CSCI 6461 Computer Systems Architecture

Lecture 5

Branch Instructions

Introduction

UNIVERSITET ◦ UNIVERSITY
ADA

Introduction

- ARM32 uses branching for control flow in conditional statements (`if/else`) and loops.
- Branching changes the program counter (PC) to jump to a label or address.
- Branching types:
  - Unconditional (always branch)
  - Conditional (based on flags in CPSR register).
- Flags: N (Negative), Z (Zero), C (Carry), V (Overflow) determine conditions.

Pipeline Flushes

- **Branching** can introduce pipeline flushes
  - the pipeline discards partially processed instructions, typically due to a control flow change
- Conditional branches (`b<cond>` execution in ARM state) depend on condition codes (flags in the CPSR register).
- The condition is evaluated in the Execute stage,
  - but the Fetch and Decode stages may already be processing subsequent instructions.
- If the branch is taken, these instructions are invalid, leading to a pipeline **flush** and **delay**.

Unconditional Branching

`b` : Simple branch to a label (PC-relative offset, ±32MB range).

`bl` : Branch with Link, saves return address in `lr` (for subroutines).

`bx` `<rm>`: Branch to address in register `rm` (useful for computed jumps).

Conditional Branching

- **`b<cond>`** , where <cond> is a suffix like `eq`, `ne`, `gt`, `lt`.
  - examples: **`beq`** (branch if equal/z=1), **`bne`** (not equal/z=0), **`bgt`** (greater than).
- based on ALU operations setting flags (e.g., **`cmp r0, r1`** sets flags for comparison).
- *limited range: ±32mb, **`pc`**-relative*.
- for longer jumps, use conditional execution or load address into a register **`rx`** and **`bx rx`**.

Branching & conditions

```
if, then, else
```

`if condition as cmp/bl`

In assembly, the simple if statement is implemented as a **pair** of assembly statements

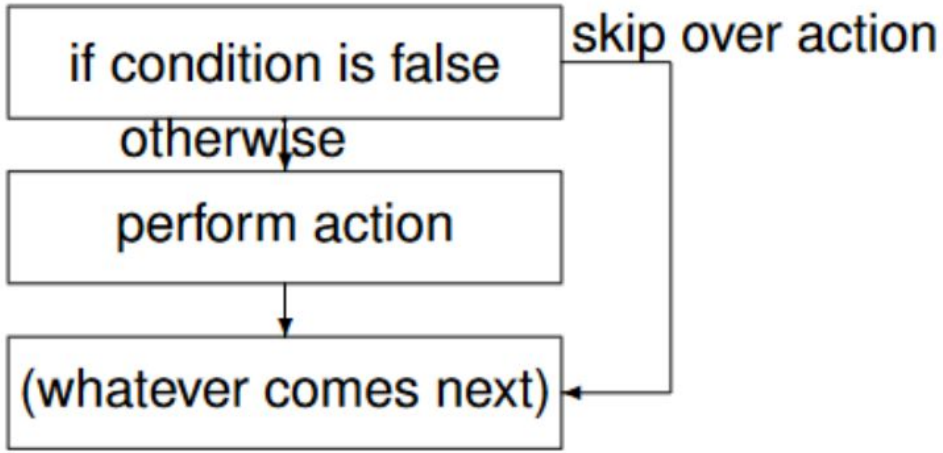
- `cmp` (compare - this sets the **conditional bits** NZCS)
- b/beq/bne/blt/bg/bge

This leads to a different approach in assembly than in C/Java etc

A generalized Java/C if-statement


```
if (condition) {
   action;
}
```

But in assembly we don't do it that way. We write code like this =>



Negating the condition

- evaluate arithmetic expressions with `cmp`
- compare inverted (NOT) conditional branch
- For example

```
if (a == b) → bne skip
if (a < b) → bge skip
if (a >= b)→ blt skip
```

Negating the condition

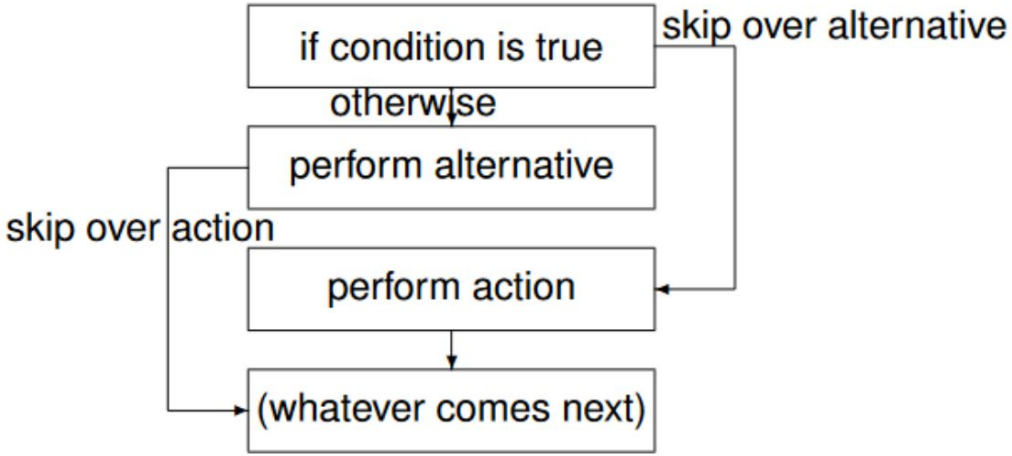
- Negating the condition is a useful optimization to favour not-taken branches and reduce pipeline stalls, especially on simple cores without branch prediction.
  - But it can make the assembly code harder to read and maintain, especially in complex control flows.
- For small code blocks,
  - prefer **conditional execution** to eliminate branches entirely.
  - When branches are unavoidable, structure the code to minimize taken branches in performance-critical paths, **but balance this with code readability**

Conditional Execution: Performance Impact

- *No Execution Cost*: When the condition evaluates to false, the instruction is not executed, meaning it does not modify registers, memory, or other system states. However, the instruction is still fetched and decoded, consuming some pipeline resources.
- *Pipeline Effects*: The penalty is typically limited to the fetch and decode stages (**1-2 cycles** in most ARM32 cores, like Cortex-A series).

Why not the other way around?


```
if (condition) {
   action;
} else {
   alternative;
}
```



Notice that if we take this approach, the code for the action is **further away** from the code for the condition test (making it less readable) and results in more branching which is highly undesirable

A note on `ldr / str`

Please note, in the following examples we use code like:

```
ldr r0, a
```

to show that the `a` variable is read from memory and placed into `r0`.

This is a syntactic simplification, and will not compile. Later we will address (!) this issue

Example if-else-statement


```
if(a == b) {
    c = d;
}  else {
    e = f;
}
```

```
a = 10, b = 12, c = 14, d = 4, e = 100
f = 5
```

```
start:
    ldr r0, a
    ldr r1, b
    cmp r0, r1
    bne else
    ldr r0, d
    str r0, c
    b both
else:
    ldr r0, f
    str r0, e
both: . . .
```

Iteration & loops

`while` loops

Loops and Branches

- Reducing the number of branching instructions is highly desirable.
- Code for the common case:
  - the negated conditions reduce stalls by avoiding a taken branch.
- Strive to reduce stalls by reducing the frequency of branching
  - especially in an architecture with poor branch prediction hardware

Example: C code with 2 Solutions

```
int a = 0;
int b = 100;
while (a < b) {
  a += 1;
  b -= 1;
}
```

```
mov r0, #0; //a
mov r1, #100; //b
loop:
  cmp r0, r1
  addlt r0, r0, #1
  sublt r1, r1, #1
  blt loop
  b end
```

```
mov r0, #0; //a
mov r1, #100; //b
test:
  cmp r0, r1
  blt less-than
  b end
less-than:
  add r0, r0, #1
  sub r1, r1, #1
  b test
```

is there a third assembly option here? yes

Best Practice

- Favor Conditional Execution:
  - Use ARM's conditional instructions (e.g., `moveq`, `addeq`) for small code blocks
  - reducing branching helps avoid pipeline flushes.
- Optimize for Not-Taken Branches:
  - If a branch is needed, structure the code so the most common case results in a not-taken branch
  - to minimize pipeline flushes, especially on cores like Cortex-M0.

Another Example: the v …. keyword?


```
while (a < b) {
    a = a * 2;
}
```

We have seen this pattern already … in C its called v….?

```
while:
    ldr r0, a
    ldr r1, b
    cmp r0, r1
    bge skip          //not lt
    mov r1, #2
    mul r0, r0, r1 //a in r0
    str r0, a
    b while ; unconditional
skip: . . .
```

Is the variable "`volatile`"?

- Prevents Optimization of Reads/Writes:
  - We might want to cache a variable's value in a register or reorder accesses for efficiency.
  - With *volatile*, every read must fetch the current value from memory, and every write must immediately update memory. This ensures the program sees the most up-to-date value.
- **No Assumptions About Thread Safety** : volatile does not make variables thread-safe (it doesn't prevent race conditions or provide atomicity).
- For that, use higher-level synchronization like mutexes. It's mainly for low-level scenarios where the compiler can't predict changes

Optimization if non-`volatile`


```
ldr r0, a
ldr r1, b
mov r2, #2
start:
    cmp r0, r1
    bge skip
    mul r0, r0, r2
    b start
skip:
    str r0, a
```

This reduces the number of **`str`** ops significantly

Which in turn improves the performance of the code

Is there any risk here?

A note on labels and flow


```
.global _start
_start:
     mov r0, #10
     mov r1, #2
test:
     cmp r1, r0
     blt loop
end:
     mov r15, #2
loop:
     add r1, r1, #2
     b test
```

What happens here when test is eventually false? Lets try this and see:

https://cpulator.01xz.net/?sys=arm

for loops

Counting up (and down)

for loops

- As you know, we use for loops to repeat a block of code a fixed number of times.
- In a for loop, the upper bound is known and we continue until that point has been reached
- So the terminating condition is when the iterator and the upper bound are the same

How do we iterate to some integer?

Consider the following for loop in C/Java/Javascript

```
int a = 0;
int q = 10
for (int i = 0; i < q; i++) {
    a += i;
}
```

How would we implement this in ARM Assembly?

What do we need?

- an accumulator (a)
- a counter (i)
- an upper bound (q)
- if test (cmp & b …)
- a setup label (so we don't repeat code unnecessarily)
- a loop label
- a finished label

Let's try it


```
setup:
    mov r0, #0        // this is a, the accumulator
    mov r1, #0        // this is our increment
    mov r2, #10       // this is q
loop:
    cmp r1, r2        // sets Z = 1 when r1 = r2
    beq finished      // we're finished
    add r0, r0, r1    // accumulate
    add r1, r1, #1    // increment
    b loop
finished:
    svc 2
```

Summary

- Branching is essential for if (conditional B) and loops (loop-back B).
- Conditional execution: Powerful alternative to branching for efficiency.
  - Faster, no jumps
  - But limited to small scopes
- Branching:
  - Flexible for complex logic
  - Can cause pipeline stalls.