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

Week 4

Control flow & conditional operations

David Tsai

School of Electrical Engineering & Telecommunications

Graduate School of Biomedical Engineering

d.tsai@unsw.edu.au

Biomedical
Microsystems
Lab

This week

- Introduction: controlling execution flow
- Condition code flags & conditional execution
- Branch instructions
  - Selection structures: if … else …
  - Repetition structures
  - Jump tables

Introduction: controlling execution

- High-level programming constructs:
  1. Sequence
  2. Selection `if {…} then {…} else {…}`
  3. Repetition `while {…}`
  4. Function `int func(…) {…}`

- Mapping these into assembly code:
  1. Sequence: PC automatically incremented by 4 bytes on each instruction fetch.
  2. Selection: two possibilities in ARM:
     - **branch instructions**
     - **conditional execution** (of most instructions)
  3. Repetition: using **branch instructions**.
  4. Functions: achieved through **branch instructions**.

Conditional flags & conditional execution

- Condition flags in CPSR: ---- current program status register.
  - **N (negative)**
    N = 1 when the most significant bit (MSB) of the ALU output is ’1’. s_{31}
  - **Z (zero)**
    Z = 1 when the ALU output is zero. $\overline{s_{31}} \wedge \overline{s_{30}} \wedge \dots \wedge \overline{s_0}$ (*)
    and &&
  - **C (carry)**
    C = 1 when there is a carry out from the MSB. The C flag is also changed by the shifting operations. C_{out}
  - **V (overflow)**
    V = 1 when the result cannot be represented in 32 bits following a signed arithmetic operation. $C_{in} \oplus C_{out}$
    MSB in
    XOR
    MSB out

ADD

(1) $a_{31} \dots a_0$

(2) $b_{31} \dots b_0$

(3) $s_{31} \dots s_0$

Cin	Cout	V
0	0	0
0	1	1
1	0	1
1	1	0

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
N	Z	C	V	Do not modify / Read as Zero	I	F	T	M	M	M	M	M

Conditional flags & conditional execution

- Condition flags can be updated by:
  - ✓ The ‘S’ option of data processing instructions: ADD**S**, MOV**S**, …
  - ✓ Flag-setting instructions: CMP, CMN, TST, TEQ.
  - ✓ Shift operations (only update C flag). ROR , RRX . . . . .
  - ✓ Special instructions to edit the CPSR bits. MSR MRS

	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
✱	N	Z	C	V	Do not modify / Read as Zero	I	F	T	M	M	M	M	M

Flag setting instructions

result ↓ op1 op2

SUB r0 r1 r2

- **CMP (compare):** compares 1st and 2nd operands by performing a subtraction, while setting the flags.

```
  CMP R1, #10 *          ; (R1-10) and sets the flags
→ CMP R1, R0 *           ; (R1-R0) and sets the flags
  CMP R1, R0, LSL #3     ; (R1-R0×2^3) and sets the flags
```

SUB

- **CMN (compare negative):** Compares 1st operand with the negative of the 2nd operand, while setting the flags.

```
CMN R1, R0             ; (R1+R0) and sets the flags
CMN R1, R0, LSL #3     ; (R1+R0×2^3) and sets the flags
```

- Assembler will replace CMP with CMN when appropriate, e.g. `CMP R1, #-10` will be replaced by `CMN R1, #10`.

Flag setting instructions

- **TST (test bits):** logical AND between operands. Affects all but the V flag.

```
TST R1, #0x14        ; (R1 AND 0x14) and sets the flags
TST R1, R0           ; (R1 AND R0) and sets the flags
TST R1, R0, LSL#3    ; (R1 AND R0×2^3) and sets the flags
```

- **TEQ (test equivalence):** logical XOR between operands. Affects all but the V flag. TEQ can be used to check whether two values are the same.

```
TEQ R1, R0           ; (R1 XOR R0) and sets the flags
TEQ R1, R0, LSL #3   ; (R1 XOR R0×2^3) and sets the flags
```

Flag setting instructions

1 0 1 1⊗1 = 0
N Z C V

- Example 1. What are the condition flags after the last instruction?

```
                                C_in
                                1
LDR R0, =#0xFFFFFFFF      ;    1111 … 1111
LDR R1, =#0xFFFFFFFF      ;    1111 … 1111
ADDS R1, R1, R0           ;    ------------
                          ;  1 1111 … 1110
                           C_out
```

  - N=1 Z=0 C=1 V=0

Result characteristics:

MSB=1, non-zero, has carry-out,

Carry-in ⊗ carry-out = 0

- Example 2. What are the condition flags after the last instruction?

```
LDR R1, =0x7FFFFFFF      ; R1:            0111 … 1111
MVN R2, #0x0             ; R2:            1111 … 1111
SUBS R1, R1, R2, LSL #2  ; R2 LSL #2:     1111 … 1100
                         ;               1 C_in       1
                         ; R1:            0111 … 1111
                         ; R2 1'comp:     0000 … 0011
                         ;                -----------
                         ;              0 1000 … 0011
                                     C_out
```

  - N=1 Z=0 C=0 V=1

A-B ⇒ A+not(B)
+1

R1 - R2 = R1 + not(R2) + 1

Result characteristics:

MSB=1, non-zero, no carry-out,

Carry-in ⊗ carry-out = 1

1 ⊗ 0 = 1

Using flags for conditional execution

- ARM instructions can execute conditionally.
- The condition is
  - specified with a **two-letter suffix** (next slide), e.g. EQ, CC, …
  - tested against the CPSR flags.
- The instruction is **a no-op if the conditions are not met.**
  Often avoids the need for branch (B, BL), so no pipeline stalls (will see why) and increasing throughput. It also increase code density.

- Data processing instructions can set condition flags by suffixing **S**. The comparison instructions CMP and TST do this implicitly.

- Example: repeat while ≠ 1

```
          MOV  r1, #10
loop                          ; do
          SUBS r1, r1, #1     ;     i = i-1
          BNE  loop           ; while (i != 0)
```

Using flags for conditional execution

- Example: if-statement. The last two instructions are executed only if a == 5

```
CMP   r0, #5     ; if (a == 5)
MOVEQ r0, #10
BLEQ  fn         ;      fn(10)
```

- Example: if … else … statement.

```
CMP   r0, #0     ; if (x <= 0)
MOVLE r0, #0     ;      x = 0
MOVGT r0, #1     ; else
                 ;      x = 1
```

- Example: if (… || …) statement.

c != 'A'

```
CMP   r0, #'A'   ; if (c == 'A'
CMPNE r0, #'B'   ;   || c == 'B')
MOVEQ r1, #1     ;      y = 1
```

C evaluates logical operators left-to-right with short-circuiting (only check 2nd possibility if 1st is false).

Using flags for conditional execution

NZCV

Field Mnemonic	Conditional code	Flags status	Meaning
* EQ	0000	Z set	Equal
* NE	0001	Z clear	Not Equal
HS/CS	0010	C Set	Unsigned ≥
CC/LO	0011	C clear	Unsigned <
MI	0100	N set	Negative
PL	0101	N clear	Positive or zero
VS	0110	V set	Overflow
VC	0111	V clear	No overflow
HI	1000	C set and Z clear	Unsigned >
LS	1001	C clear and Z set	Unsigned ≤
GE	1010	N=V	Signed ≥
LT	1011	N≠V	Signed <
* GT	1100	Z clear, N = V	Signed >
* LE	1101	Z set, N ≠ V	Signed ≤
AL	1110	Always	Default

to zero

Using flags for conditional execution

- Can combine the S bit with conditional execution:

```
...
ADDEQS r0, r1, r2
```

B BL

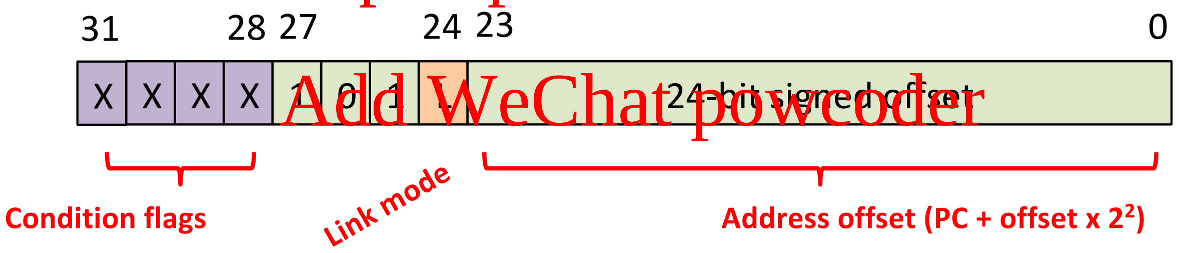
- Branching vs conditional execution: the CPU's branch penalty is at least 3 cycles.

- The ARMCC (not GCC) compiler make extensive use of condition codes in an optimization step called **branch removal**.

Branch instructions

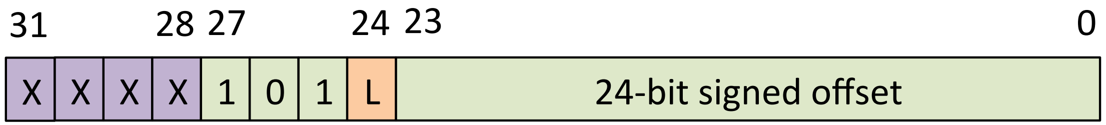
- Two branch instructions:
  - **B (branch)**: Regular branch. Often required to implement loops.
  - **BL (branch and link)**: Branch to a new location, then return to the original location. The link register (LR or R14) is used to hold the return address. Used to implement subroutines and function calls.

- The branch instruction bit fields:



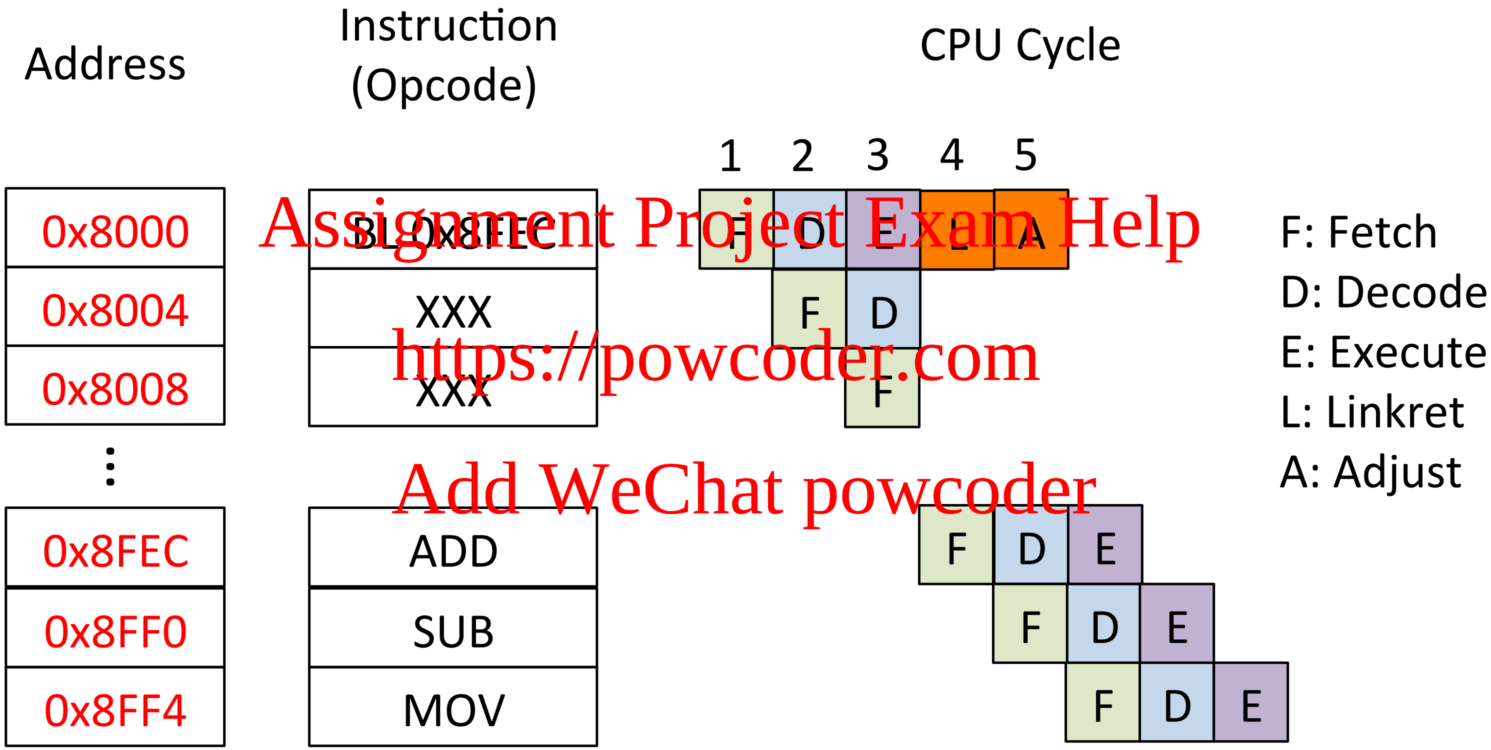
Branch instructions

- ARM address bus is 32-bit wide. How to specify 32-bit branch target address with 32-bit instruction? **PC-relative addressing**.
- Branch address = (current PC) + (offset x 2^2).
- The 24-bit signed offset specifies the **word offset** from the current PC. The offset is multiplied by 4 during target address computation.
- 24-bits signed offset ⇒ ± 32MB limit for branch target.
  - Problem: **cannot branch more than 32 MB away from PC.**



31			28	27			24	23	0
X	X	X	X	1	0	1	L	24-bit signed offset	

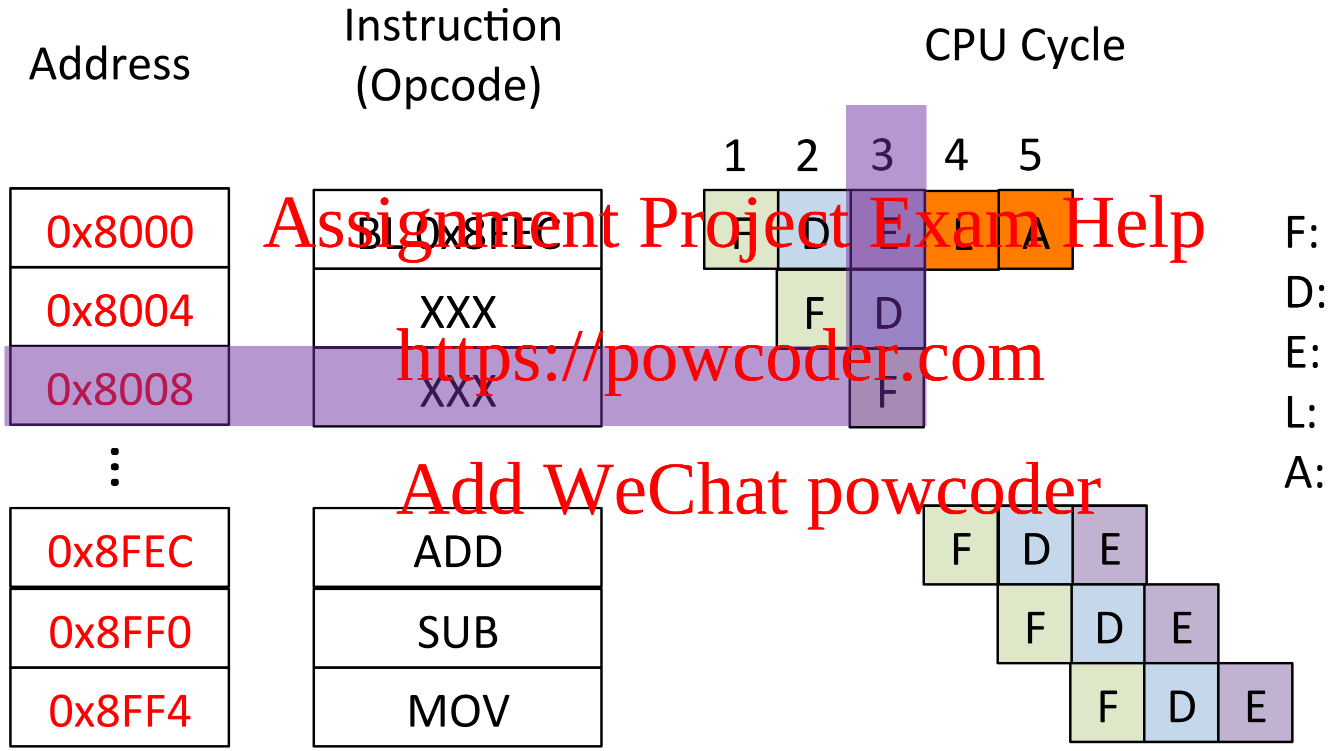
Branch instructions: 3-stage pipeline



Address	Instruction (Opcode)
0x8000	BL 0x8FEC
0x8004	XXX
0x8008	XXX
⋮	
0x8FEC	ADD
0x8FF0	SUB
0x8FF4	MOV

F: Fetch
D: Decode
E: Execute
L: Linkret
A: Adjust

Branch instructions: 3-stage pipeline

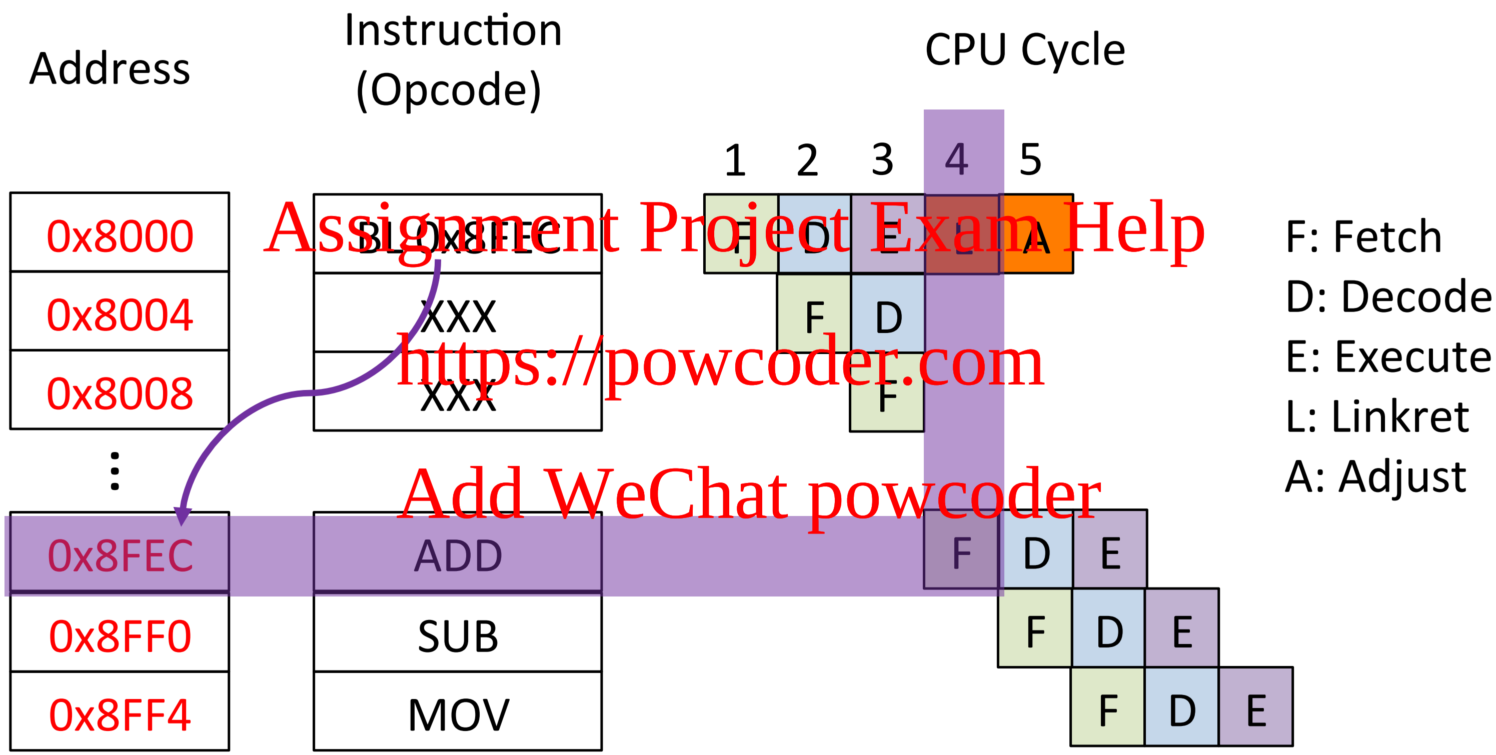


Address	Instruction (Opcode)	1	2	3	4	5			
0x8000	BL 0x8FEC	F	D	E	L	A			
0x8004	XXX		F	D					
0x8008	XXX			F					
⋮									
0x8FEC	ADD				F	D	E		
0x8FF0	SUB					F	D	E	
0x8FF4	MOV						F	D	E

CPU Cycle

F: Fetch
D: Decode
E: Execute
L: Linkret
A: Adjust

Branch instructions: 3-stage pipeline



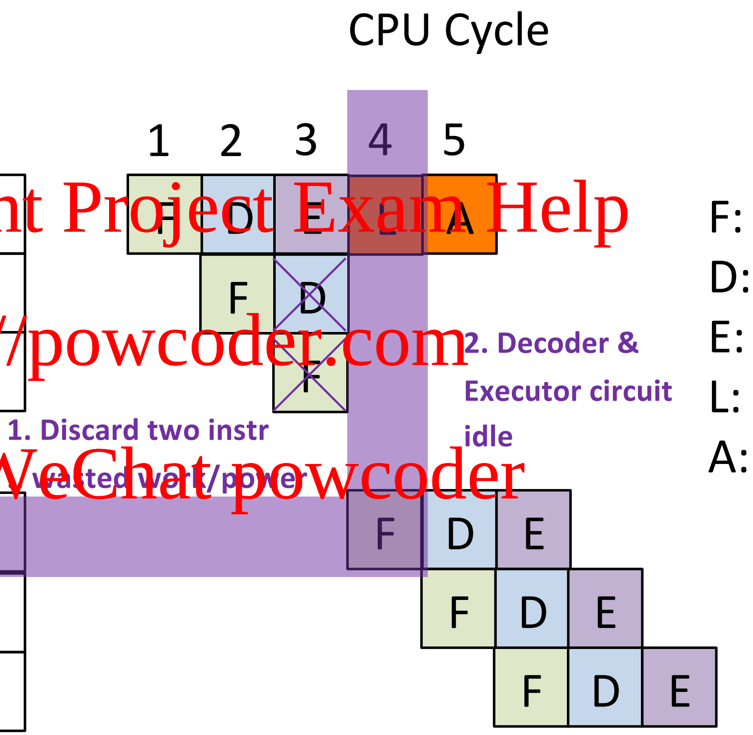
Address	Instruction (Opcode)	1	2	3	4	5			
0x8000	BL 0x8FEC	F	D	E	L	A			
0x8004	XXX		F	D					
0x8008	XXX			F					
⋮									
0x8FEC	ADD				F	D	E		
0x8FF0	SUB					F	D	E	
0x8FF4	MOV						F	D	E

CPU Cycle

F: Fetch
D: Decode
E: Execute
L: Linkret
A: Adjust

Branch instructions: 3-stage pipeline

Address	Instruction (Opcode)
0x8000	BL 0x8FEC
0x8004	XXX
0x8008	XXX
⋮	
0x8FEC	ADD
0x8FF0	SUB
0x8FF4	MOV

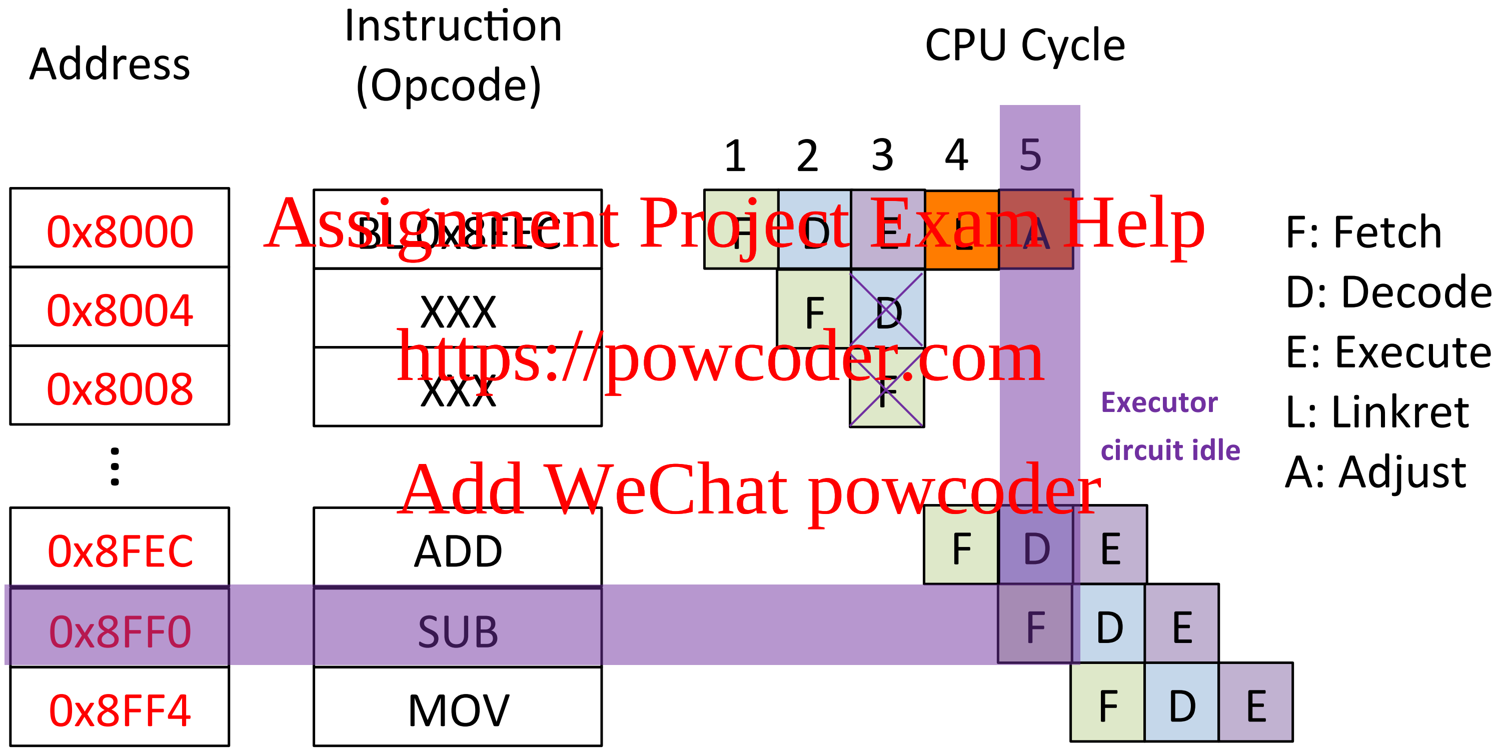


1. Discard two instr wasted work/power

2. Decoder & Executor circuit idle

F: Fetch
D: Decode
E: Execute
L: Linkret
A: Adjust

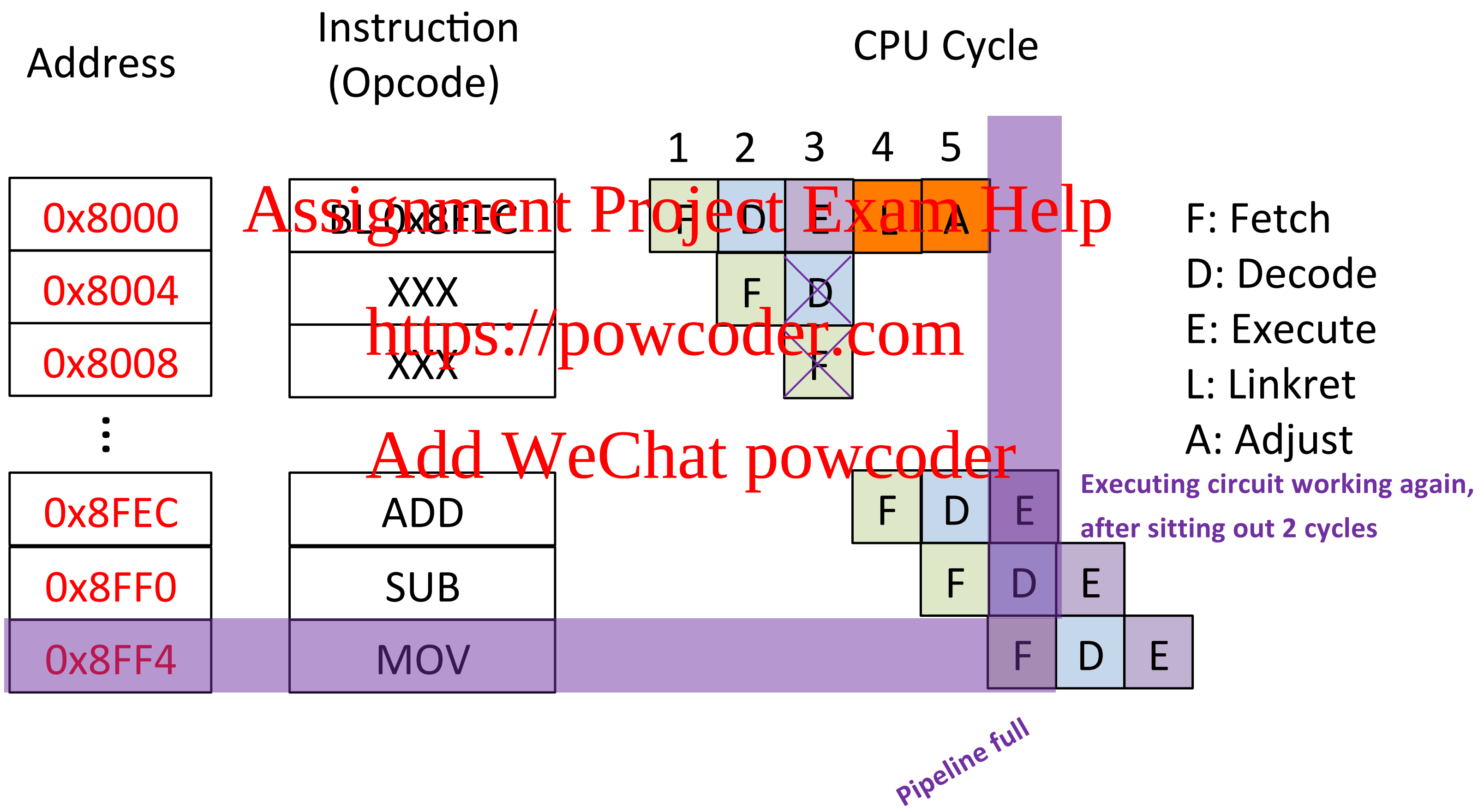
Branch instructions: 3-stage pipeline



Address	Instruction (Opcode)
0x8000	BL 0x8FEC
0x8004	XXX
0x8008	XXX
⋮	
0x8FEC	ADD
0x8FF0	SUB
0x8FF4	MOV

F: Fetch
D: Decode
E: Execute
L: Linkret
A: Adjust

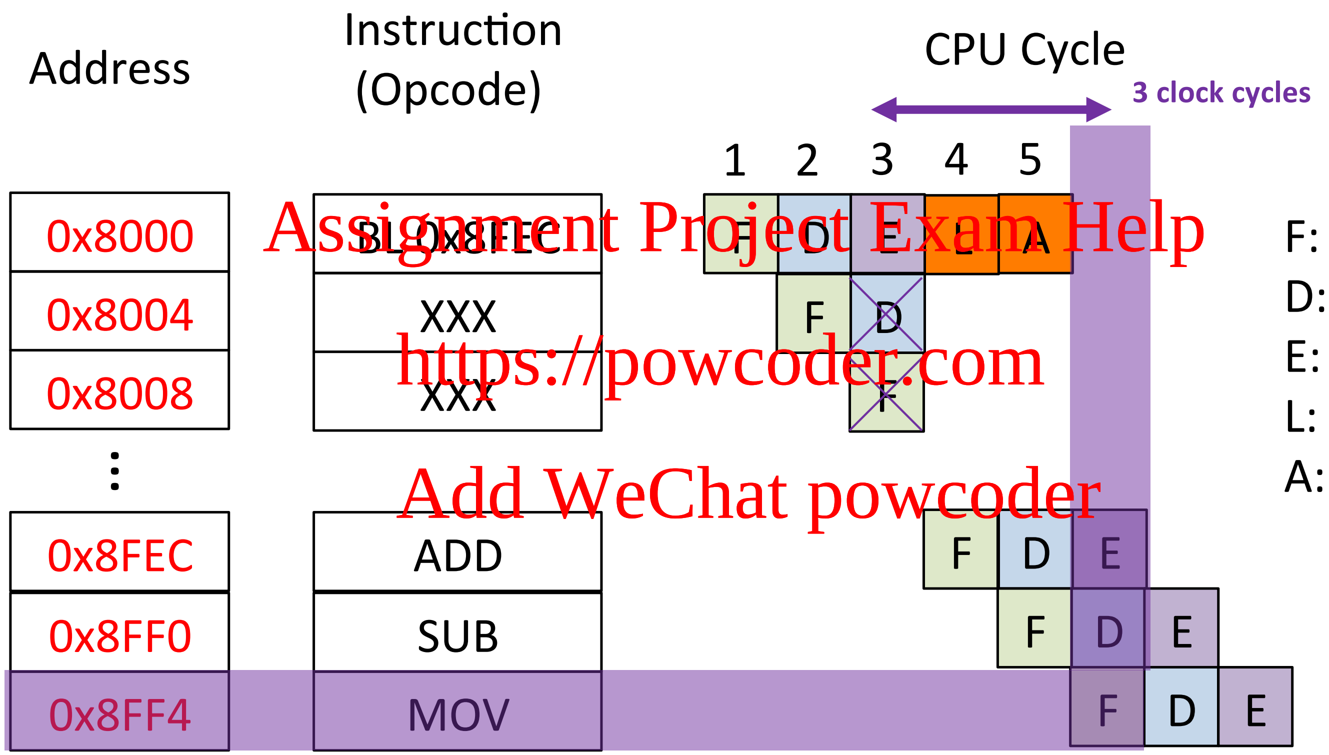
Branch instructions: 3-stage pipeline



Address	Instruction (Opcode)
0x8000	BL 0x8FEC
0x8004	XXX
0x8008	XXX
⋮	
0x8FEC	ADD
0x8FF0	SUB
0x8FF4	MOV

F: Fetch
D: Decode
E: Execute
L: Linkret
A: Adjust

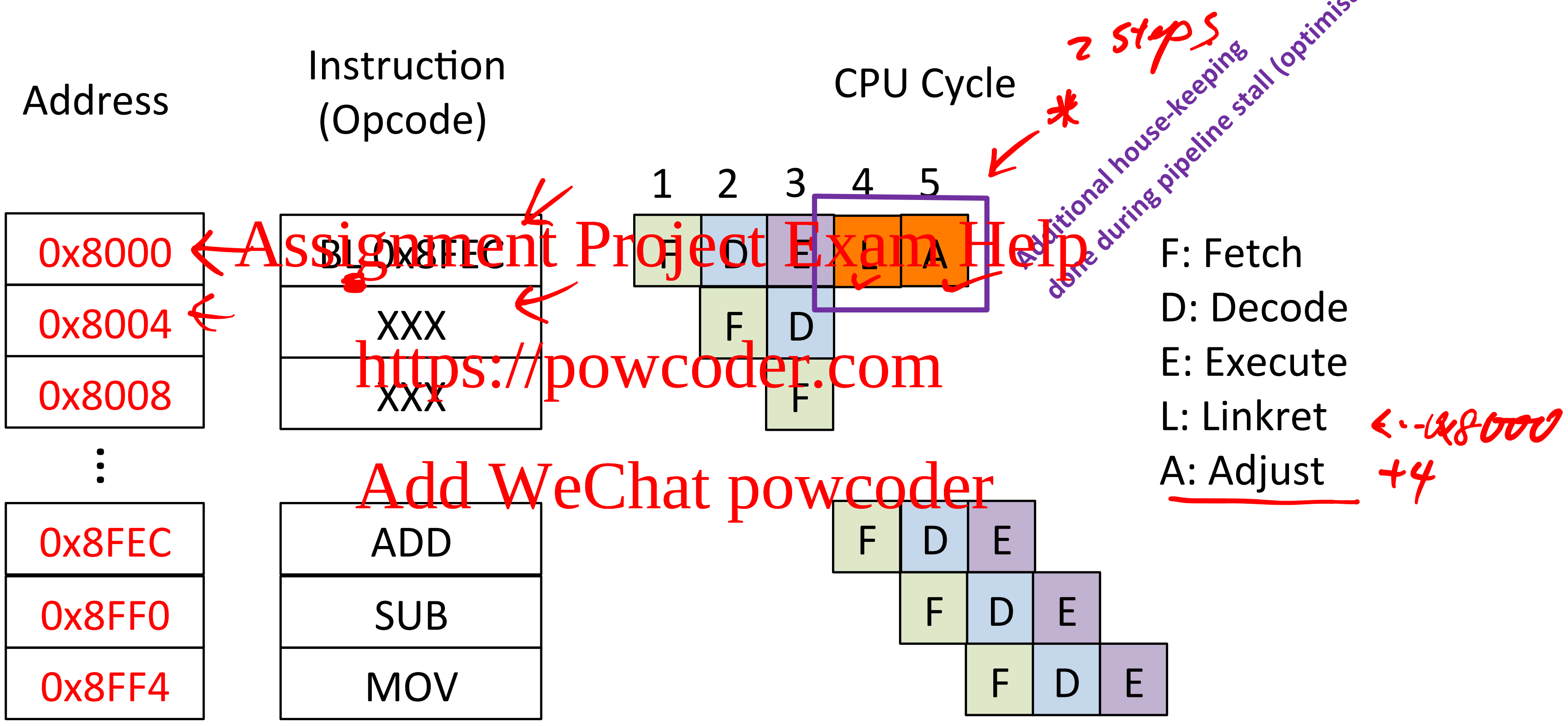
Branch instructions: 3-stage pipeline



Address	Instruction (Opcode)	1	2	3	4	5			
0x8000	BL 0x8FEC	F	D	E	L	A			
0x8004	XXX		F	~~D~~					
0x8008	XXX			~~F~~					
⋮									
0x8FEC	ADD				F	D	E		
0x8FF0	SUB					F	D	E	
0x8FF4	MOV						F	D	E

F: Fetch
D: Decode
E: Execute
L: Linkret
A: Adjust

Branch instructions: 3-stage pipeline



Address	Instruction (Opcode)	1	2	3	4	5			
0x8000	BL 0x8FEC	F	D	E	L	A			
0x8004	XXX		F	D					
0x8008	XXX			F					
⋮									
0x8FEC	ADD				F	D	E		
0x8FF0	SUB					F	D	E	
0x8FF4	MOV						F	D	E

CPU Cycle

F: Fetch
D: Decode
E: Execute
L: Linkret
A: Adjust

Additional house-keeping done during pipeline stall (optimisation)

Branch instruction: summary

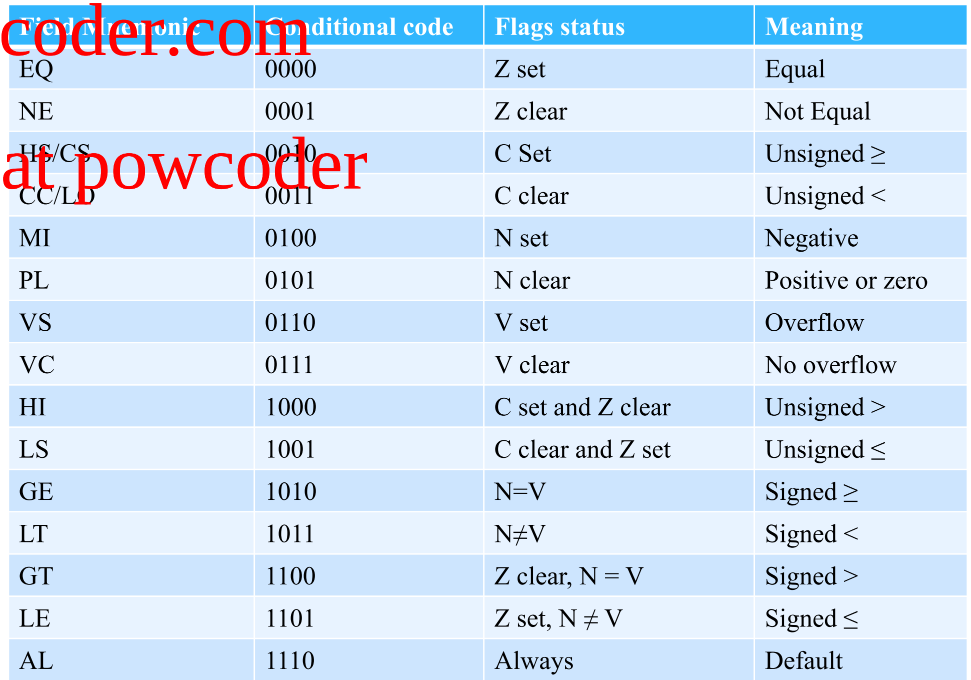
- We have seen:
  - In cycle 3, the BL instruction is at the execution stage while instructions at 0x8004 and 0x8008 are being decoded and fetched, respectively. If the branch is taken, these two instructions should not proceed to execute and decode during cycle 4.
  - So whenever a branch is taken, the instructions already fetched and decoded are thrown away.
  - Once BL is executed, CPU fetches the next instruction from the branch target address 0x8FEC.
- Additional house-keeping for linking:
  - In cycle 4, current PC value 0x8008 is stored in the LR (R14) However, this is not the proper return address.
  - In cycle 5, LR is adjusted as LR - 4, thereby pointing to the correct return address 0x8004.

Conditional branching

- B <label>: unconditional branch. Always taken.
- Conditional branch:
  - **BEQ** (branch of equal), **BNE** (branch if not equal)
  - Signed comparisons: **BLT**, **BLE**, **BGT**, **BGE**
  - Unsigned comparisons: **BLO**, **BLS**, **BHI**, **BHS**

- Example: if V1 = 0xFFFFFFFA and V2= 0x0000FFFA

```
CMP V1, V2
BGT label1   ; branch is not taken
BHI label2   ; branch is taken
```



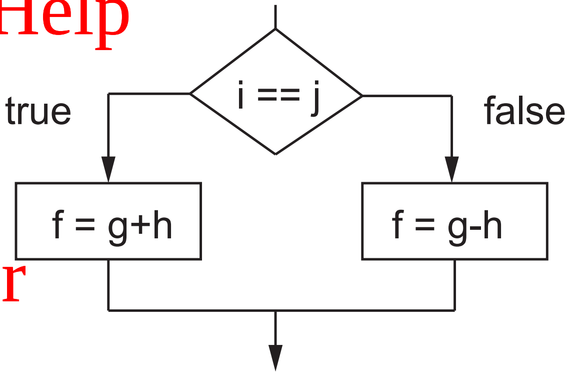
Field Mnemonic	Conditional code	Flags status	Meaning
EQ	0000	Z set	Equal
NE	0001	Z clear	Not Equal
HS/CS	0010	C Set	Unsigned ≥
CC/LO	0011	C clear	Unsigned <
MI	0100	N set	Negative
PL	0101	N clear	Positive or zero
VS	0110	V set	Overflow
VC	0111	V clear	No overflow
HI	1000	C set and Z clear	Unsigned >
LS	1001	C clear and Z set	Unsigned ≤
GE	1010	N=V	Signed ≥
LT	1011	N≠V	Signed <
GT	1100	Z clear, N = V	Signed >
LE	1101	Z set, N ≠ V	Signed ≤
AL	1110	Always	Default

Selection structures

- Example: with the following integer variables and register assignments:
  f →V1, g→V2, h→V3, i→V4, j→V5
  Consider the following example.

  C code:

```
if (i == j)
    f = g + h;
else
    f = g - h;
```



  ARM assembly:

```
    CMP V4, V5
    BEQ true
    SUB V1, V2, V3
    B   exit
true
    ADD V1, V2, V3
exit
```

	true	false
	1	1
	3	1
		1
		3
	1	
TOTAL	5	6

- Q: how many clock cycles does this take?

i == j: 1 + 3 + 1 = 5

i != j: 1 + 1 + 1 + 3 = 6

Selection structures

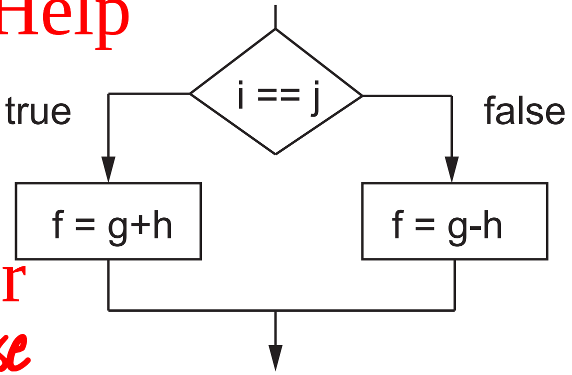
- Example: with the following integer variables and register assignments:
  f →V1, g→V2, h→V3, i→V4, j→V5
  Consider the following example.

  C code:

```
if (i == j)
    f = g + h;
else
    f = g - h;
```

  ARM assembly:

```
    CMP V4, V5
    BEQ true
    SUB V1, V2, V3
    B   exit
true
    ADD V1, V2, V3
exit
```



- Q: can you make it more efficient?

True | False

⇒ 3

```
CMP   V4, V5        ; if (i==j)
ADDEQ V1, V2, V3    ;     f = g+h
SUBNE V1, V2, V3    ; else f = g-h
```

Repetition structures

- while loops:

```
while (cond==true) {
    ...
    ...
    ...
}
```

```
        B eval
loop    ...
        ...
        ...
eval    xxxx ; evaluate cond
        BEQ loop
```

```
loop    xxxx   ; evaluate cond
        BNE exit
        ...
        ...
        ...
        B loop
exit
```

Repetition structures

- do … while loops:

```
do {
    ...
    ...
    ...
} while (cond==true)
```

```
loop    ...
        ...
        ...
        xxxx  ; eval cond
        BEQ loop
```

- for loops:

```
for (j=0; j<10; j++) {
    ...
    ...
    ...
}
```

```
        MOV R1, #0
loop    CMP R1, #10
        BGE exit
        ...
        ...
        ...
        ADD R1, R1, #1
        B loop
exit
```

```
        MOV R1, #10
loop    ...
        ...
        ...
        SUBS R1, R1, #1
        BNE loop
```

Shorter, Starting from 10 and counts down to 0

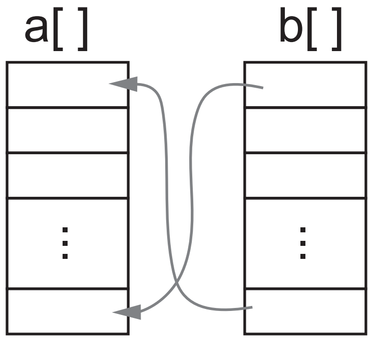
Repetition structures

- Example: translate the following c program to assembly.

```
for (i = 0; i < 8; i++) {
    a[i] = b[7-i];
}
```

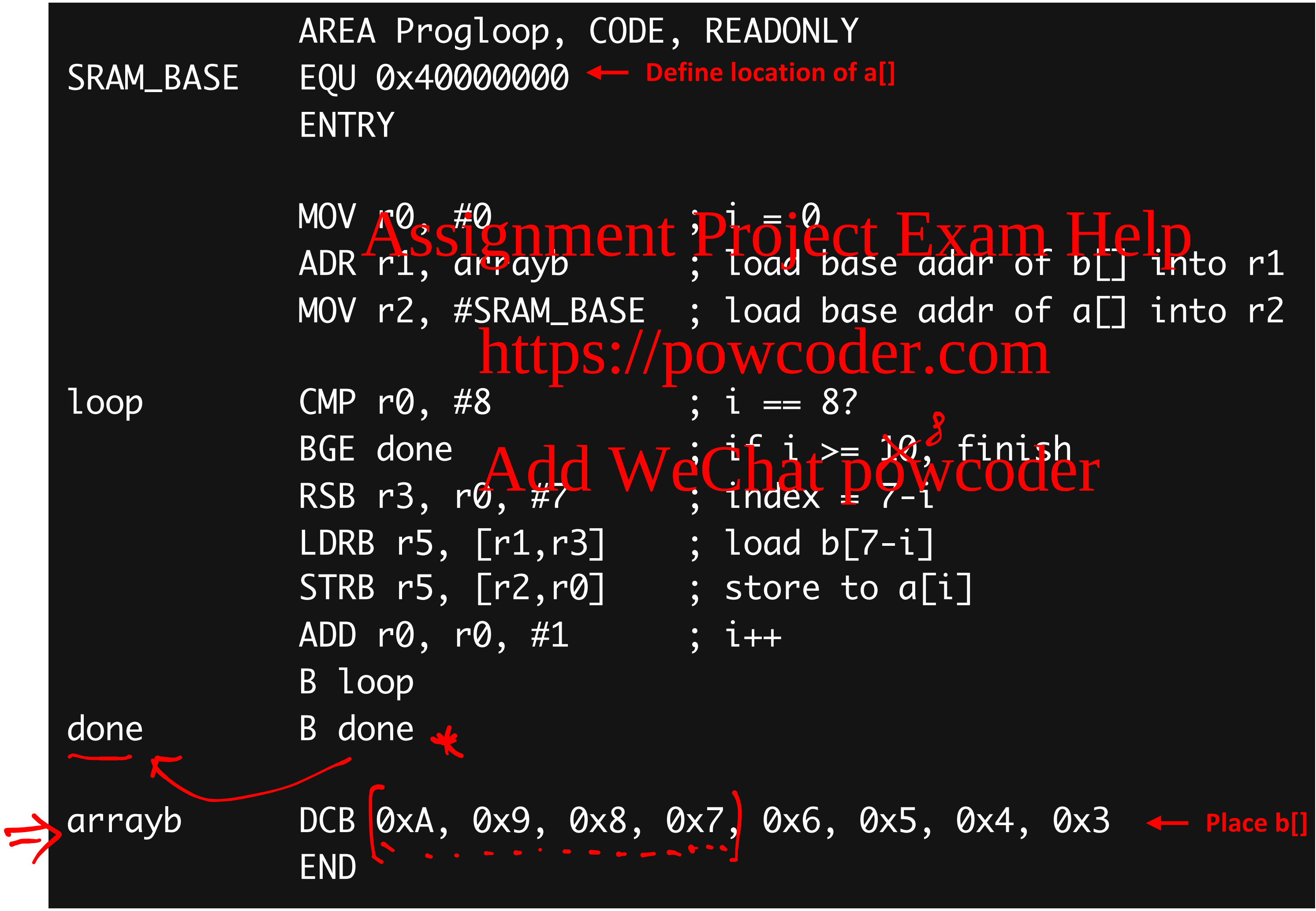
  - Arrays a[] and b[] contain bytes. Array a[] has base address 0x40000000, located in the SRAM. Array b[] is place within the program memory using DCB directives.
  - Two considerations: the algorithms, memory arrangement.

- Algorithm: swapping elements between a[] and b[].



Repetition structures

- Index upwards



```
                AREA Progloop, CODE, READONLY
SRAM_BASE       EQU 0x40000000
                ENTRY

                MOV r0, #0              ; i = 0
                ADR r1, arrayb          ; load base addr of b[] into r1
                MOV r2, #SRAM_BASE      ; load base addr of a[] into r2

loop            CMP r0, #8              ; i == 8?
                BGE done                ; if i >= ~~10~~ 8, finish
                RSB r3, r0, #7          ; index = 7-i
                LDRB r5, [r1,r3]        ; load b[7-i]
                STRB r5, [r2,r0]        ; store to a[i]
                ADD r0, r0, #1          ; i++
                B loop
done            B done

arrayb          DCB 0xA, 0x9, 0x8, 0x7, 0x6, 0x5, 0x4, 0x3
                END
```

Define location of a[]

Place b[] at end of program space

Repetition structures

- Index upwards

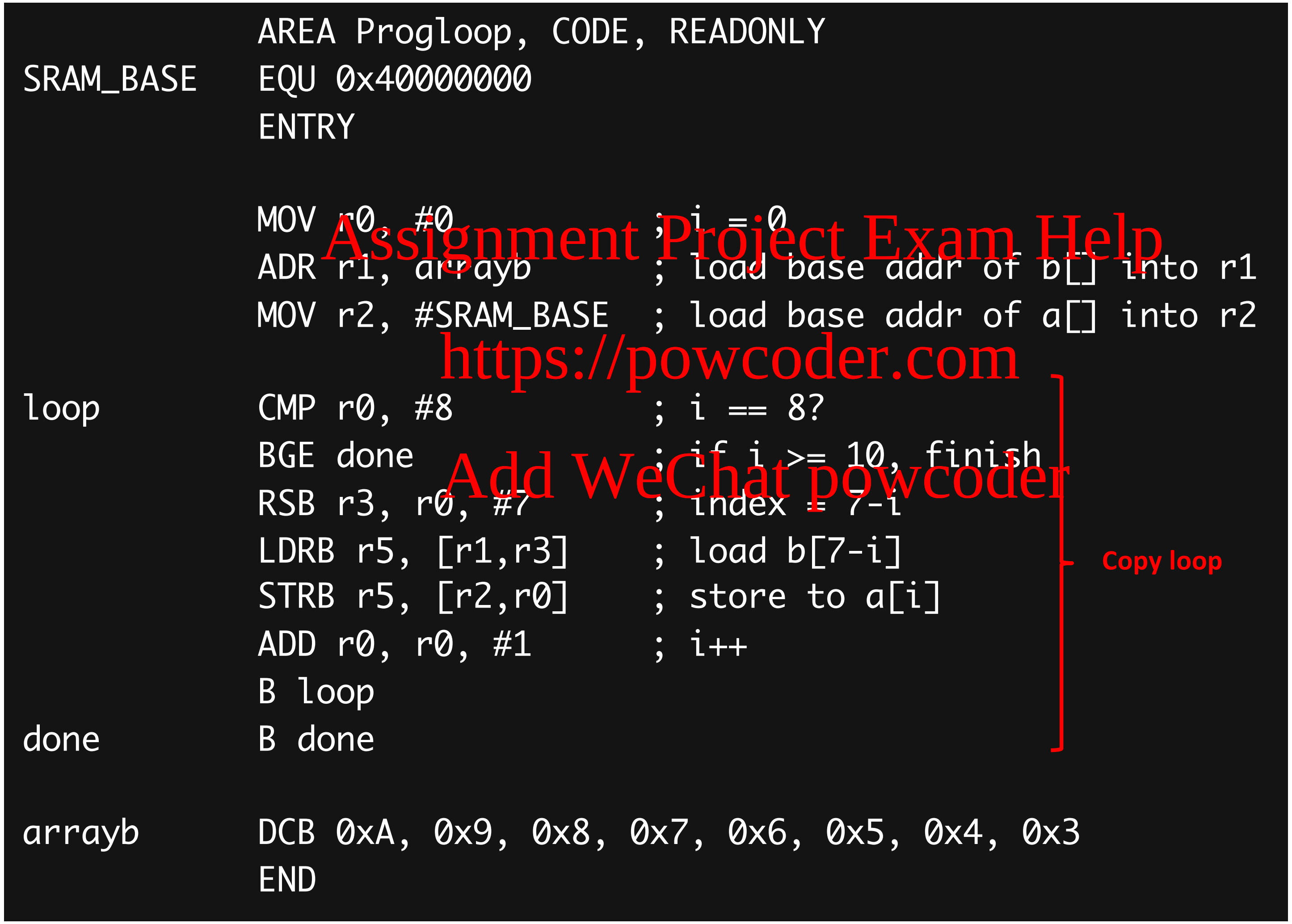
```
            AREA Progloop, CODE, READONLY
SRAM_BASE   EQU 0x40000000
            ENTRY

            MOV r0, #0          ; i = 0
            ADR r1, arrayb      ; load base addr of b[] into r1
            MOV r2, #SRAM_BASE  ; load base addr of a[] into r2

loop        CMP r0, #8          ; i == 8?
            BGE done            ; if i >= 10, finish
            RSB r3, r0, #7      ; index = 7-i
            LDRB r5, [r1,r3]    ; load b[7-i]
            STRB r5, [r2,r0]    ; store to a[i]
            ADD r0, r0, #1      ; i++
            B loop
done        B done

arrayb      DCB 0xA, 0x9, 0x8, 0x7, 0x6, 0x5, 0x4, 0x3
            END
```

Pseudo instr. to copy mem addr to reg

Set up mem. addresses

Repetition structures

- Index upwards



```
            AREA Progloop, CODE, READONLY
SRAM_BASE   EQU 0x40000000
            ENTRY

            MOV r0, #0          ; i = 0
            ADR r1, arrayb      ; load base addr of b[] into r1
            MOV r2, #SRAM_BASE  ; load base addr of a[] into r2

loop        CMP r0, #8          ; i == 8?
            BGE done            ; if i >= 10, finish
            RSB r3, r0, #7      ; index = 7-i
            LDRB r5, [r1,r3]    ; load b[7-i]
            STRB r5, [r2,r0]    ; store to a[i]
            ADD r0, r0, #1      ; i++
            B loop
done        B done

arrayb      DCB 0xA, 0x9, 0x8, 0x7, 0x6, 0x5, 0x4, 0x3
            END
```

Copy loop

Repetition structures

- Example: translate the following c program to assembly.

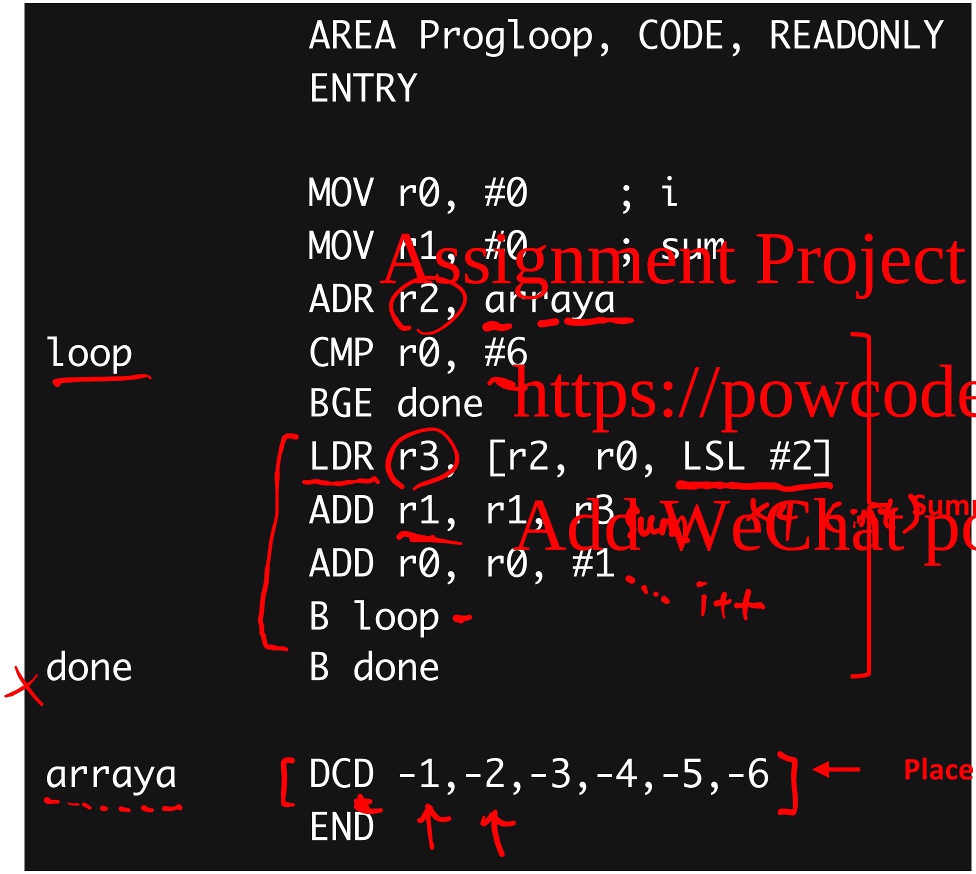
```
sum = 0;
for (i = 0; i < 6; i++) {
    sum += a[i];
}
```

DCD (32 bit)

- The array a[] should be declared within the code space (program memory) using appropriate directives. All variables are integers. 32-bit

Repetition structures

- Indexing upwards



```
              AREA Progloop, CODE, READONLY
              ENTRY

              MOV r0, #0    ; i
              MOV r1, #0    ; sum
              ADR r2, arraya
loop          CMP r0, #6
              BGE done
              LDR r3, [r2, r0, LSL #2]
              ADD r1, r1, r3
              ADD r0, r0, #1
              B loop
done          B done

arraya        DCD -1,-2,-3,-4,-5,-6
              END
```

sum

i++

Summing loop

Place a[] at end of program space, note ’D’ for 32-bit width

Repetition structures

- Indexing upwards

```
            AREA Progloop, CODE, READONLY
            ENTRY

            MOV r0, #0    ; i
            MOV r1, #0    ; sum
            ADR r2, arraya
loop        CMP r0, #6
            BGE done
            LDR r3, [r2, r0, LSL #2]
            ADD r1, r1, r3
            ADD r0, r0, #1
            B loop
done        B done

arraya      DCD -1,-2,-3,-4,-5,-6
            END
```

4 x offset (32-bit wide)

Repetition structures

- Indexing upwards

```
                AREA Progloop, CODE, READONLY
                ENTRY

                MOV r0, #0     1 ; i
                MOV r1, #0     1 ; sum
                ADR r2, arraya     1
loop    ┌  1    CMP r0, #6     1 ┐
        │  1    BGE done       3 ┘ 7th - exit
        │  3    LDR r3, [r2, r0, LSL #2]
6 reps ─┤  1    ADD r1, r1, r3
        │  1    ADD r0, r0, #1
        └  3    B loop
done            B done

arraya          DCD -1,-2,-3,-4,-5,-6
                END
```

Most instr.: 1 cycle

B/BL: 3 cycles

Memory access: 3 cycles

- Q: how many CPU cycles does this summation code use?

Cycles = 3 + (10 * 6 repetitions) + 4 = 67

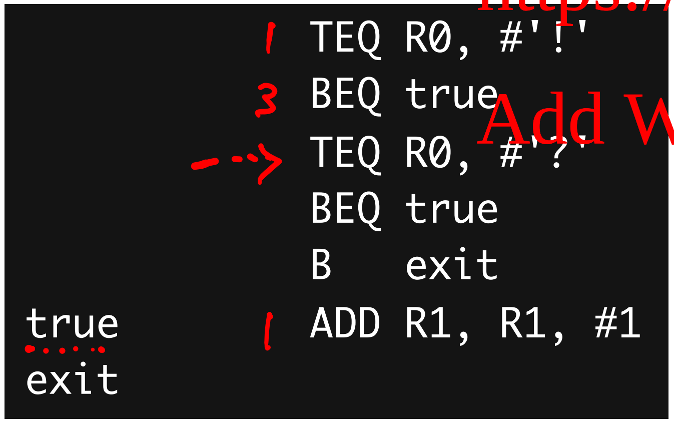
More branch removal

- Example: translate the following c program to assembly.

```
if (var == ‘!’ || var == ‘?’)
    found++;
```

- The variables *var* and *found* correspond to registers R0 and R1, respectively.

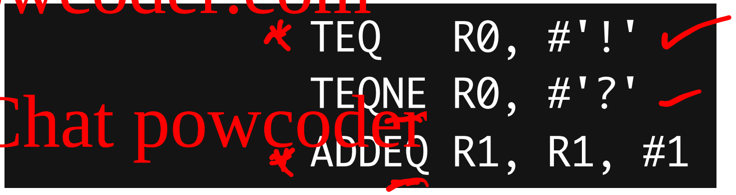
Standard implementation:



```
              1  TEQ R0, #'!'
              3  BEQ true
           -->   TEQ R0, #'?'
                 BEQ true
                 B   exit
true          1  ADD R1, R1, #1
exit
```

≥ 5 cycles

Optimized implementation:



```
          *  TEQ   R0, #'!'  ✓
             TEQNE R0, #'?'  ✓
          *  ADDEQ R1, R1, #1
```

3 cycles

Case selection structure

- Switch case statements:

```
switch (k) {
    case 0:
        f=i+j; break;
    case 1:
        f=g+h; break;
    case 2:
        f=g-h; break;
    case 3:
        f=i-j; break;
}
```

- Nested if … else ladder:

```
if (k==0)
    f=i+j;
elseif (k==1)
    f=g+h;
elseif (k==2)
    f=g-h;
elseif (k==3)
    f=i-j;
```

Assuming f →V1, i→V2, j→V3, g→V4, h→V5 , k→V6

```
            CMP v6, #0      ; k==0?
            BNE L1
            ADD v1, v2, v3
            B exit
L1          CMP v6, #1      ; k==1?
            BNE L2
            ADD v1, v4, v5
            B exit
L2          CMP v6, #2      ; k==2?
            BNE L3
            SUB v1, v4, v5
            B exit
L3          CMP v6, #3      ; k==3?
            BNE exit
            SUB v1, v2, v3
exit
```

If no match: 4 branches… very inefficient

Efficient case selection structure: jump table

- General strategy: avoid branch instruction, index to target code by changing the PC.
- Implementation:

1. Check whether the controlling variable is **within range** (0 ≤ V6 ≤ 3):

   ```
   CMP V6, #0
   BLT exit
   CMP V6, #3
   BGT exit
   ```

2. Implement the code for each case as **separate code block with a label**.
3. Put these labels into a table using the **DCD** directive and load the starting address of this table into a base register using the **ADR** instruction.
4. Change the PC using LDR with this base register and the controlling variable as the offset.

Efficient case selection structure: jump table

```
                .
                .
                CMP V6, #3
                BHI exit
                ADR R0, JUMP_TABLE
                LDR PC, [R0, V6, LSL #2]
JUMP_TABLE      DCD L0, L1, L2, L3
L0              .
                .
                B exit
L1              .
                .
                B exit
L2              .
                .
                B exit
L3              .
                .
                B exit
exit            B exit
```

Ensuring 0 ≤ V6 ≤ 3

Branch to case code directly

Table of case labels (label = code address)

Case 0 code

Case 1 code

Case 2 code

Case 3 code

(base addr + k4)

base addr

k×4

0≤k≤3

reg. V6

k

k==3

0

1

2

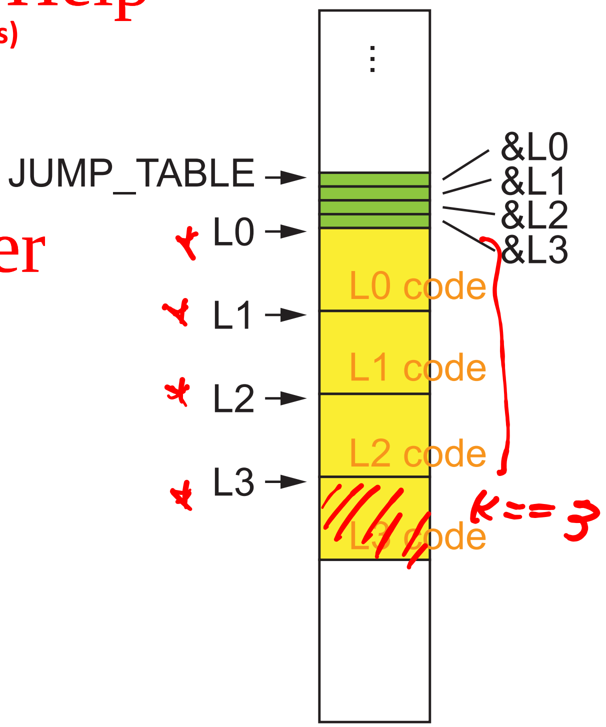
3

* PC = PC+4

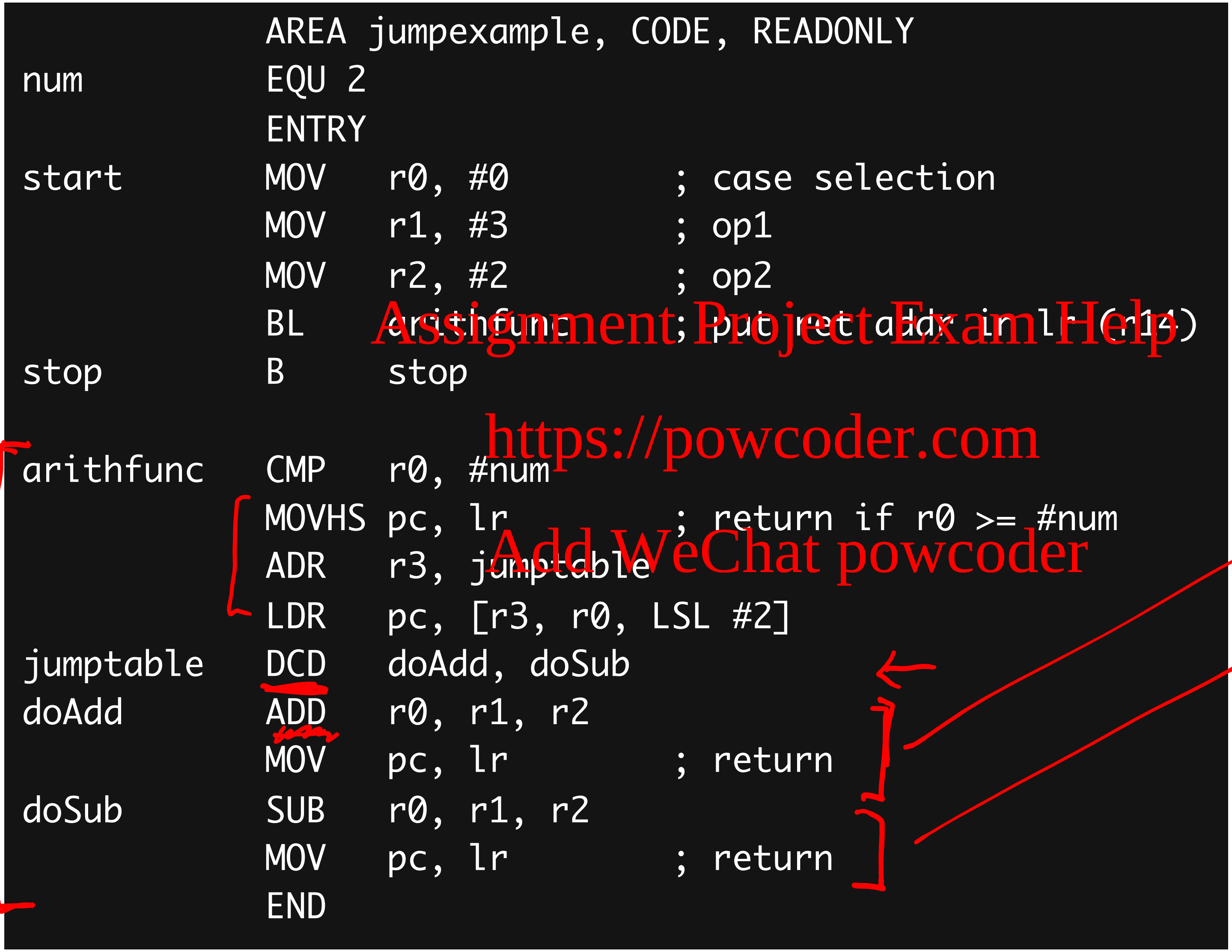
... HW

≠

Program mem. space



Efficient case selection structure: jump table



```
            AREA jumpexample, CODE, READONLY
num         EQU 2
            ENTRY
start       MOV    r0, #0        ; case selection
            MOV    r1, #3        ; op1
            MOV    r2, #2        ; op2
            BL     arithfunc     ; put ret addr in lr (r14)
stop        B      stop

arithfunc   CMP    r0, #num
            MOVHS  pc, lr        ; return if r0 >= #num
            ADR    r3, jumptable
            LDR    pc, [r3, r0, LSL #2]
jumptable   DCD    doAdd, doSub
doAdd       ADD    r0, r1, r2
            MOV    pc, lr        ; return
doSub       SUB    r0, r1, r2
            MOV    pc, lr        ; return
            END
```

```
switch (r0)
{
    case 0:
        return r1+r2;
    case 1:
        return r1-r2;
}
```

This week

- Introduction: controlling execution flow
- Condition code flags & conditional execution
- Branch instructions
  - Selection structures: if … else …
  - Repetition structures
  - Jump tables

In Moodle:

- **Start working on Lab (Due: end of your 3-hr lab)**
- **Start doing Week 4 exercise**

References

[1] William Hohl, ARM Assembly Language: Fundamentals and Techniques, CRC Press, 2015 (2nd Edition).

[2] ARM Architecture Reference Manual.