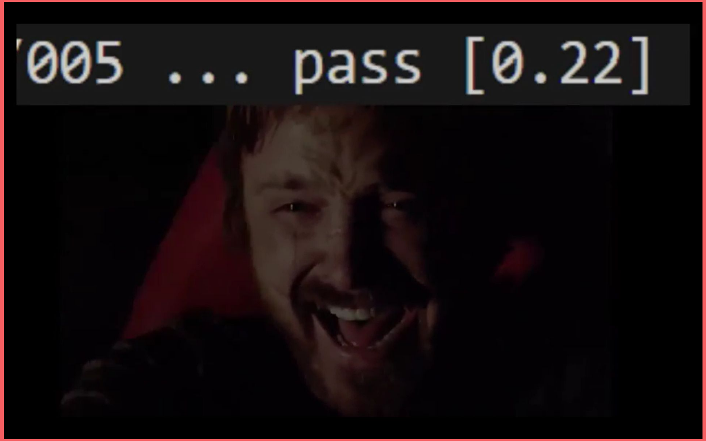
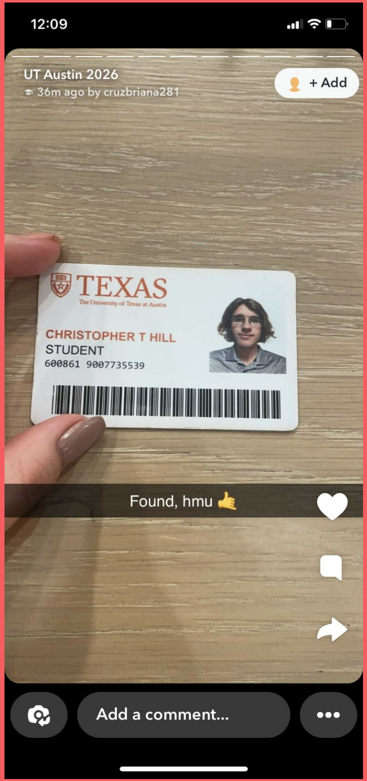
Welcome back^2 to CS429H!

Week 2

Best Ed meme of the week:



```
'005 ... pass [0.22]
```



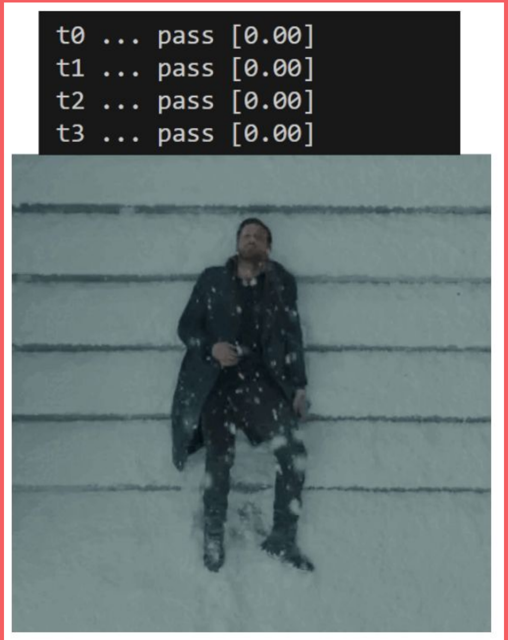
12:09

UT Austin 2026
36m ago by cruzbriana281

+ Add

Found, hmu 🤙

Add a comment...



```
t0 ... pass [0.00]
t1 ... pass [0.00]
t2 ... pass [0.00]
t3 ... pass [0.00]
```

Questions on lecture content?
Or about cats?

Quiz (review) everyone say YAY!

Question 1

[1 pts; 5 min] Give one example of **temporal** locality and one example of **spatial** locality within hardware or software.

Temporal Locality - Data is likely to be used again a short time after being used initially.

Ex. Registers storing values temporarily / for loop constantly using var i;

Spatial Locality - Data located close to other data that is used, is also likely to be used.

Ex. Cache bringing in 64 bytes of data at a time / moving through an array.

Question 2

[2 pts; 10 min] Most programming languages have a way to reference symbols defined in other files. In Java, this can be done with an **import** statement, which simply allows programmers to reference names without typing out the whole path, while in C this can be done with a **#include** directive. What is one benefit and one drawback of a language having **#include** rather than **import**?

Question 2

[2 pts; 10 min] Most programming languages have a way to reference symbols defined in other files. In Java, this can be done with an **import** statement, which simply allows programmers to reference names without typing out the whole path, while in C this can be done with a **#include** directive. What is one benefit and one drawback of a language having **#include** rather than **import**?

Benefits - More flexibility, can include non-symbols (strings, partial functions, etc), header-only libraries, file names don't have to match symbol names

Drawbacks - More error prone, double definition, circular dependencies, more work to use #ifdef/#ifndef to only include certain symbols, increases pre-processing time, linker errors

Question 3

```
mov $0xc0ffeecafe, %rax

mov $0xff, %rdi

sub $0xfe, %rdi

add %rax, %rdi

mov $0xb0ba, %ax
```

Question 3

```
mov $0xc0ffeecafe, %rax
mov $0xff, %rdi
sub $0xfe, %rdi
add %rax, %rdi
mov $0xb0ba, %ax
```

Initial state of registers is unknown

```
rax  0x????????????????     rdi  0x????????????????
```

Question 3

```
mov $0xc0ffeecafe, %rax

mov $0xff, %rdi

sub $0xfe, %rdi

add %rax, %rdi

mov $0xb0ba, %ax
```

Since the first argument is a literal, it can't be the dest, showing that this problem uses AT&T syntax

```
rax  0x000000c0ffeecafe    rdi  0x????????????????
```

Question 3

```
mov $0xc0ffeecafe, %rax

mov $0xff, %rdi

sub $0xfe, %rdi

add %rax, %rdi

mov $0xb0ba, %ax
```

rax	0x000000c0ffeecafe	rdi	0x00000000000000ff

Question 3

```
mov $0xc0ffeecafe, %rax

mov $0xff, %rdi

sub $0xfe, %rdi

add %rax, %rdi

mov $0xb0ba, %ax
```

```
sub src, dest

dest = dest - src

0xff - 0xfe = 0x01
```

```
rax  0x000000c0ffeecafe    rdi  0x0000000000000001
```

Question 3

```
mov $0xc0ffeecafe, %rax

mov $0xff, %rdi

sub $0xfe, %rdi

add %rax, %rdi

mov $0xb0ba, %ax
```

```
add src, dest

dest = dest + src

0x01 + 0xc0ffeecafe =
0xc0ffeecaff
```

```
rax  0x000000c0ffeecafe    rdi  0x000000c0ffeecaff
```

Question 3

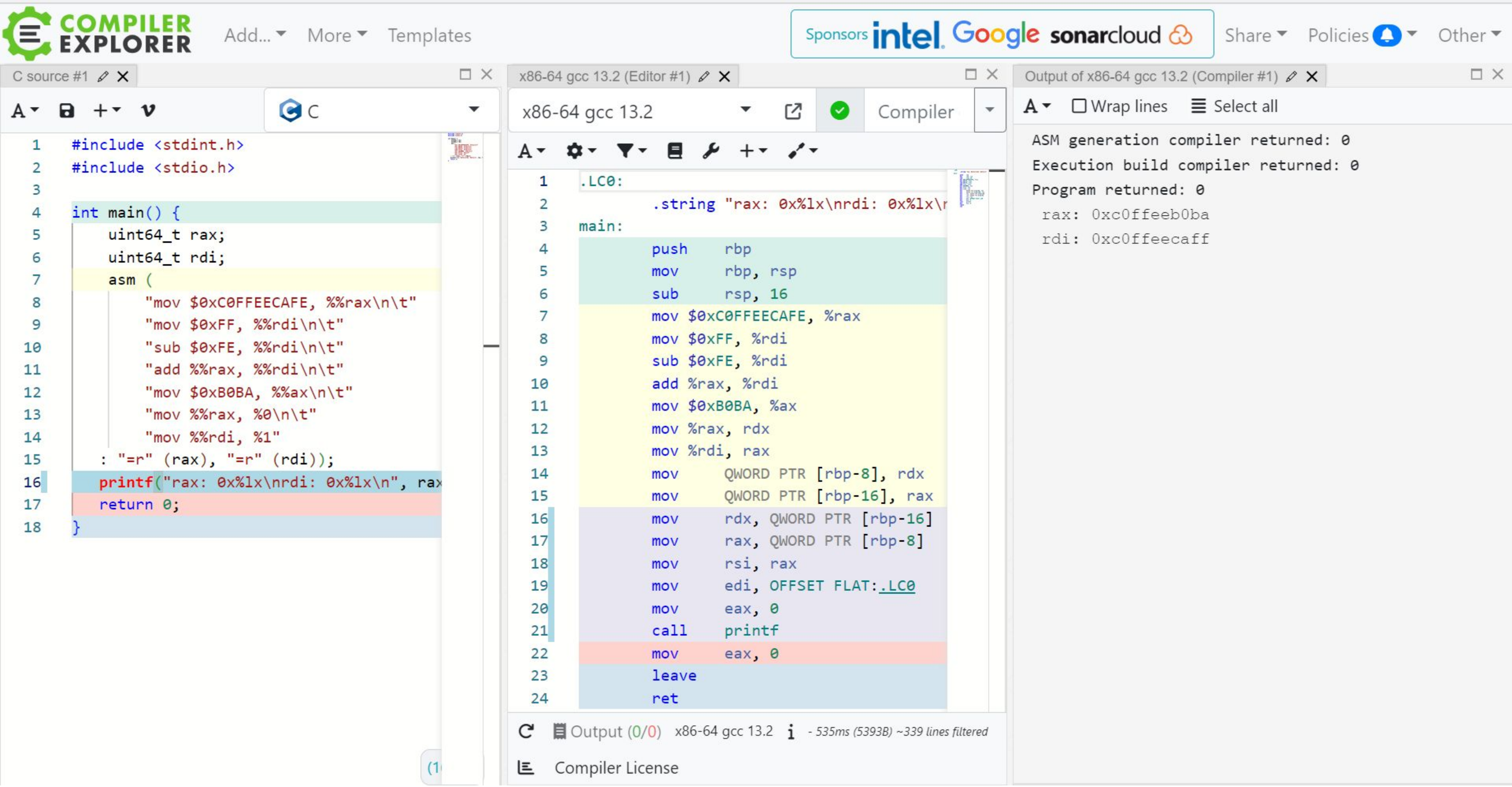
```
mov $0xc0ffeecafe, %rax

mov $0xff, %rdi

sub $0xfe, %rdi

add %rax, %rdi

mov $0xb0ba, %ax
```

%ax is the 16-bit version of `%rax` (%eax also exists and is the 32-bit version)

Writing to %ax does not cause zero-extension

rax 0x000000c0ffeeb0ba

rdi 0x000000c0ffeecaff

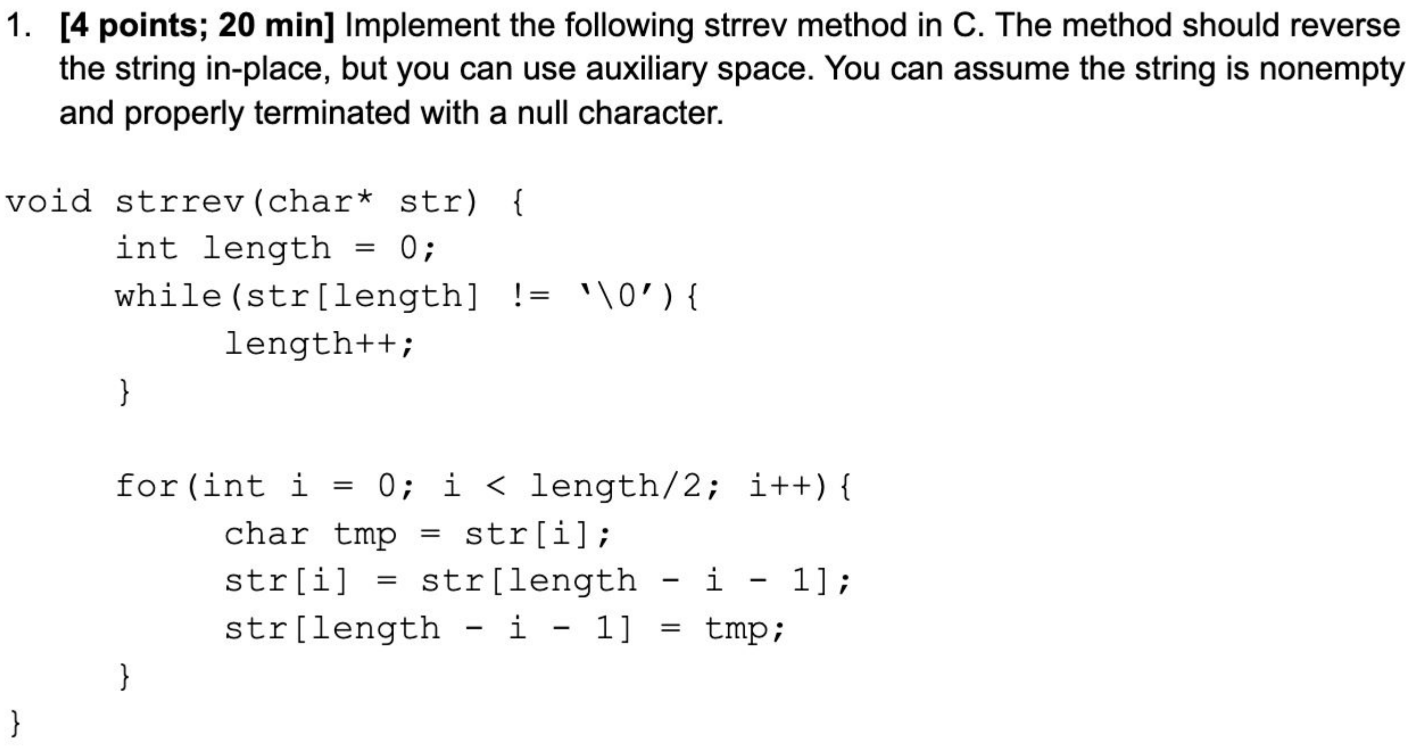


```c
#include <stdint.h>
#include <stdio.h>

int main() {
    uint64_t rax;
    uint64_t rdi;
    asm (
        "mov $0xC0FFEECAFE, %%rax\n\t"
        "mov $0xFF, %%rdi\n\t"
        "sub $0xFE, %%rdi\n\t"
        "add %%rax, %%rdi\n\t"
        "mov $0xB0BA, %%ax\n\t"
        "mov %%rax, %0\n\t"
        "mov %%rdi, %1"
      : "=r" (rax), "=r" (rdi));
      printf("rax: 0x%lx\nrdi: 0x%lx\n", rax
      return 0;
}
```

```asm
.LC0:
        .string "rax: 0x%lx\nrdi: 0x%lx\r
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov $0xC0FFEECAFE, %rax
        mov $0xFF, %rdi
        sub $0xFE, %rdi
        add %rax, %rdi
        mov $0xB0BA, %ax
        mov %rax, rdx
        mov %rdi, rax
        mov     QWORD PTR [rbp-8], rdx
        mov     QWORD PTR [rbp-16], rax
        mov     rdx, QWORD PTR [rbp-16]
        mov     rax, QWORD PTR [rbp-8]
        mov     rsi, rax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        leave
        ret
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
 rax: 0xc0ffeeb0ba
 rdi: 0xc0ffeecaff
```

Question 4



1. **[4 points; 20 min]** Implement the following strrev method in C. The method should reverse the string in-place, but you can use auxiliary space. You can assume the string is nonempty and properly terminated with a null character.

```
void strrev(char* str) {
     int length = 0;
     while(str[length] != ‘\0’){
          length++;
     }

     for(int i = 0; i < length/2; i++){
          char tmp = str[i];
          str[i] = str[length - i - 1];
          str[length - i - 1] = tmp;
     }
}
```

Question 4

- sizeof()
    - This gets the number of bytes a type takes up. It will return the size of a char*
- Double dereferencing
    - str[i] internally does; it's equivalent to *(str + i)
- Double swapping
    - If you go to length instead of length/2, the string gets reversed twice
- Reassigning str
    - In place means the memory location str pointed to at the beginning of the method execution should now contain the reversed string

P2

Poll

How’s your status on P2?

A. What’s P2?
B. I’ve heard of it
C. I’ve cloned the starter code and/or looked through it
D. I’ve started planning/writing code
E. I’m mostly done but might still have bugs
F. P2 any% speedrun

Miscellaneous p2 things

- Function evaluation - it's really just an operand with two expressions on either side...
- Recursive descent - understanding order of execution
- Tokenization / ASTs - if you don't know what this means, it's not too late

Assembly Review

- What is assembly?
  - It is the lowest-level human-readable interface to encode a sequence of instructions
- Why should we care about assembly?
  - It helps us understand what the machine is doing when we run compiled code
- What are the different types of assembly?
  - There are a *lot*: x86[_64], ARM, RISC-V, PowerPC, and more!
- **Why** are there different types of assembly?
  - Each corresponds to a different underlying **architecture**, with different abstractions and operations
- In this class, we will be discussing 2 architectures: AMD64 (x86_64), and AArch64 (ARM)
  - What are some differences between these architectures?

AMD64 vs. AArch64

- They both start with an A
- CISC
- Faster or slower per instruction?
- Why do you think AMD64 is so popular for laptop/desktop/server machines?
    - Will it be in the future?

- They both end with 64
- RISC
- More energy efficient or less energy efficient?
- Why is AArch64 so popular for embedded/mobile/microcontroller platforms?
    - Will it be in the future?

Emulator (P3)

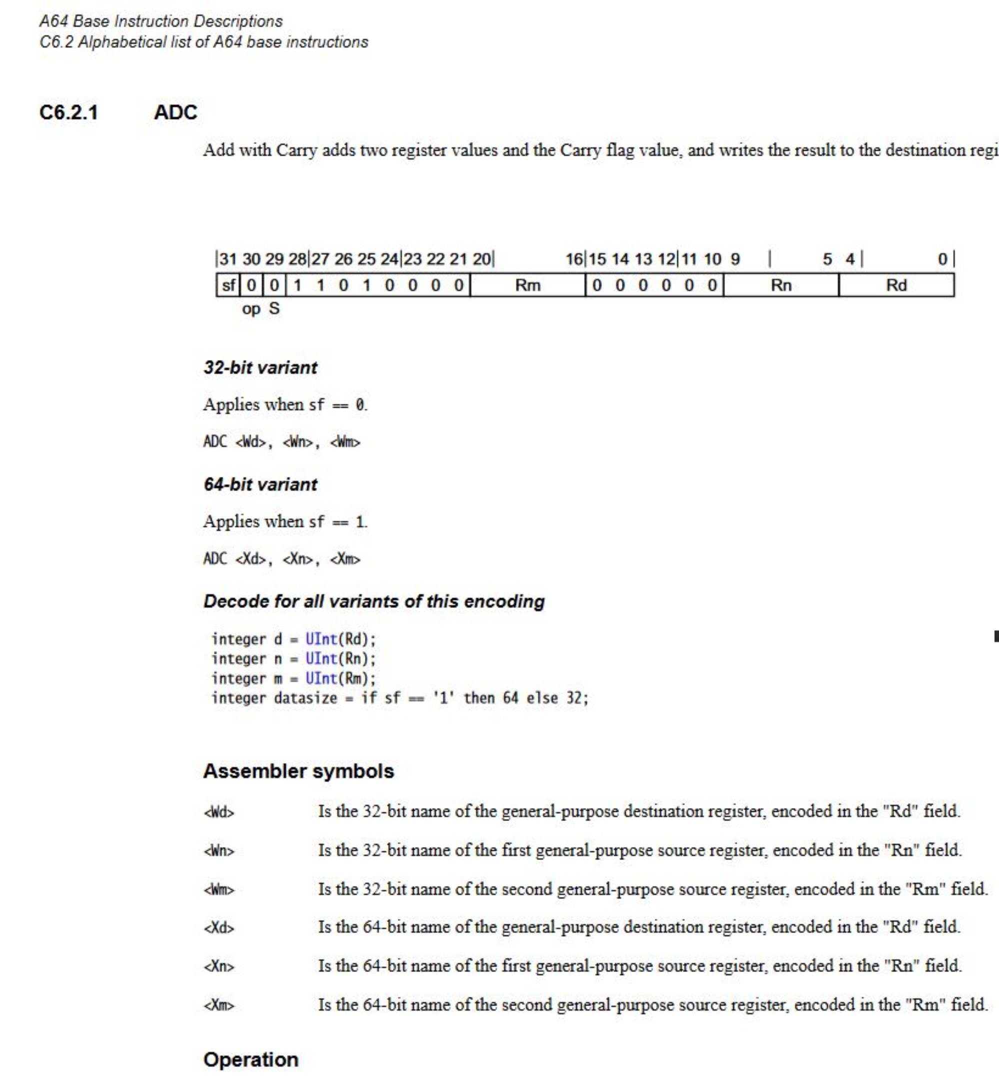
*all of this is very speculative since we don’t have the project yet so we’re mostly going off memory from when Alex did this project in 2022

What’s an emulator? Something ducks walk on?

- Software that imitates another system
- Architecture emulators - interpret machine language rather than directly using hardware of the host system
- Allow you to run software made for specific systems on other systems
  - Examples: qemu, Project64, BlueStacks

Assembly Crash Course (aka how to read)

https://developer.arm.com/documentation/ddi0487/latest/ ← this thing is going to be your best friend for the next few weeks



C6.2.1 ADC

Add with Carry adds two register values and the Carry flag value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
sf	0	0	1	1	0	1	0	0	0	0	Rm	0	0	0	0	0	0	Rn	Rd
	op	S																	

32-bit variant

Applies when `sf == 0`.

```
ADC <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
ADC <Xd>, <Xn>, <Xm>
```

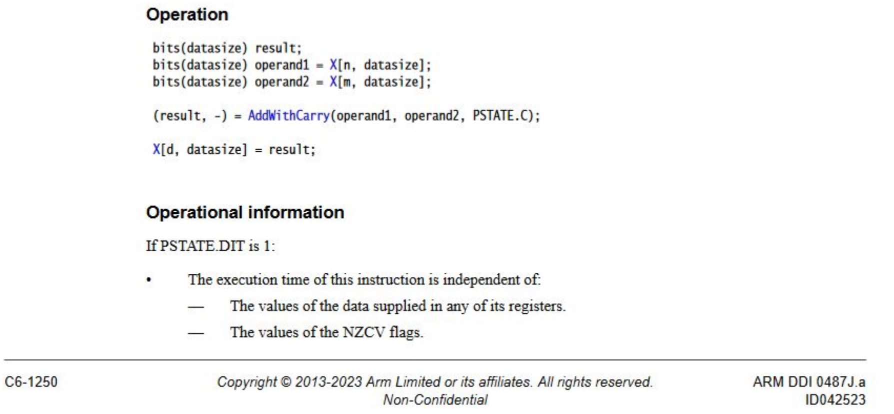
Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation



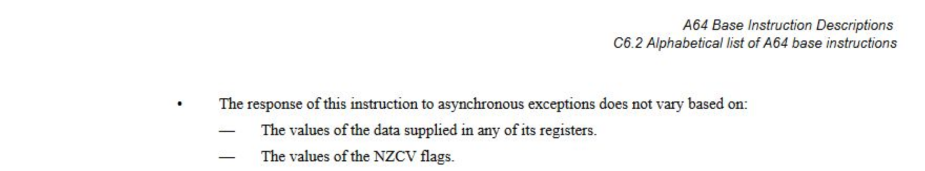
Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];

(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - — The values of the data supplied in any of its registers.
  - — The values of the NZCV flags.



- The response of this instruction to asynchronous exceptions does not vary based on:
  - — The values of the data supplied in any of its registers.
  - — The values of the NZCV flags.

Test Cases!

- Written in ARM (csid.s), assembled to machine code (csid.arm), and an out file (csid.ok)
- Good code quality for test cases:
  - Especially comments. Assembly is hard to read.
  - Make everyone's lives easier and include descriptive comments, explaining what your code is doing and what you are trying to test.
- The .s file may not be part of the test files. Consider adding them to the test case validity sheet, so people can actually debug with/understand your test.
  - But you can convert the machine code to assembly if someone doesn't provide this

Writing Assembly (ARM)

```
.section .data                         // initialized global/static variables
hello:
    .asciz "Hello\n"
num:
    .byte 15

.section .text                         // code goes in this section
.global _start
_start:
    movz  x0, #15
    adrp x1, :pg_hi21:hello            // load page number of hello
    add x1, x1, :lo12:hello            // store pointer to hello in x1

    adrp x3, :pg_hi21:num              // load page number of num into x3
    ldr x4, [x3, :lo12:num]            // load num into x4
```

save your file as csid.s

Compiling Assembly (ARM)

```
~gheith/public/gcc-arm-10.3-2021.07-x86_64-aarch64-none-linux-gnu/bin/
aarch64-none-linux-gnu-gcc -nostdlib csid.S -o csid.arm
```

(stay tuned to see if there are any changes to this command)

You can also add this directory to your PATH so you don’t have to type this all out:

```
export PATH=~gheith/public/gcc-arm-10.3-2021.07-x86_64-aarch64-none-linux-gnu/bin:$PATH
```

```
aarch64-none-linux-gnu-gcc -nostdlib csid.S -o csid.arm
```

Disassemble Machine Code (to ARM)

- objdump is your friend!!
- .arm test case files are binary files and pretty unreadable by default
- objdump can output human-readable assembly code

```
~gheith/public/gcc-arm-10.3-2021.07-x86_64-aarch64-none-linux-gnu/bin
/aarch64-none-linux-gnu-objdump -d csid.arm
```

Same directory as last slide so if that is on your PATH then this should work:

```
aarch64-none-linux-gnu-objdump -d csid.arm
```

Questions?

```
                          oooo$$$$$$$$$$$$oooo
                      oo$$$$$$$$$$$$$$$$$$$$$$$$o
                   oo$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$o         o$   $$ o$
   o $ oo        o$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$o       $$ $$ $$o$
oo $ $ "$      o$$$$$$$$$    $$$$$$$$$$$$$    $$$$$$$$$o       $$$o$$o$
"$$$$$$o$     o$$$$$$$$$      $$$$$$$$$$$      $$$$$$$$$$o    $$$$$$$$
  $$$$$$$    $$$$$$$$$$$      $$$$$$$$$$$      $$$$$$$$$$$$$$$$$$$$$$$
  $$$$$$$$$$$$$$$$$$$$$$$    $$$$$$$$$$$$$    $$$$$$$$$$$$$$  """$$$
   "$$$""""$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$     "$$$
    $$$   o$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$     "$$$o
   o$$"   $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$       $$$o
   $$$    $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$" "$$$$$$ooooo$$$$o
  o$$$oooo$$$$$  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$   o$$$$$$$$$$$$$$$$$
  $$$$$$$$"$$$$   $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$     $$$$""""""""
 """"       $$$$    "$$$$$$$$$$$$$$$$$$$$$$$$$$$$"      o$$$
            "$$$o     """$$$$$$$$$$$$$$$$$$"$$"         $$$
              $$$o          "$$""$$$$$$""""           o$$$
               $$$$o                                o$$$"
                "$$$$o      o$$$$$$o"$$$$o        o$$$$
                  "$$$$$oo     ""$$$$o$$$$$o   o$$$$""
                     ""$$$$$oooo  "$$$o$$$$$$$$$"""
                        ""$$$$$$$oo $$$$$$$$$$
                                """"$$$$$$$$$$$
                                    $$$$$$$$$$$$
                                     $$$$$$$$$$"
                                      "$$$""""
```