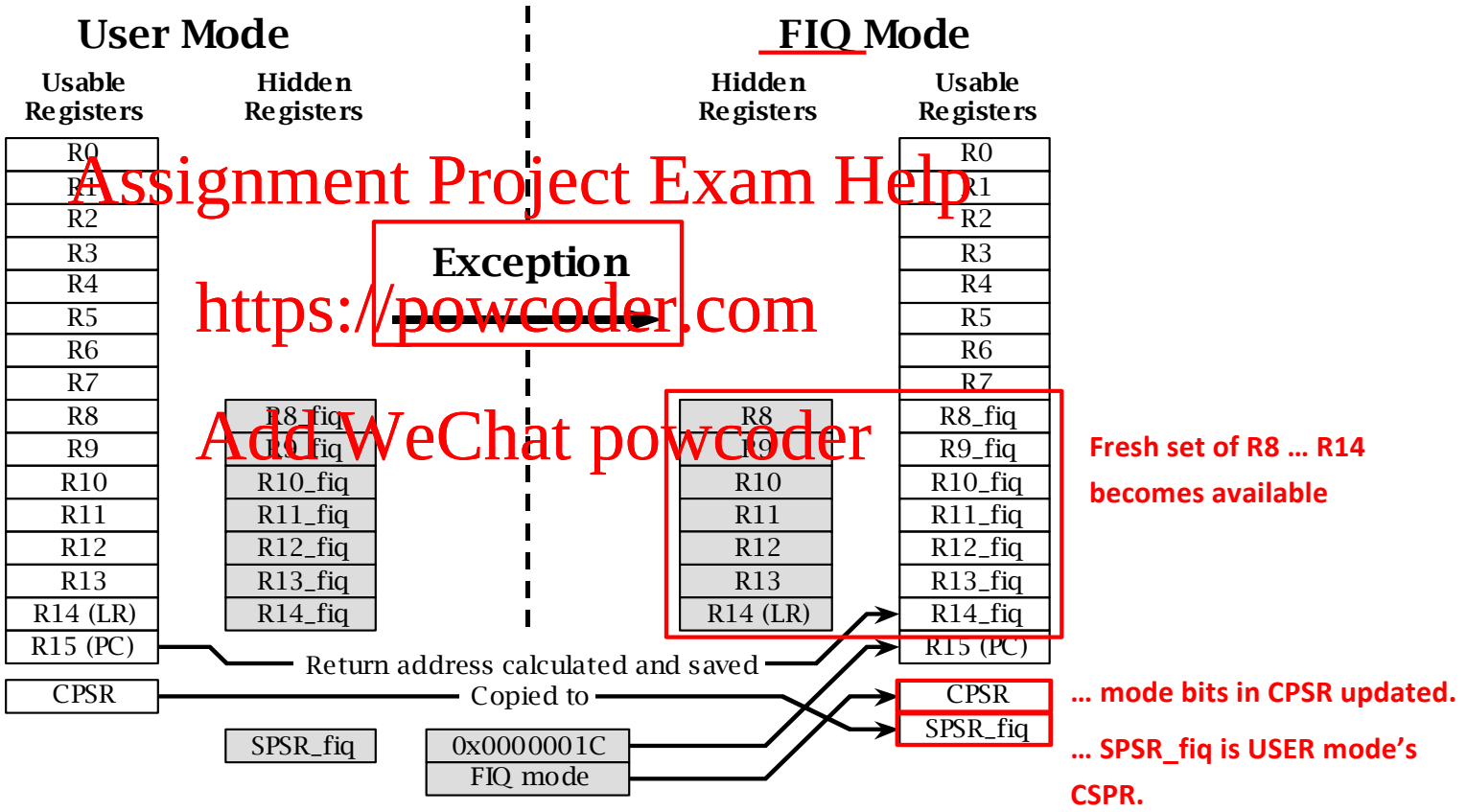
- Binary format for MSR (register):




  - R = 0 for CPSR
  - R = 1 for SPSR
  - fields:
    - c (bit 16)
      Modifies only the control bits ([7:0]) of CPSR or SPSR
    - x (bit 17)
      Modifies only the extension bits ([15:8]) of CPSR or SPSR
    - s (bit 18)
      Modifies only the status bits ([23:16]) of CPSR or SPSR
    - f (bit 19)
      Modifies only the flags bits ([31:24]) of CPSR or SPSR

# Mode switching

- Hardware does the register swap (fast, 1-clk cycle).





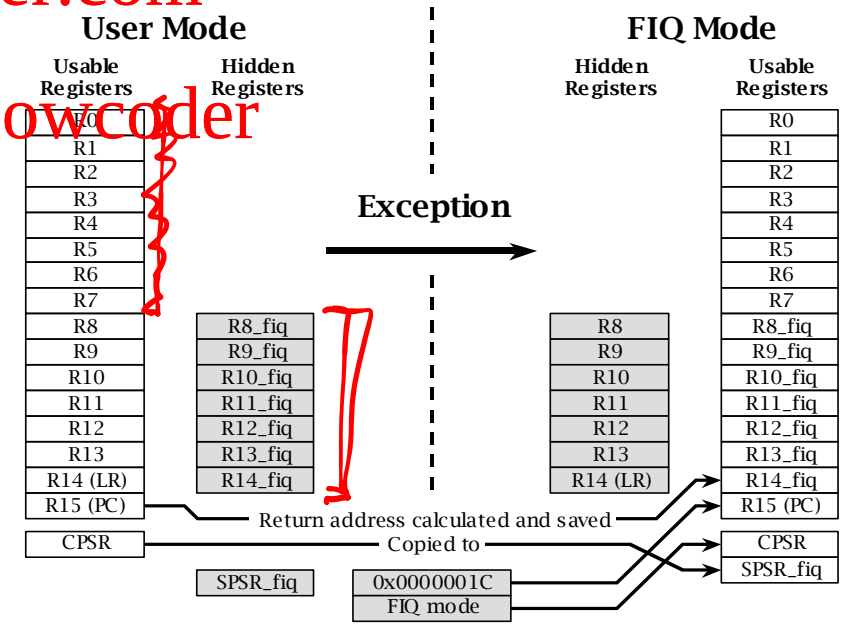

- The banked registers replace the ordinary ones during the exception. The ordinary ones cannot be seen directly.

# Mode switching

- Questions: while in FIQ mode:
  - What does the mode bits ([5:0]) of CPSR indicates? What about those of the SPSR_fiq?
  - Can you access registers R8 ~ R14 of USER mode?



- **CPSR[5:0] indicate "FIQ"**
- **SPSR_fiq[5:0] would indicate the prev. state before the FIQ occurred… so it's "User".**
- **Cannot access R8 ~ R14 of USER mode.**



# Mode switching: to exception handler

- When exception occurs, ARM processor takes the following steps:
  1. Save current (USER mode) CPSR in SPSR_<mode>.
  2. Some USER mode registers become invisible, replaced by banked registers.
  3. Update CPSR:
     - Switch to ARM state if it was in THUMB state.
     - Setting the mode bits in CPSR.
     - IRQ automatically disabled (CPSR[7] = 1) on entry to all exceptions.
     - FIQ automatically disabled (CPSR[6] = 1) on entry to RESET and FIQ exceptions.
  4. Copy address of next instruction (return address) into LR_<mode>.
  5. Loads address of the **exception handler** into PC.
     Exception handler – a piece of code that deals with the exception condition.
- The processor does the above for you.



- **We use the names "LR_fiq", "R13_fiq", etc… But in practice you still write "LR", "R13", etc., respectively, in your assembly code to refer to these regs while in FIQ mode.**

# Mode switching: from exception handler

- To return to the USER mode, the programmer must:
  1. Move the return address (in LR_<mode>) into PC.
  2. Restore original (before the exception occurred) CPSR.
- Must be done simultaneously, using the 'S' variant:

```
MOVS PC, LR                    ; restore PC and CPSR
```

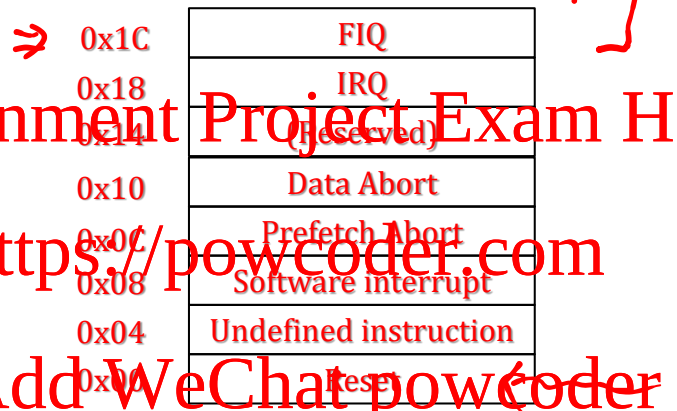
- Similarly achieved using ^ qualifier with LDMFD:

```
LDMFD SP!, {R0-R1, PC}^        ; restore regs, PC and CPSR
```

- On executing above instruction:
  1. Processor goes into USER mode.
  2. Banked registers are hidden, USER mode registers become available.

# The vector table

- Beside SYSTEM exception, there is a dedicated block of code, called an **exception handler**, used by the processor to handle the exception.
- The **vector table** has an entry for every exception type:






PC := 0x00
B start_up

- When an exception occurs, the PC is automatically loaded with the matching exception handler's address from the vector table.
- Besides FIQ, each entry is 32 bits… enough for one instruction.
- 0x00 RESET is where PC starts upon system power-up.

# The vector table

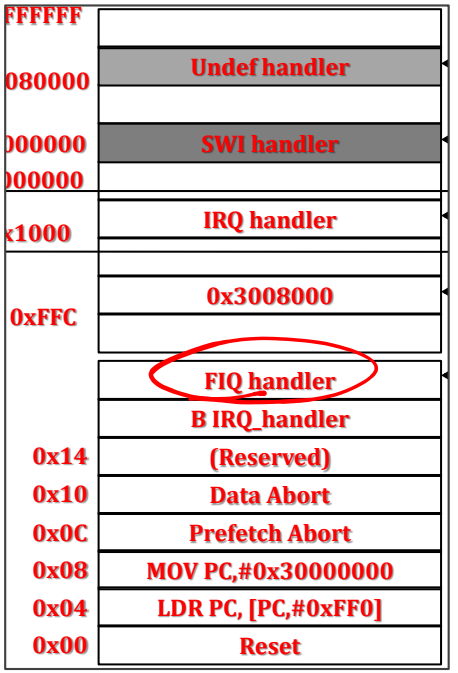
- The vector table contains change-of-flow instruction(s):
  - Use `B` Jump to exception handler that is within 32 MB range.
  - Use `MOV` Update PC with a new address representable by a 8-bit value with rotation.
  - Use `LDR` Data can be stored in instruction memory and accessed using an PC-offset (assisted by literal pools), e.g. `LDR PC, [PC, offset]`



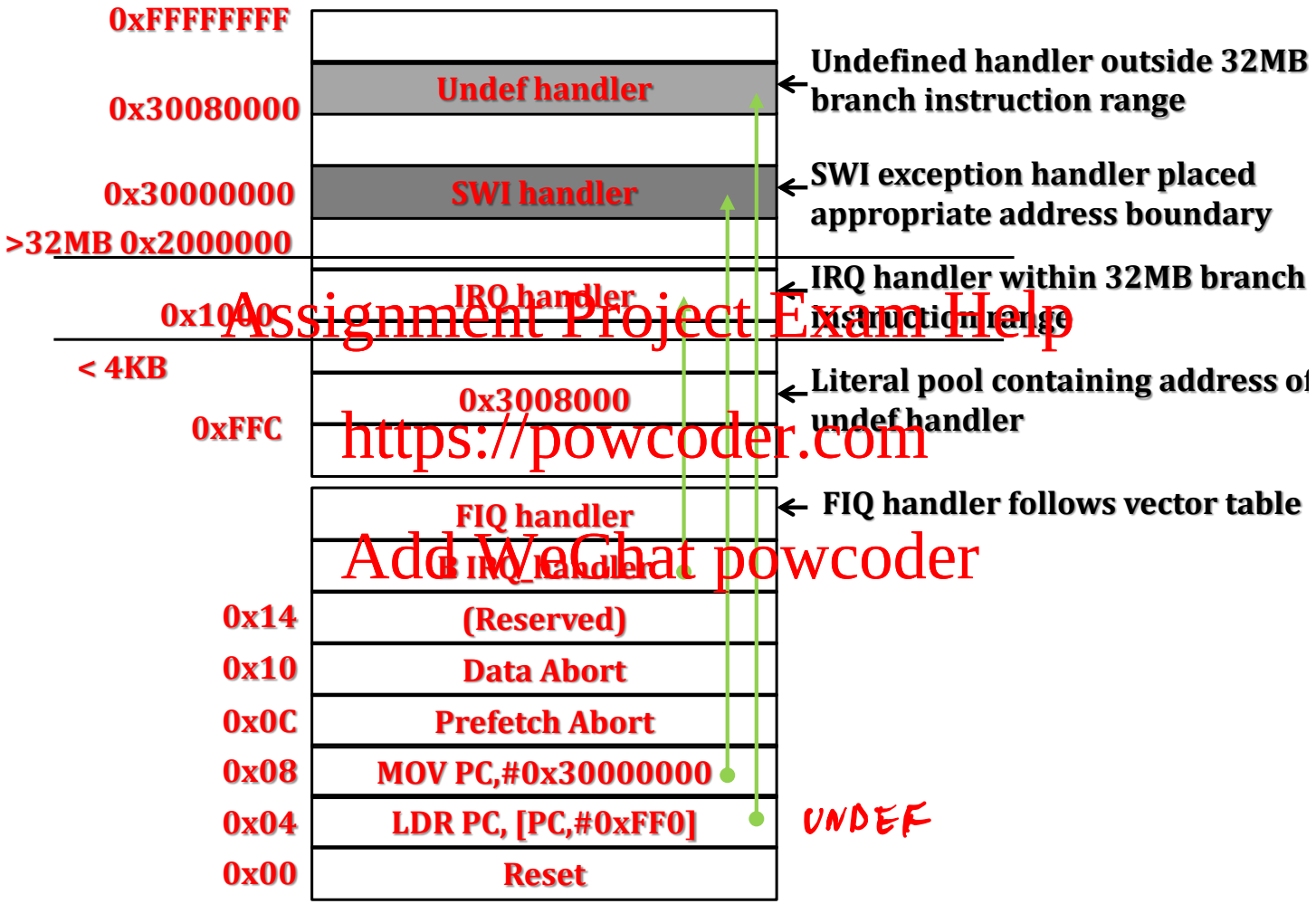


- FIQ (last entry) does not have the 32-bit space limit.
  - Servicing routines can be placed directly in the vector table.
  - Allows FIQ interrupts to be serviced quickly... no branching, no pipeline stalling.





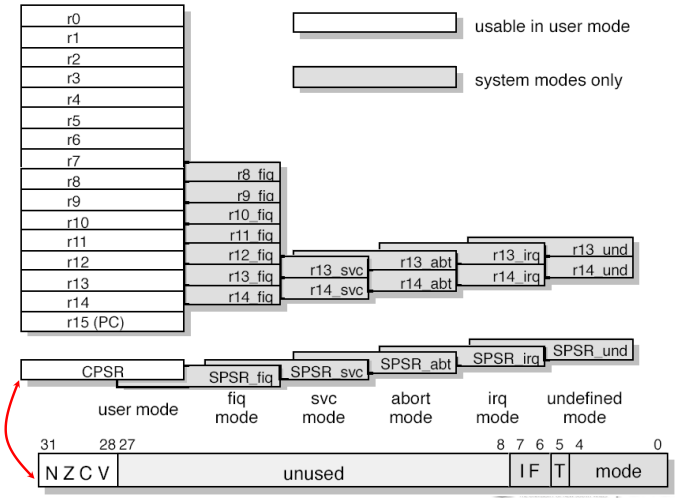
# The vector table



# Writing exception handlers

- SUMMARY: when exception (other than SYSTEM) occurs:
  1. Saves USER mode CPSR to SPSR_<mode>
  2. Modifies the appropriate bits of the CPSR
  3. Puts the return address in LR_<mode>
  4. Loads the PC with the appropriate handler address from the vector table
- All of above are done by the processor.
- Following mode change, the exception handler has access to its own **SP**, **LR** and **CPSR** (except SYSTEM mode).
- Your task is to write an appropriate exception handler:
  1. Deal with the exception
  2. Cleanly resume original execution path in User mode.





# Writing exception handlers

- The hardware swaps in banked registers (e.g. R13 / SP and R14 / LR).
- Other registers will need to be saved if you use them, like function calls:
  ```
  STMFD SP!, {R0-R12, LR}  ; save all non-banked registers and return addr
  ```
- Note the SP above is the privileged mode's stack pointer (why?), which can be different from the USER mode's SP.
- After the exception we resume the original instruction stream:
  ```
  LDMFD SP!, {R0-R12, PC}^ ; restore non-banked regs and ret addr
  ```
- The above is an **atomic operation** (done in a single step).
  - Imagine what happens if another exception is allowed to interrupt the processor while the above instruction takes place.



MOVS PC LR

# Exception priorities

- When multiple exceptions occur simultaneously, higher priority exceptions are handled first.

| Exception priorities | | |
|---|---|---|
| **Priority** | **Exception** | **comment** |
| Highest | Reset | Handler usually branches straight to main routine |
| | Data Abort | Can sometimes be helped with MMU |
| | FIQ | Current instruction completes, then the interrupt is acknowledged |
| | IRQ | Current instruction completes, then interrupt is acknowledged. |
| | Prefetch Abort | Can sometimes be helped with hardware (MMU) |
| | SWI | Execution of the instruction causes the exception |
| Lowest | Undefined instruction | SWI and Undef are actually exclusive, so they have the same priority |



- Q: Suppose A/D converter has asserted IRQ line and the processor tries to access a memory location that is undefined, while FIQ interrupt tells the processor of impending power failure. What's the servicing order?

**All 3 occurs simultaneously, but will serviced in order: Data Abort → FIQ → IRQ**

# Exception priorities

- While the processor is in an exception handler, a new exception of:
  - **higher priority** causes the processor to switch to the new exception.
    - Will save CPSR to SPSR, and regs to banked reg, of the higher-priority exception
  - **equal priority** is **blocked** until the current exception is serviced.
  - **lower priority** is **blocked** until the current exception is serviced.

# Final exam – key points

- Saturday August 20th 2 PM (Sydney time), via Moodle
- Time allowed: 1 hr 50 min (suggested: 10 min reading, 90 min working time, 15 min submission time)
- 3 Questions. Styled like the weekly exercises.
- Answers not submitted via Moodle will not get marked. Don't email answers to me.
- **Attempt this exam individually**.
- open-book exam.
- All **responses must be hand-written on paper, photographed, and uploaded in Moodle**.
- The **last page of your response for each question must be photographed with your UNSW student ID card visible in the image**.
- Responses not hand-written, or without student ID visible in last response page receive zero mark.
- Responses to calculation questions without working-out receive zero mark.

# Final exam – key points

- Will set up trial exam during Week-11 in Moodle (only 1 Q) for you to practice.
- Make sure your computer is ready (e.g. no unexpected software updates), and you have reliable internet.
- UNSW Multifactor Authentication (MFA) should be set up.
- Have your UNSW student card, pen and paper ready.
- Know how to operate your camera (e.g. your phone) and sending these to PC for Moodle upload.

# Final exam - key points

- UNSW takes academic misconduct seriously.
  - Randomized questions
  - Answer similarity checks
  - Moodle logging (IP address, internet transactions…)
- Don't be part of the statistic!
  - 2021 – 7 students (of 117) caught cheating and received zero
  - 2022 – 5 students (of 150) caught cheating and received zero
  - All these students were added to the UNSW Academic Misconduct Register



# This week

- Exceptions
- Interrupts and errors
- ARM processor modes
- Configuring the CPSR and SPSR
- Vector table & exception handler



**In Moodle:**

- **Start working on Lab**
  **(Due: end of your 3-hr lab)**
- **No weekly exercise this week ☺**