COMP311: *COMPUTER ORGANIZATION!*

Lecture 25: Caching, Stack Review, Hardware Security, The End

tinyurl.com/comp311-sp26

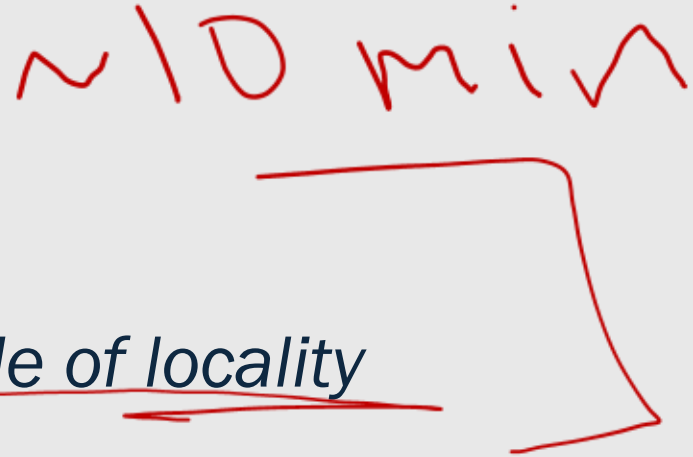
Logistics Check-in

- Lab 5 due tonight
- Final review session tomorrow (4/28)
  - *7pm, SN014*
- Final exam…

Final Exam Logistics!

- In this room! **4/30th 4-7pm (or may 4th if you are in section 1)**
- **Updated assigned seats** will be posted on Canvas the day before
- Test format will be the same as the quizzes
  - *Expected length: ~2.5x quizzes*
- Exam is **cumulative**
  - *Equally distributed across all topics.*
- A few extra practice problems will be posted on the assignments tab.
  - *Not exhaustive – just a study tool! Your awesome TAs are generous (and haven't seen the exam ☺)*
  - *Going over *in-class practice* & written assignments will be helpful!*
- You will get a multi-page reference sheet. I will post it tomorrow!

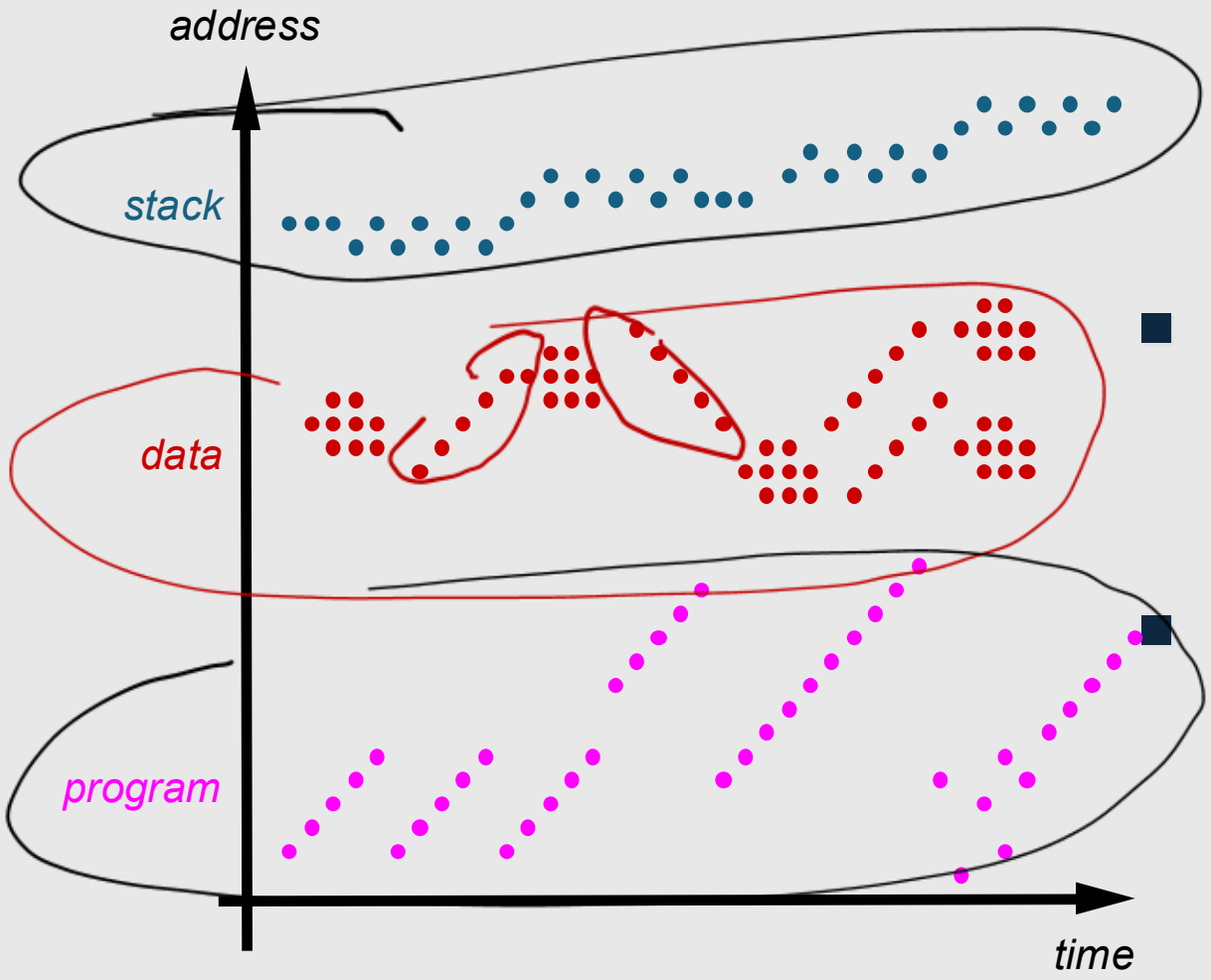
Plan for today…



- Finish caching! It will be on final:
  - *Conceptually, what is caching & the principle of locality*
  - *In-class practice from last time*
- Review of stack concepts ☺
  - *One more in-class practice to review*
- Hardware Security!

CACHING CONT.

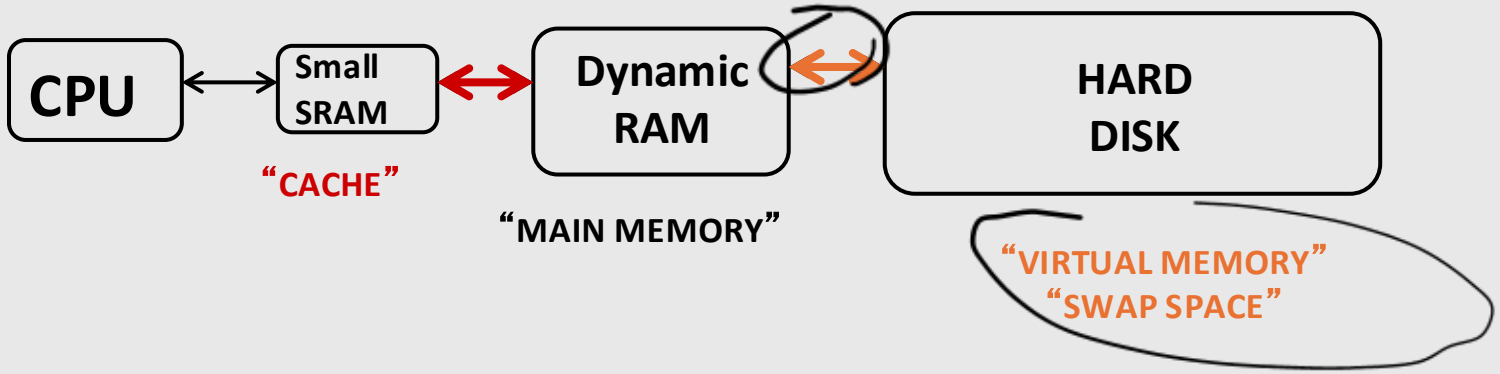
Typical Memory Reference Patterns



- Memory Trace
  - *A temporal sequence of memory references (addresses) from a real program.*
- Temporal Locality
  - *If an item is referenced, it will tend to be referenced again soon*
- Spatial Locality
  - *If an item is referenced, nearby items will tend to be referenced soon.*

Exploiting the Memory Hierarchy

- CPU speed is dominated by memory performance
  - *More significant than: ISA, circuit optimization, pipelining, etc.*

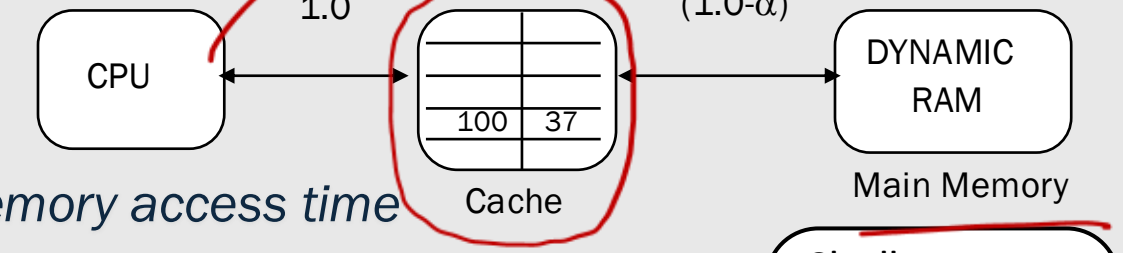


- *TRICK #1: Make slow MAIN MEMORY appear faster*
  - Technique: CACHING
- *TRICK #2: Make small MAIN MEMORY appear bigger*
  - Technique: VIRTUAL MEMORY

The Cache Idea

Program-Transparent Memory Hierarchy:

- *Cache contains TEMPORARY COPIES of selected main memory locations…*
  - e.g. Mem[100] = 37



- Two Goals:
  - *Improve the average memory access time*
    - HIT RATIO (α): Fraction of refs found in cache
    - MISS RATIO (1- α): Remaining references
    - avg total access time depends on these parameters

$$t_{ave} = \alpha t_c + (1-\alpha)(t_c + t_m) = t_c + (1-\alpha)t_m$$

$$t_c = \text{time to access cache}$$

$$t_m = \text{time to access main memory}$$

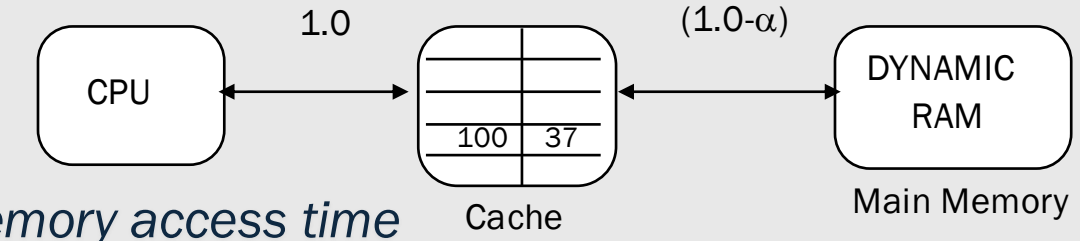
  - *Transparency (compatibility, programming ease)*

Challenge:
To make the hit ratio as high as possible.

The Cache Idea

Program-Transparent Memory Hierarchy:

- *Cache contains TEMPORARY COPIES of selected main memory locations…*
  - e.g. Mem[100] = 37



- Two Goals:
  - *Improve the average memory access time*
    - HIT RATIO (α): Fraction of refs found in cache
    - MISS RATIO (1- α): Remaining references
    - avg total access time depends on these parameters

$$t_{ave} = \alpha t_c + (1-\alpha)(t_c + t_m) = t_c + (1-\alpha)t_m$$

$$t_c = \text{time to access cache}$$

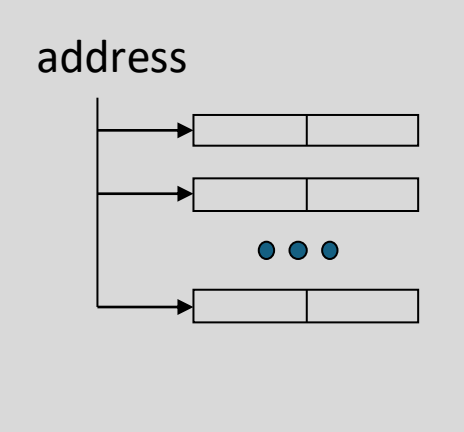
$$t_m = \text{time to access main memory}$$

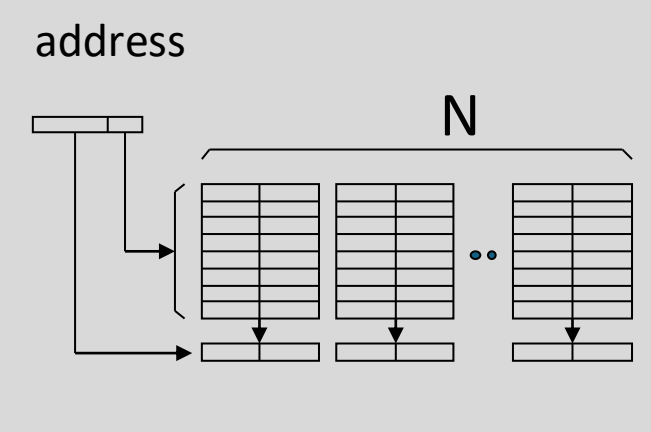
  - *Transparency (compatibility, programming ease)*

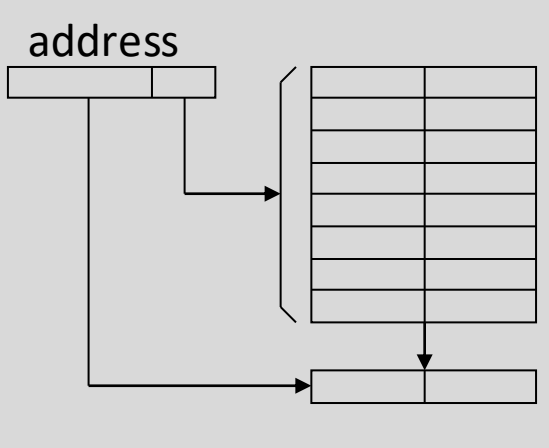
Challenge:
To make the hit ratio as high as possible.

Continuum of Associativity

Fully associative | N-way set associative | Direct-mapped







Compares addr with ALL tags simultaneously location A can be stored in any cache line.

Compares addr with N tags simultaneously. Data can be stored in any of the N cache lines belonging to a "set" like N direct-mapped caches.

Compare addr with only ONE tag. Location A can be stored in exactly one cache line.

+ less conflicts

What happens on a MISS?

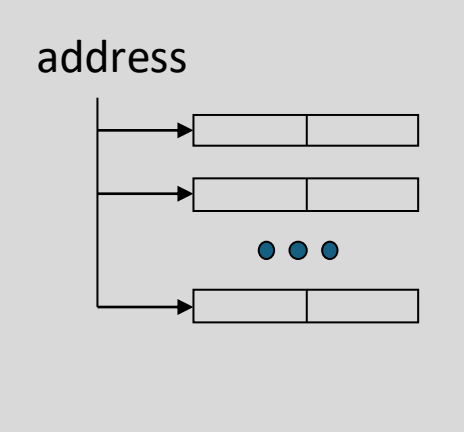
- expensive

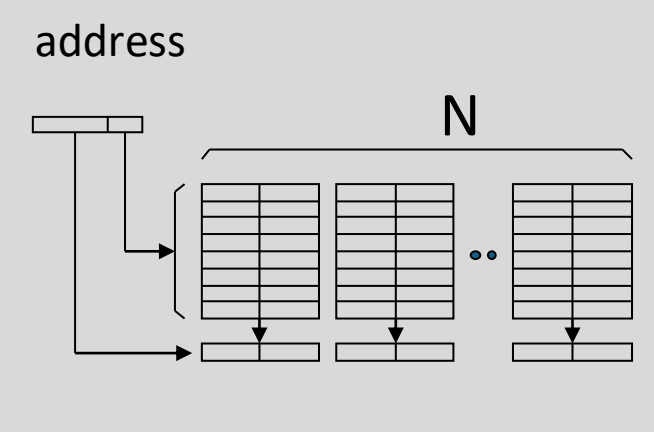
+ simple/fast

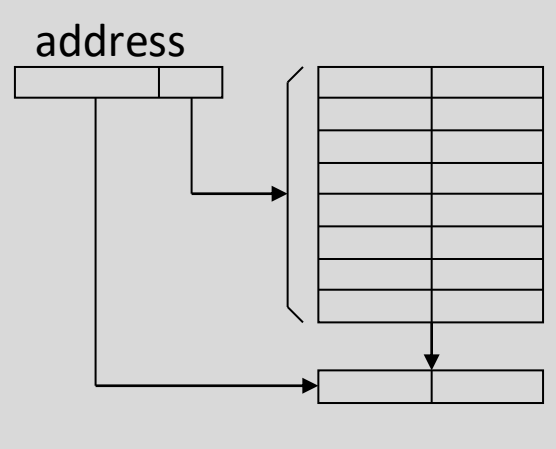
- more collisions

Continuum of Associativity

Fully associative | N-way set associative | Direct-mapped







Compares addr with ALL tags simultaneously location A can be stored in any cache line.

Compares addr with N tags simultaneously. Data can be stored in any of the N cache lines belonging to a "set" like N direct-mapped caches.

Compare addr with only ONE tag. Location A can be stored in exactly one cache line.

What happens on a MISS?

Finds one entry out of entire cache to evict

Finds one entry out of N in a particular row to evict

There is only one place it can go

Three Replacement Strategies

- When an entry has to be evicted, how to pick the victim?
  - *LRU (Least-recently used)*
    - replaces the item that has gone UNACCESSED the LONGEST
    - favors the most recently accessed data
  - *FIFO/LRR (first-in, first-out/least-recently replaced)*
    - replaces the OLDEST item in cache
    - favors recently loaded items over older STALE items
  - *Random*
    - replace some item at RANDOM
    - no favoritism – uniform distribution
    - no "pathological" reference streams causing worst-case results
    - use pseudo-random generator to get reproducible behavior

Handling WRITES

- Observation: Most (80+%) of memory accesses are reads, but writes are essential. How should we handle writes?
  - *Two different policies:*
    - WRITE-THROUGH: CPU writes are cached, but also written to main memory (stalling the CPU until write is completed). Memory always holds the latest values.
    - WRITE-BACK: CPU writes are cached, but not immediately written to main memory. Main memory contents can become "stale". Only when a value has to be evicted from the cache, and only if it had been modified (i.e., is "dirty"), only then it is written to main memory.
  - *Pros and Cons?*
    - WRITE-BACK typically has higher performance
    - WRITE-THROUGH typically causes fewer consistency problems

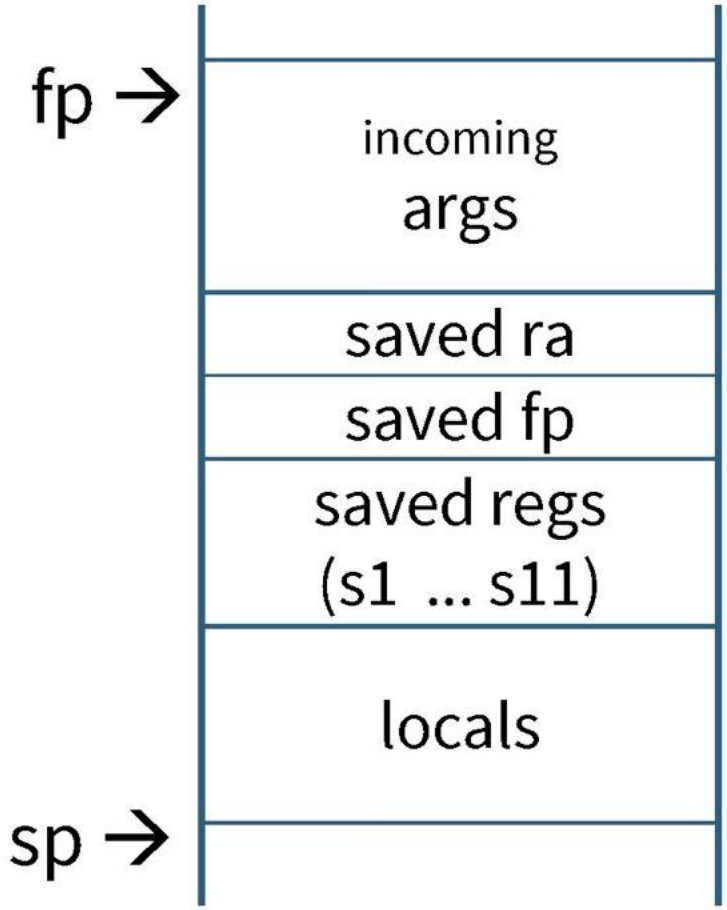
Memory Hierarchy Summary

- Give the illusion of fast, big memory
  - *small fast cache makes the entire memory appear fast*
  - *large main memory provides ample storage*
  - *even larger hard drive provides huge virtual memory (TB)*
- Various design decisions affect caching
  - *total cache size, line size, replacement strategy, write policy*
- Performance
  - *Put your money on bigger caches, then bigger main memory.*
  - *Don't put your money on CPU clock speed (GHz) because memory is usually the culprit!*

STACK REVIEW

Adapted from material from Leonard McMillian,
Don Porter, Montek Singh

RISC-V Calling Convention Cheat Sheat/Summary



Source:
https://www.cs.cornell.edu/courses/cs3410/2025fa/notes/asm-call.html

- *first eight* args passed in registers `a0`, `a1`, ..., `a7`
- Space for args passed in *childs's* stack frame
- return value (if any) in `a0`, `a1`
- stack frame at `sp`
  - contains `ra` (clobbered on JAL to sub-functions)
  - contains `fp`
  - contains *local vars* (possibly clobbered by sub-functions)
  - contains *space for incoming args*
- *Saved registers* (callee save regs) are *preserved*
- *Temporary registers* (caller save) regs are *not*

RISC-V Registers

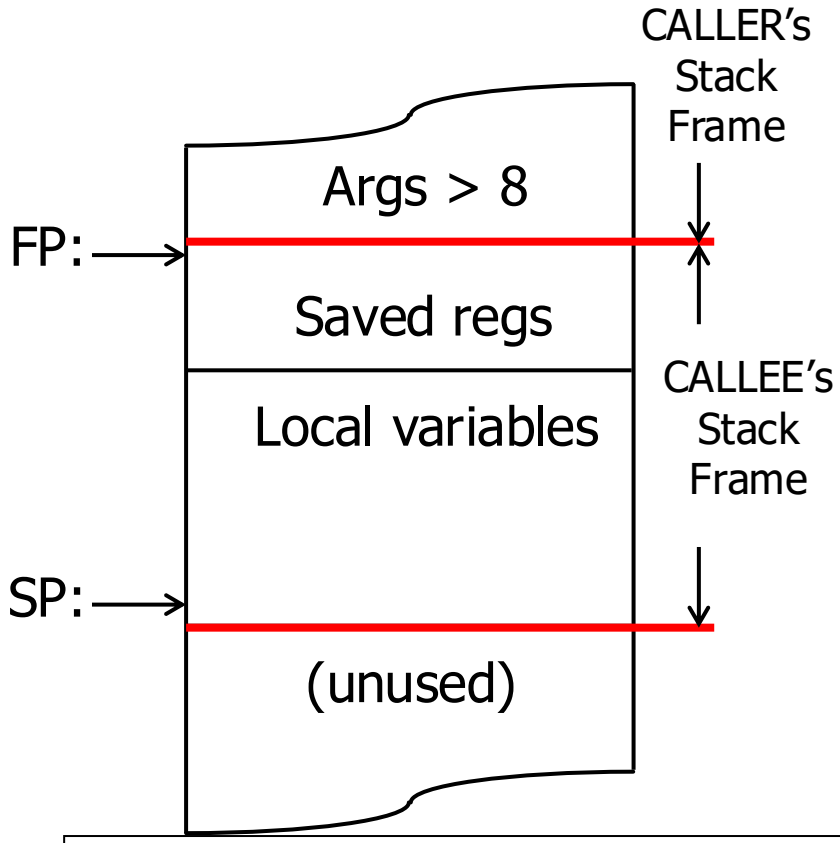
- Return address: `x1` (`ra`)
- Stack pointer: `x2` (`sp`)
- Frame pointer: `x8` (`fp/s0`)
- First eight arguments: `x10-x17` (`a0-a7`)
- Return result: `x10-x11` (`a0-a1`)
- Callee-save free regs: `x18-x27` (`s2-s11`)
- Caller-save free regs: `x5-x7`, `x28-x31` (`t0-t6`)
- Global pointer: `x3` (`gp`)
- Thread pointer: `x4` (`tp`)

Stack Frame or "Activation Record"

The STACK FRAME contains storage for the CALLER's volatile state that it wants preserved after the invocation of CALLEEs.

In addition, the CALLEE will use the stack for the following:

1) Accessing the arguments that the CALLER passes to it (specifically, the 9th and greater)
2) Saving non-temporary registers that it wishes to modify
3) Accessing its own local variables



It is possible to use only the SP to access a stack frame, but offsets may change due to ALLOCATEs and DEALLOCATEs. For convenience a fp is (optionally) used to provide CONSTANT offsets to local variables and arguments

Procedure Stack Usage

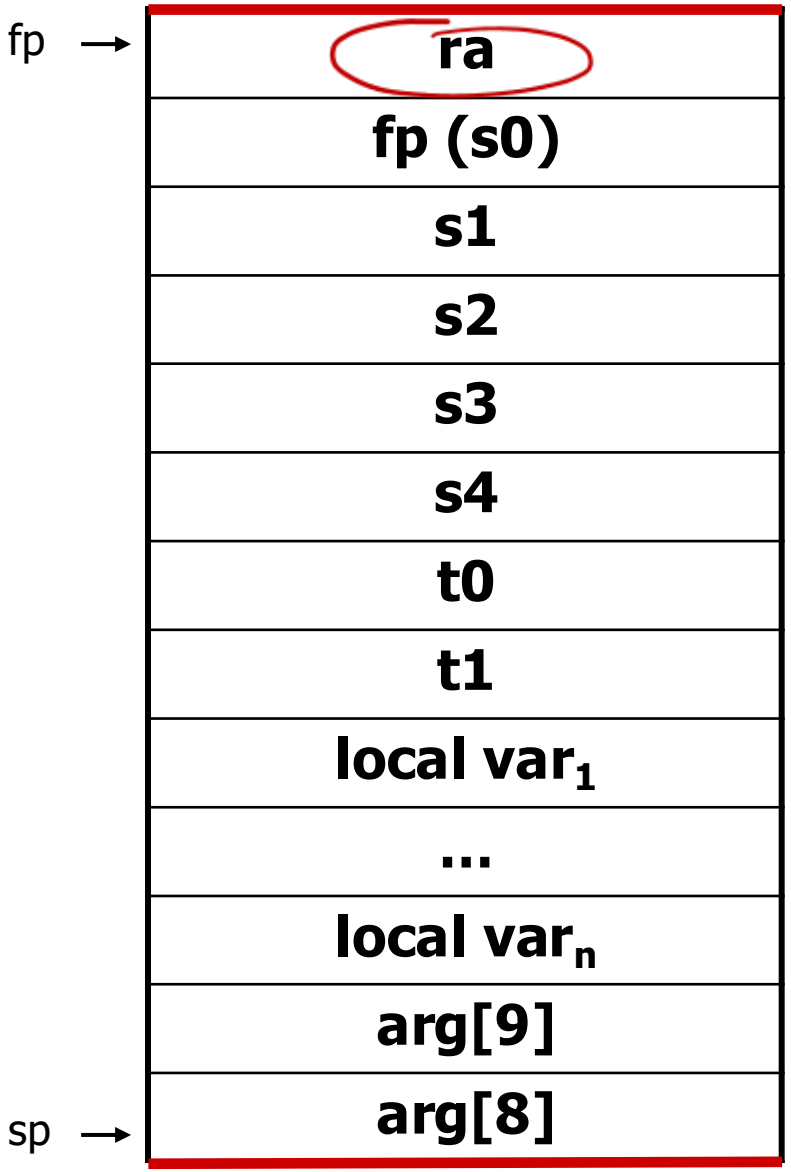
ADDITIONAL space must be allocated in the stack frame for:

1. Any SAVED registers the procedure uses (s0-s11, ra)
2. Any TEMPORARY registers that the procedure wants preserved IF it calls other procedures (t0-t6, a0-a7)
3. Any LOCAL variables declared within the procedure
4. Other TEMP space IF the procedure runs out of registers (RARE)
5. Enough "outgoing" arguments to satisfy the worse case ARGUMENT SPILL of ANY procedure it calls. (SPILL is the number of arguments greater than 8).

Each procedure must keep track of how many saved registers, temporary registers, and local variables are on the stack in order to calculate offsets..

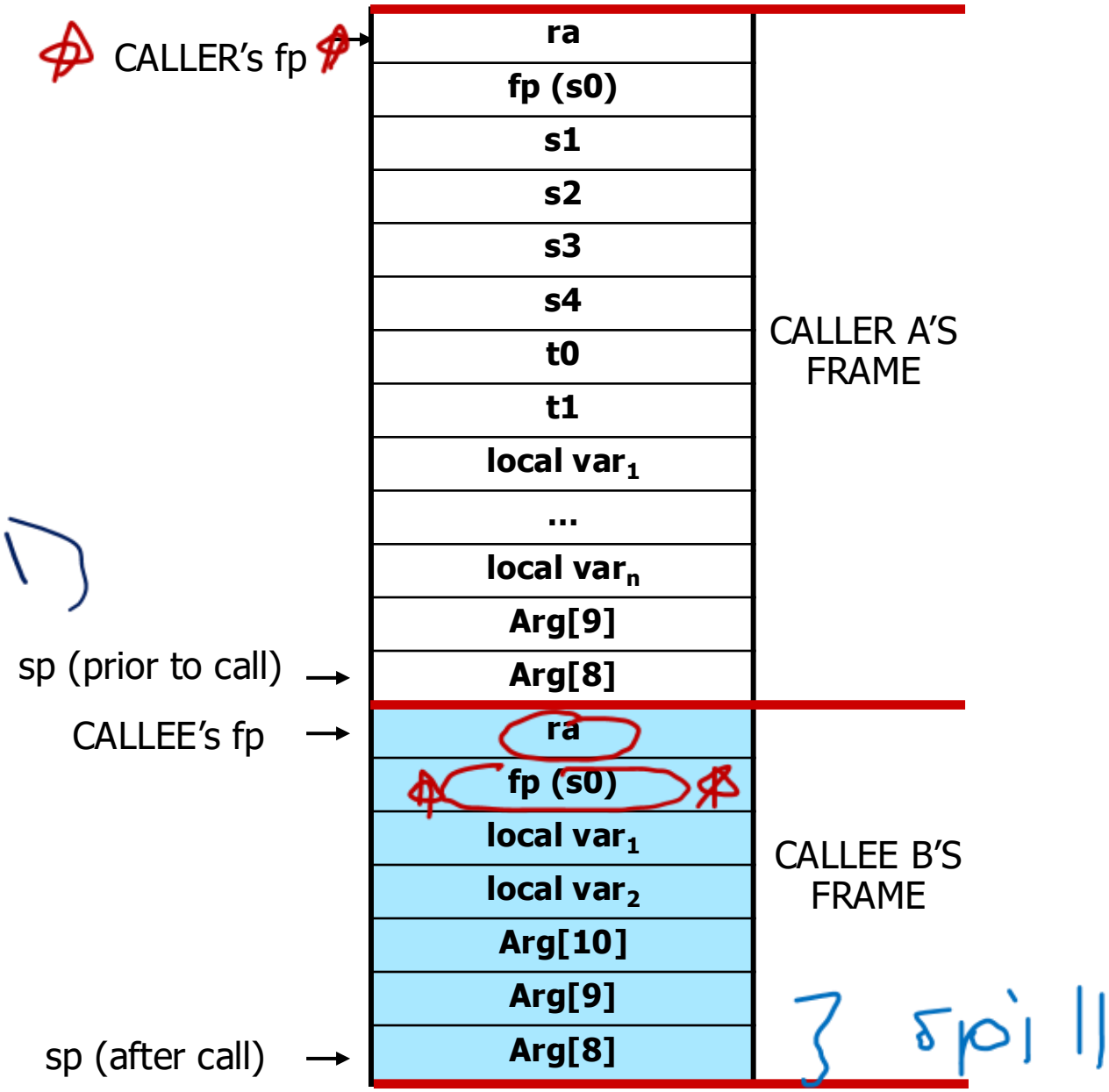
Stack Snap Shot: A typical procedure

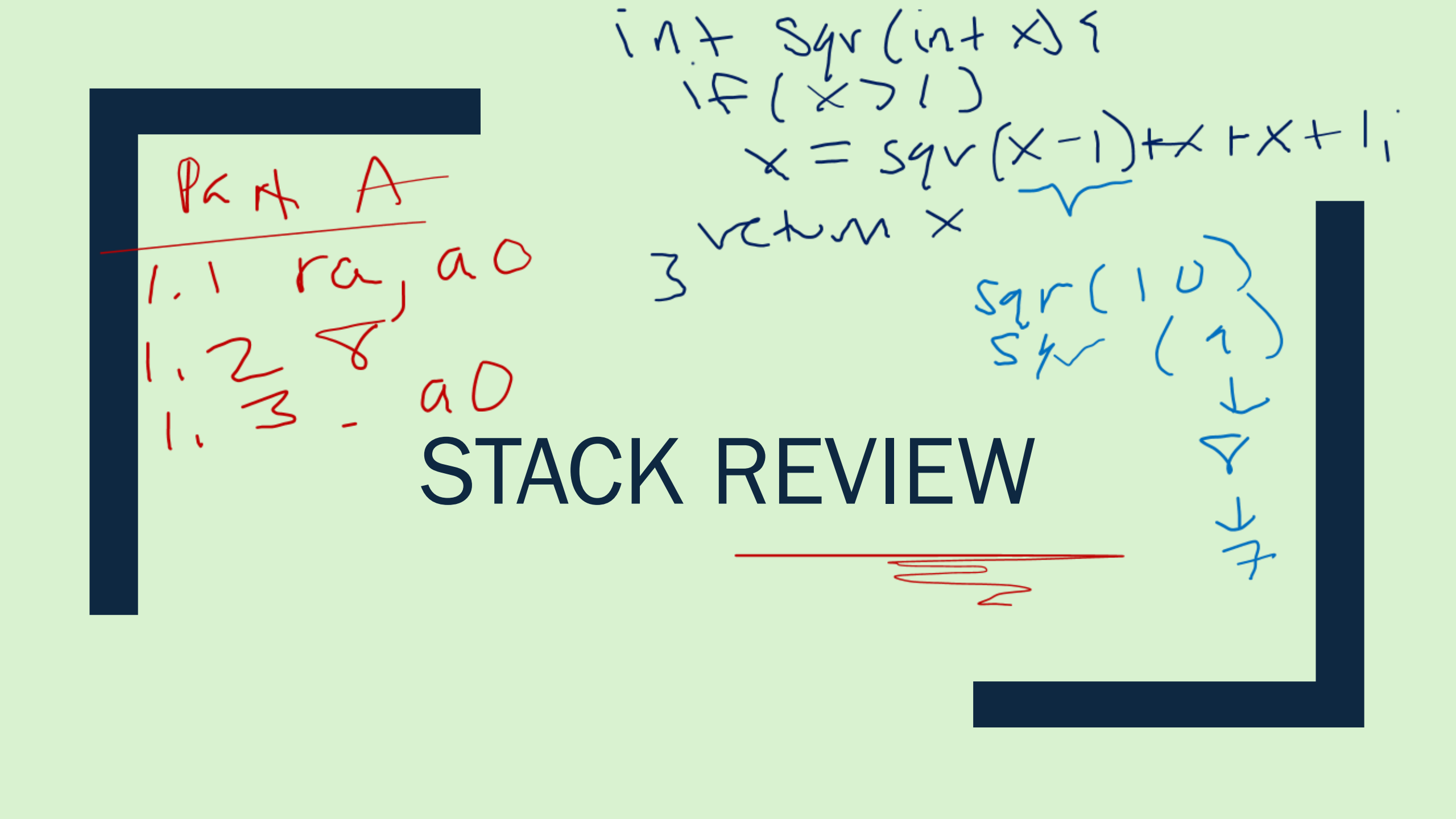
- A typical stack frame
  - save ra and (fp/s0 if used)
  - save values of "saved regs" modified by this proc
    - e.g.: s1, s2, s3
  - save values of "caller-saved regs" that must survive calls to other procs from this proc
    - e.g.: t0, t1
    - should be saved immediately before calling other proc; restored immediately after
  - any local variables needed (that are not in regs) reside on the stack
    - e.g.: locals var_1 … var_n
  - any spillover args for calling other procs
    - e.g.: arg[8], arg[9]



Stack Snap Shot: Caller + Callee

- proc A calls proc B
  - B has less stuff that needs to be saved on stack
  - Can you tell the number of args for B?
    - NOPE! (only if spill)
    - Registers hide part of the calling convention → stack only shows overflow
  - Where in CALLEE's stack frame might one find CALLER's fp?
    - At -4(fp)

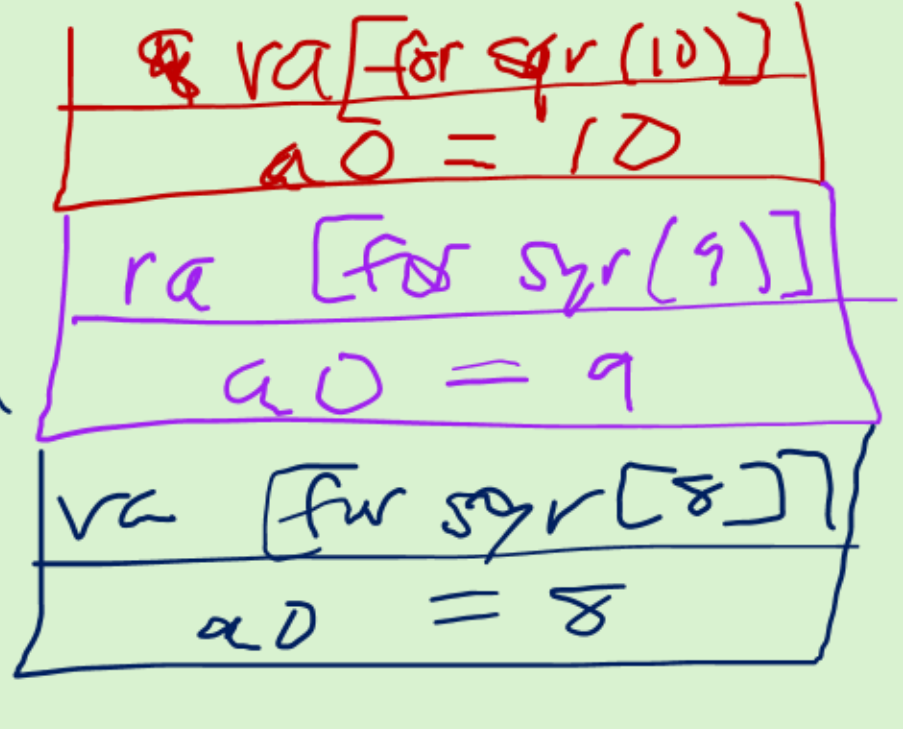



int Sqr(int x){
if(x>1)
x = sqr(x-1)+x+x+1;
return x
3
Part A
1.1 ra, a0
1.2 8
1.3 - a0
sqr(10)
sqr(1)
7
STACK REVIEW

STACK REVIEW

Part D

1.1
ra is overwritten by JALR



1.2
a0 is modified, so we saved the original val to use after the call

1.3 False

Procedure Linkage is Nontrivial

- The details can be overwhelming. How do we manage this complexity?
  - Abstraction: High-level languages hide the details

- There are great many implementation choices:
  - which variables are saved
  - who saves them
  - where are arguments stored?

- Solution: CONTRACTS!
  - Caller and Callee must agree on the details

Procedure Linkage: Caller Contract

The CALLER will:

- Save all caller-saved registers that it wants to survive subsequent calls in its stack frame
  (t0-t6, a0-a7)
- Pass the first 8 arguments in registers a0-a7, and save subsequent arguments on stack,

  Call procedure, using a **`jal`** instruction
  (places return address in ra).
- Access procedure's return values in a0-a1

Code Lawyer

- Our running example is a CALLER. Let's make sure it obeys its contractual obligations

```
    addi sp, sp, -8
    sw   ra, 4(sp)
    sw   a0, 0(sp)

    slti t0, a0, 2
    bne  t0, x0, base

    addi a0, a0, -1
    jal  ra, sqr

    lw   a0, 0(sp)
    add  a0, a0, a0
    add  a0, a0, a0
    addi a0, a0, -1
    j    done

base:
    # return n

done:
    lw   ra, 4(sp)
    addi sp, sp, 8
    jr   ra
```

Code Lawyer

- Our running example is a CALLER. Let's make sure it obeys its contractual obligations

The CALLER will:

- Save all caller-saved registers that it wants to survive subsequent calls in its stack frame
  (t0-t6, a0-a7)
- Pass the first 8 arguments in registers a0-a7, and save subsequent arguments on stack,

  Call procedure, using a **`jal`** instruction
  (places return address in ra).
- Access procedure's return values in a0-a1

```
    addi sp, sp, -8
    sw   ra, 4(sp)
    sw   a0, 0(sp)

    slti t0, a0, 2
    bne  t0, x0, base

    addi a0, a0, -1
    jal  ra, sqr

    lw   a0, 0(sp)
    add  a0, a0, a0
    add  a0, a0, a0
    addi a0, a0, -1
    j    done

base:
    # return n

done:
    lw   ra, 4(sp)
    addi sp, sp, 8
    jr   ra
```

Procedure Linkage: Callee Contract

If needed the CALLEE will:

1) Allocate a stack frame with space for saved registers, local variables, and spilled args

2) Save any "preserved" registers used:
(s0-s11, and ra if we call another fn)

3) If CALLEE has local variables -or- needs access to args on the stack, save CALLER's frame pointer and set fp (s0) to a fixed location in stack frame (optional)

4) EXECUTE procedure
5) Place return values in a0-a1
6) Restore saved registers
7) Restore sp to its original value
8) Return to CALLER with jr ra or ret

Code Lawyer: w/ Callee this Time

- Our running example is also a CALLEE. Are these contractual obligations satisfied?

If needed the CALLEE will:

1) Allocate a stack frame with space for saved registers, local variables, and spilled args

2) Save any "preserved" registers used: (s0-s11, and ra if we call another fn)

3) If CALLEE has local variables -or- needs access to args on the stack, save CALLER's frame pointer and set fp (s0) to a fixed location in stack frame (optional)

4) EXECUTE procedure
5) Place return values in a0-a1
6) Restore saved registers
7) Restore sp to its original value
8) Return to CALLER with jr ra or ret

```
    addi sp, sp, -8
    sw   ra, 4(sp)
    sw   a0, 0(sp)

    slti t0, a0, 2
    bne  t0, x0, base

    addi a0, a0, -1
    jal  ra, sqr

    lw   a0, 0(sp)
    add  a0, a0, a0
    add  a0, a0, a0
    addi a0, a0, -1
    j    done

base:
    # return n

done:
    lw   ra, 4(sp)
    addi sp, sp, 8
    jr   ra
```

Conclusions

- Need a convention (contract) between caller and callee
- Implement stack for storing each procedure's variables
- Procedure calls can now be arbitrarily nested
  - Recursion possible too

- FOLLOW the convention meticulously when programming!

PROCEDURE CALL TEMPLATES

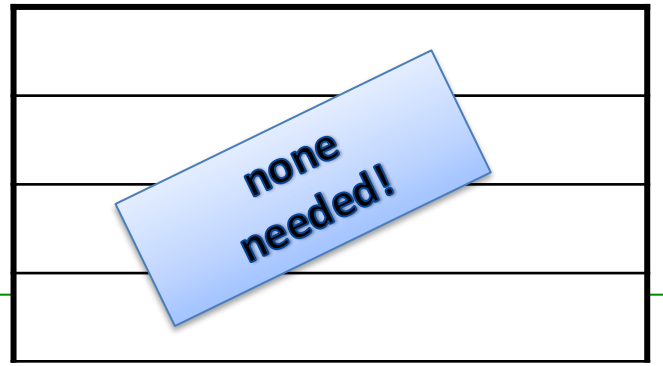
Adapted from material from Leonard McMillian,
Don Porter, Montek Singh

Ex1: simple leaf proc (w/o stack frame)

- main:
  - doesn't need to preserve any temporary registers
- proc1:
  - is a leaf proc (doesn't call any other proc)
  - doesn't touch any saved regs

```
main:
    jal proc1           # call proc1
    ...                 # other main stuff

proc1:
    ...                 # do the task
    jr ra               # return
```

stack frame



You only need a stack frame if you have something to save!

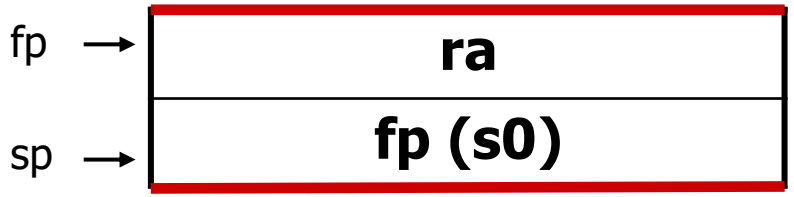
Ex2: leaf proc with stack frame

- main:
  - doesn't need to preserve any temporary registers
- proc1:
  - is leaf; doesn't touch any saved regs
  - creates minimal stack frame (but doesn't really need it)


```
main:
    jal  ra, proc1          # call proc1
    ...                     # other main stuff

proc1:
    addi sp, sp, -8         # allocate stack frame
    sw   ra, 4(sp)          # save ra
    sw   s0, 0(sp)          # save old fp
    addi s0, sp, 8          # set fp

    ...                     # do the task

    lw   s0, 0(sp)          # restore old fp
    lw   ra, 4(sp)          # restore ra
    addi sp, sp, 8          # deallocate stack frame
    jr   ra                 # return
```

Ex3: minimal non-leaf proc

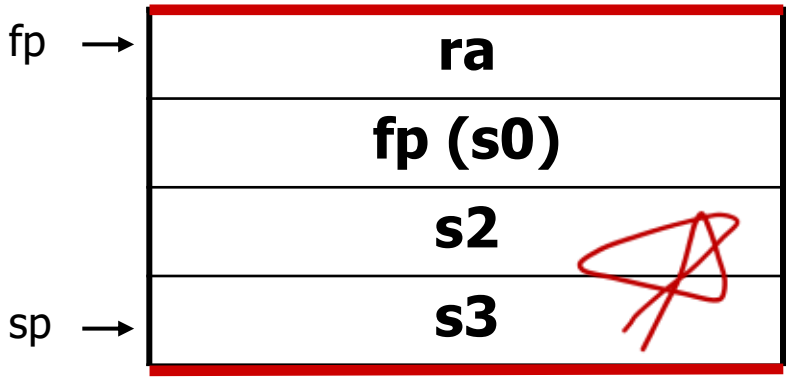
- proc1:
  - calls proc2
  - doesn't need to preserve any data values in regs
  - creates minimal stack frame (and needs it for saving return address)




```
proc1:
    addi sp, sp, -8         # allocate stack frame
    sw   ra, 4(sp)          # save ra
    sw   s0, 0(sp)          # save old fp
    addi s0, sp, 8          # set fp

    ...
    jal proc2       # call another
    ...

    lw   s0, 0(sp)          # restore old fp
    lw   ra, 4(sp)          # restore ra
    addi sp, sp, 8          # deallocate stack frame
    jr   ra                 # return
```

Ex4: leaf proc that saves regs

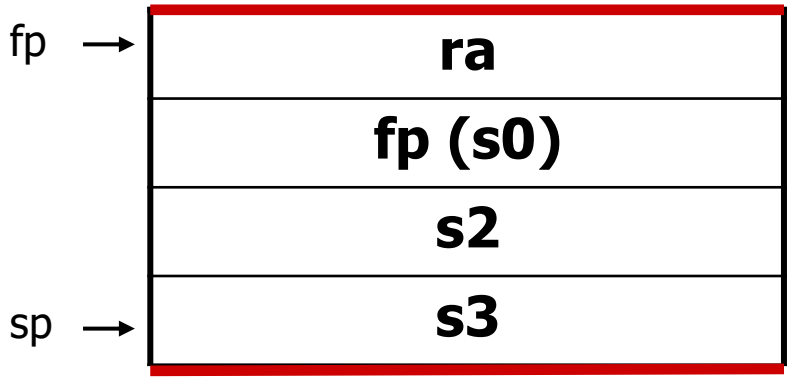
- proc1:
  - is leaf
  - needs to save/restore s2-s3
  - creates more of a stack frame




```
proc1:
    addi sp, sp, -8         # room for s2-s3
    sw   s2, 4(sp)          # save s2
    sw   s3, 0(sp)          # save s3

    ...                     # do the task

    lw   s2, 4(sp)          # restore s2
    lw   s3, 0(sp)          # restore s3
    addi sp, sp, 8          # deallocate stack frame
    jr   ra                 # return
```

Ex4: more general non-leaf proc

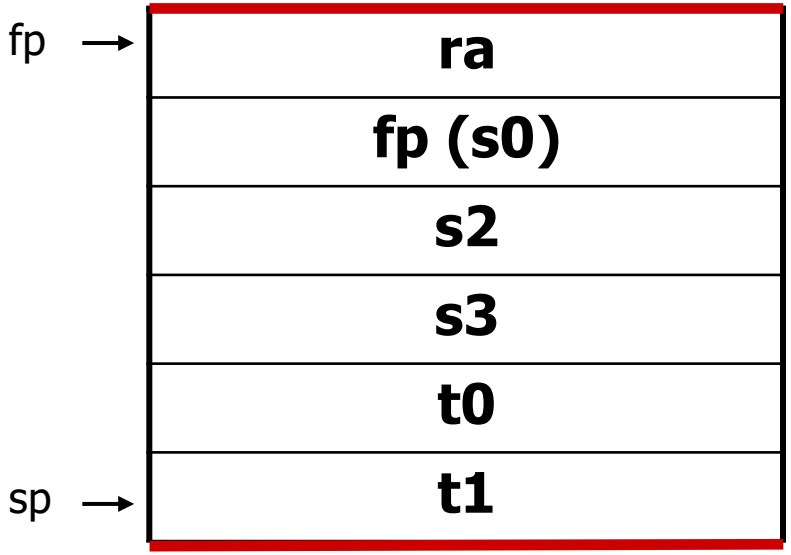
- proc1:
  - calls proc2
  - needs to save/restore s2-s3
  - creates more of a stack frame




```
proc1:
    addi sp, sp, -8         # room for s2-s3
    sw   s2, 4(sp)          # save s2
    sw   s3, 0(sp)          # save s3

    ...
    jal proc2  # call another FN
    ...
    lw   s2, 4(sp)          # restore s2
    lw   s3, 0(sp)          # restore s3
    addi sp, sp, 8          # deallocate stack frame
    jr   ra                 # return
```

Ex5: proc that needs to protect temporaries

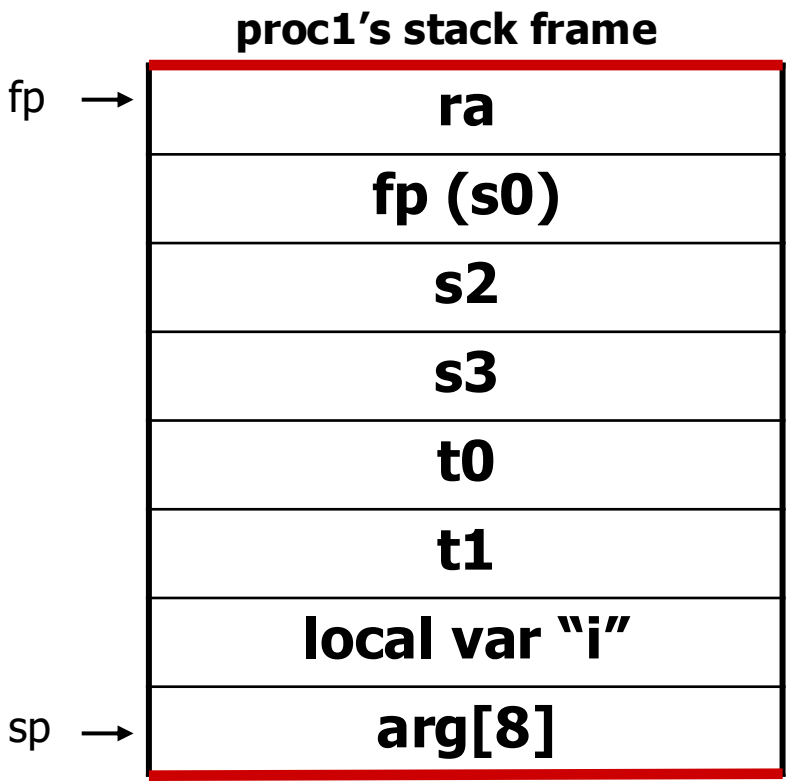
- proc1:
  - calls proc2
  - needs to protect $t0-$t1 from proc2




```
proc1:
    addi sp, sp, -16        # room for ra, s0/fp, t0, t1
    sw   ra, 12(sp)         # save ra because proc1 calls proc2
    sw   s0, 8(sp)          # save old fp
    sw   t0, 4(sp)          # protect t0 from proc2
    sw   t1, 0(sp)          # protect t1 from proc2
    addi s0, sp, 16         # set fp

    ...                     # use t0 and t1

    jal  ra, proc2          # call proc2; may overwrite t0/t1

    lw   t1, 0(sp)          # restore t1
    lw   t0, 4(sp)          # restore t0

    ...                     # continue using t0 and t1

    lw   s0, 8(sp)          # restore old fp
    lw   ra, 12(sp)         # restore ra
    addi sp, sp, 16         # deallocate stack frame
    jr   ra                 # return
```

Ex6: call proc2 with more than 8 args

- proc1:
  - calls proc2
  - … with more than 4 args

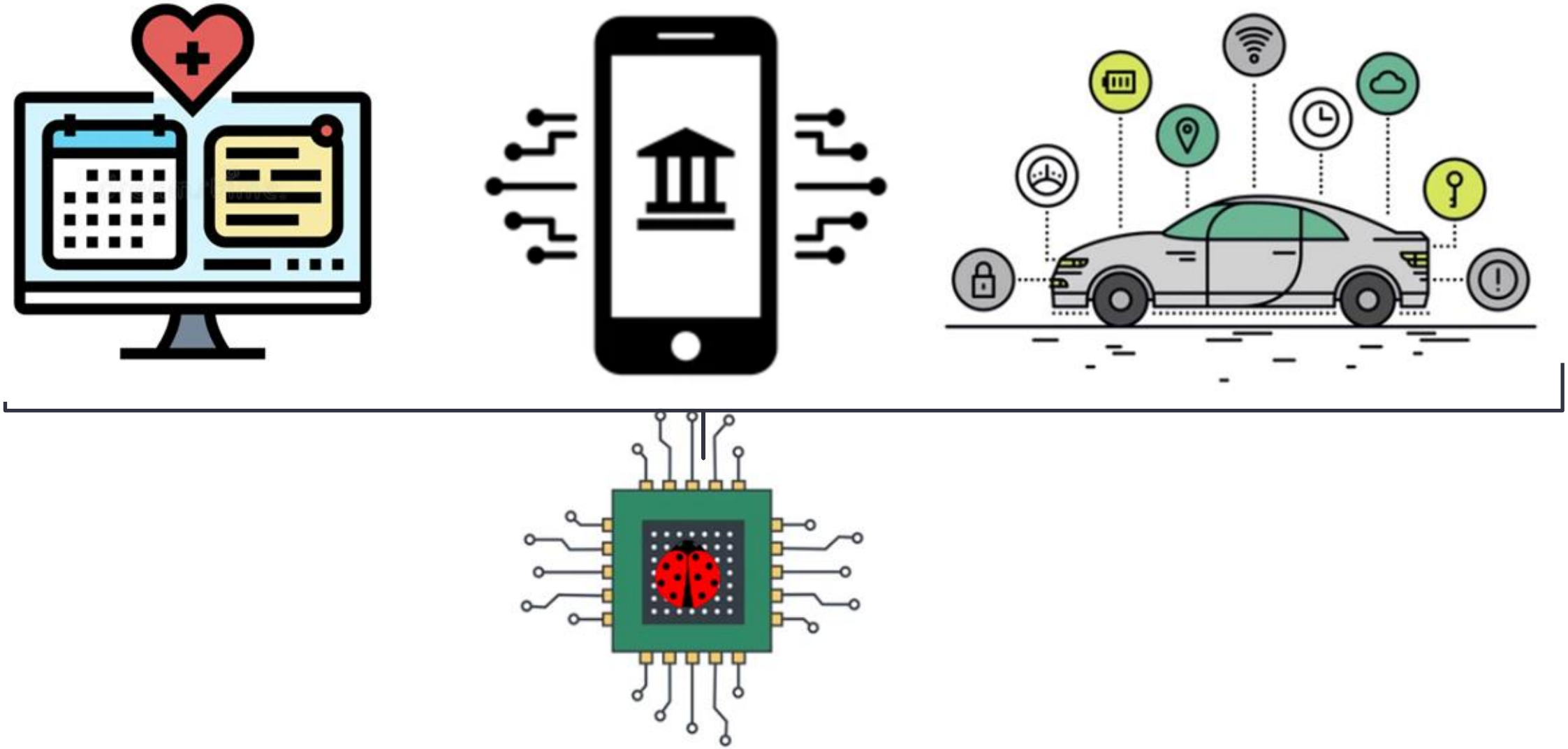



```
proc1:
    addi sp, sp, -20        # room for ra, s0/fp, t0, t1,
arg8
    sw   ra, 16(sp)         # save ra
    sw   s0, 12(sp)         # save old fp
    sw   t0, 8(sp)          # protect t0
    sw   t1, 4(sp)          # protect t1
    li   t2, 80
    sw   t2, 0(sp)          # 9th argument: arg8
    addi s0, sp, 20         # set fp

    li   a0, 0              # arg0
    li   a1, 10             # arg1
    li   a2, 20             # arg2
    li   a3, 30             # arg3
    li   a4, 40             # arg4
    li   a5, 50             # arg5
    li   a6, 60             # arg6
    li   a7, 70             # arg7

    jal  ra, proc2          # call proc2

    lw   t1, 4(sp)          # restore t1
    lw   t0, 8(sp)          # restore t0
    lw   s0, 12(sp)         # restore old fp
    lw   ra, 16(sp)         # restore ra
    addi sp, sp, 20         # deallocate stack frame
    jr   ra                 # return
```

HARDWARE SECURITY

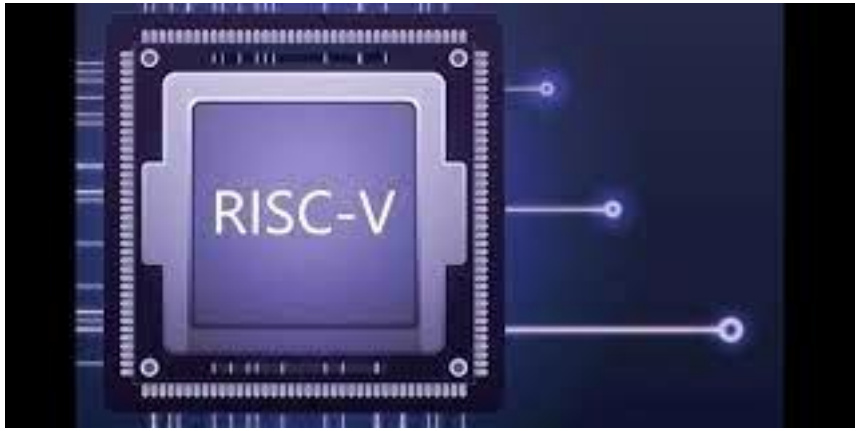
Our lives depend on secure systems!

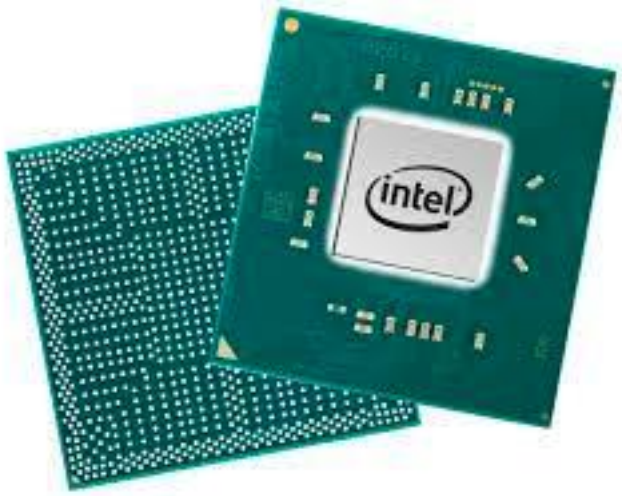
Hardware is the root of trust in all systems!

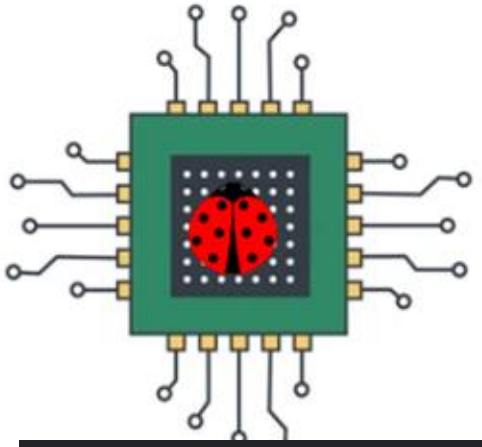


Processor Security

- The **processor** is part of the **Trusted Computing Base (TCB)**, which comprises all the security critical software, hardware and firmware in a computer system
- As we've spent the semester learning, modern processors are optimized for **performance**
  - ***Security is often secondary!***







Hardware Backdoor Discovered in RFID Cards Used in Hotels and Offices Worldwide

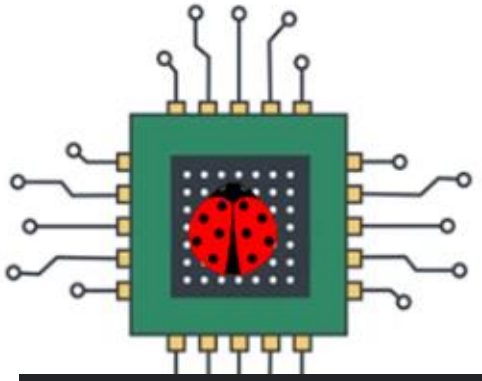
Aug 22, 2024 Ravie Lakshmanan

Unpatchable vulnerability in Apple chip leaks secret encryption keys

Fixing newly discovered side channel will likely take a major toll on performance.

Hardware vulnerabilities in Hitachi Energy, ABB, B&R ICS devices pose critical infrastructure threat

APRIL 07, 2025



Hardware Backdoor Discovered in RFID Cards Used in Hotels and Offices Worldwide

Aug 22, 2024 Ravie Lakshmanan

Unpatchable vulnerability in Apple chip leaks secret encryption keys

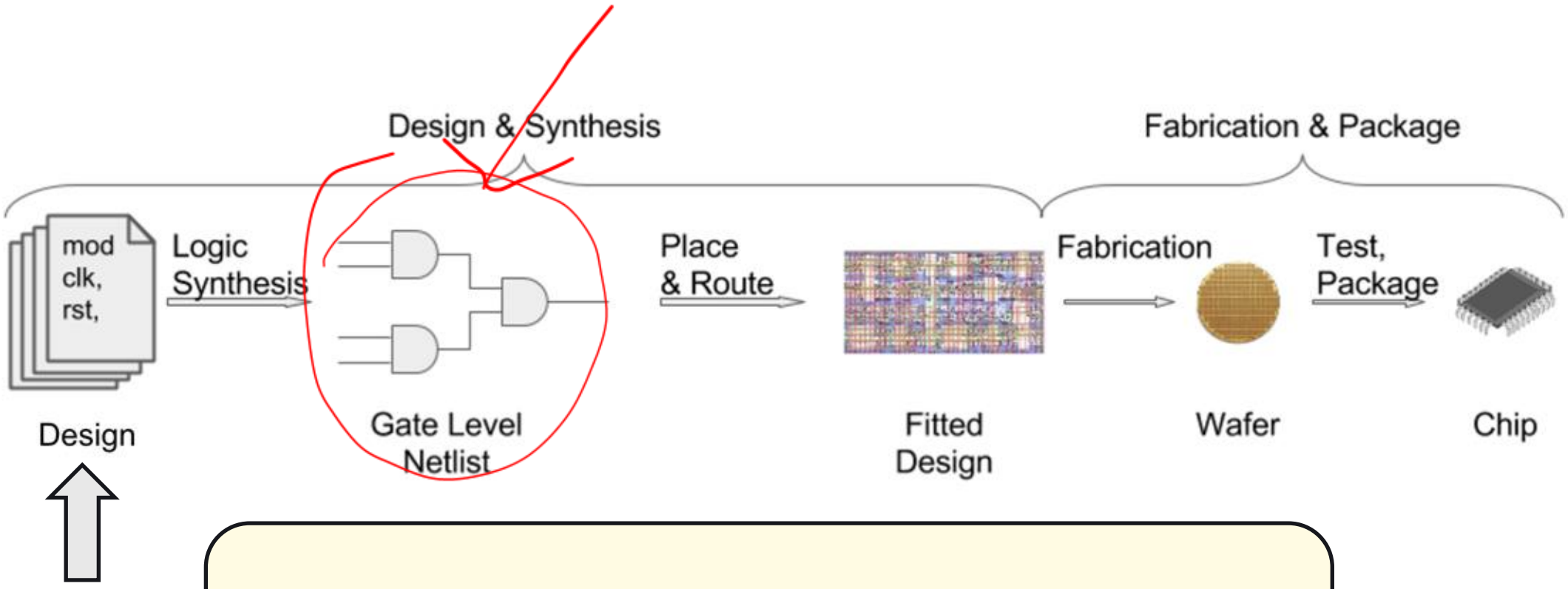
Fixing newly discove

High-assurance verification is critical!

Hardware vulnerabilities in Hitachi Energy, ABB, B&R ICS devices pose critical infrastructure threat

APRIL 07, 2025

The Problem(s)...



Hardware bugs are hard to patch post-deployment!

RTL
↳ Register transfer level

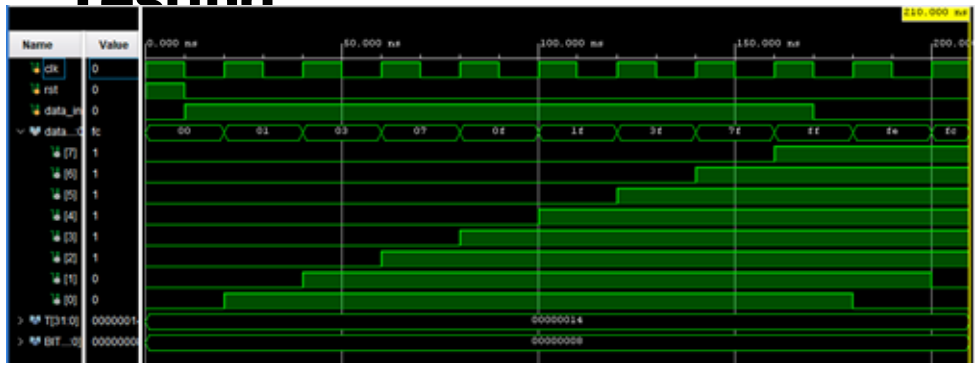
The Problem(s)...

Increasing design complexity!

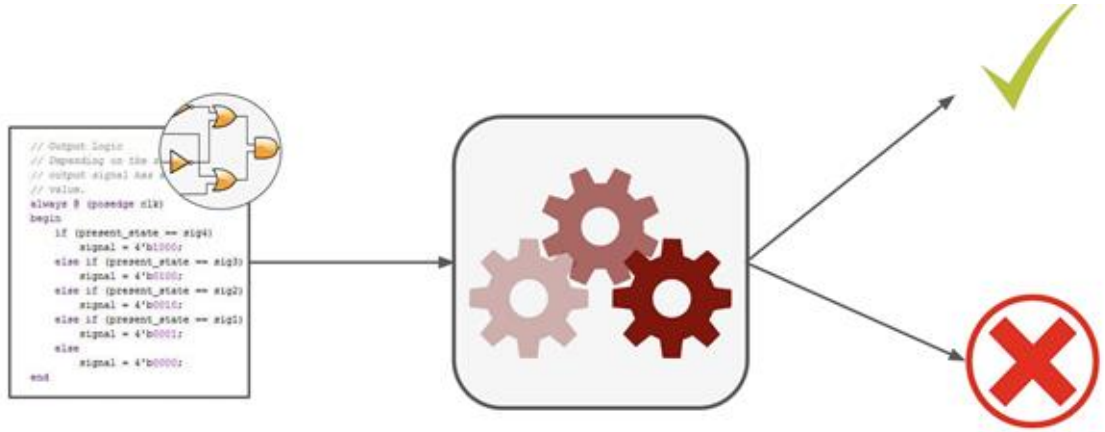
Number of Transistors in CPUs over Time

Current State of the Art in Hardware Verification

Simulation Based Testing



Model Checking



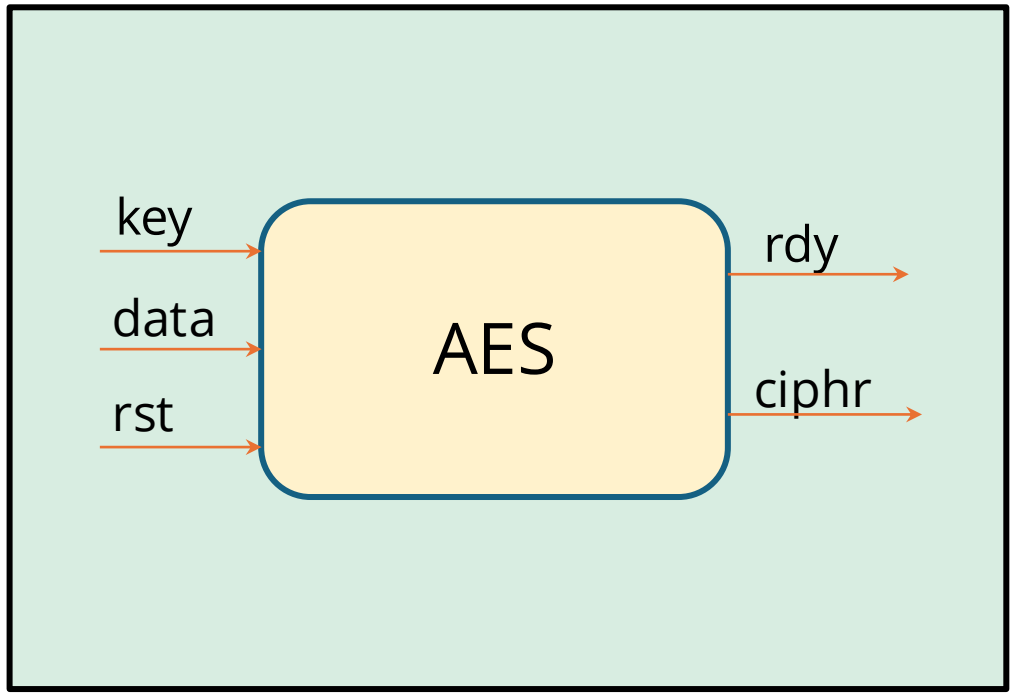
Quick Background: Information Flow

```
1  input  A
2  output E

3  reg B, C, D;

4  assign E = D;

5  if (A)
6    D <= B;
7  else
8    D <= C;
```

Line 6 shows an **explicit flow** from B to D.

Lines 5-6 shows an **implicit flow** from A to D.

Information Flow Property Verification



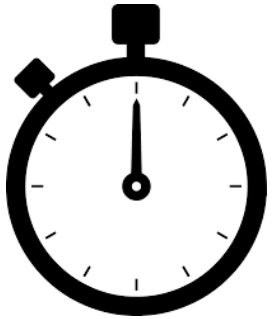
φ: key $\not\rightarrow$ rdy

Information Flow Property

Beyond Program Logic: Side Channels

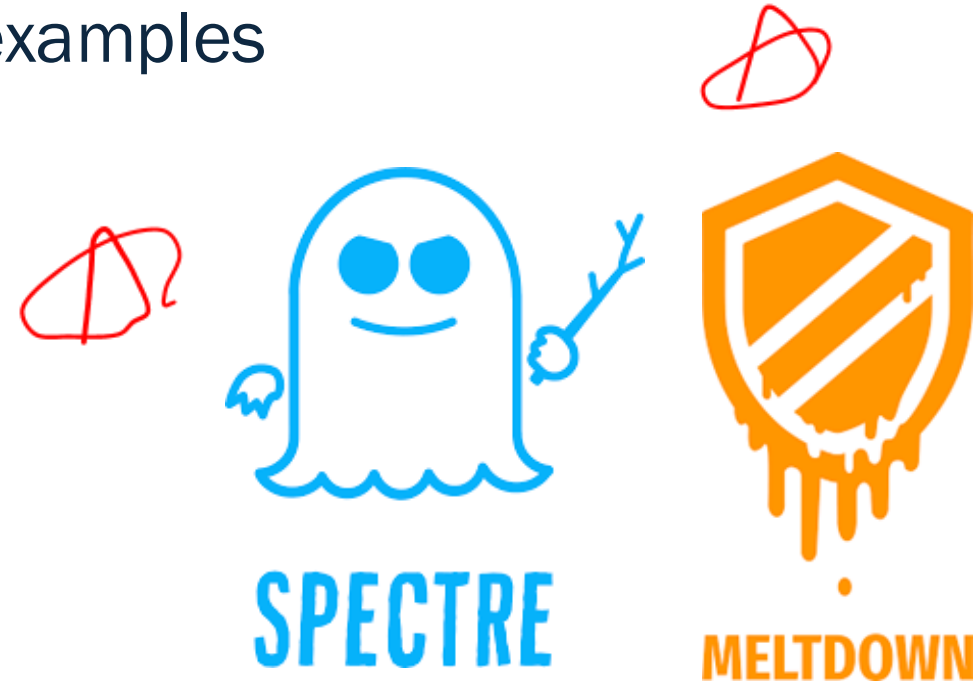
Information leaks through **how a program executes**
not just what it computes

Attacker observes:
- Timing
- Memory/cache behavior
- Control flow/speculation
- Power Usage, too!




Side Channel Attacks

- **Side-Channel Attack**: Exploit generated by collecting information during program execution
  - *Ex: Memory access patterns, cache state, branch predictor state*
- Spectre and Meltdown are two recent examples



Example: Spectre

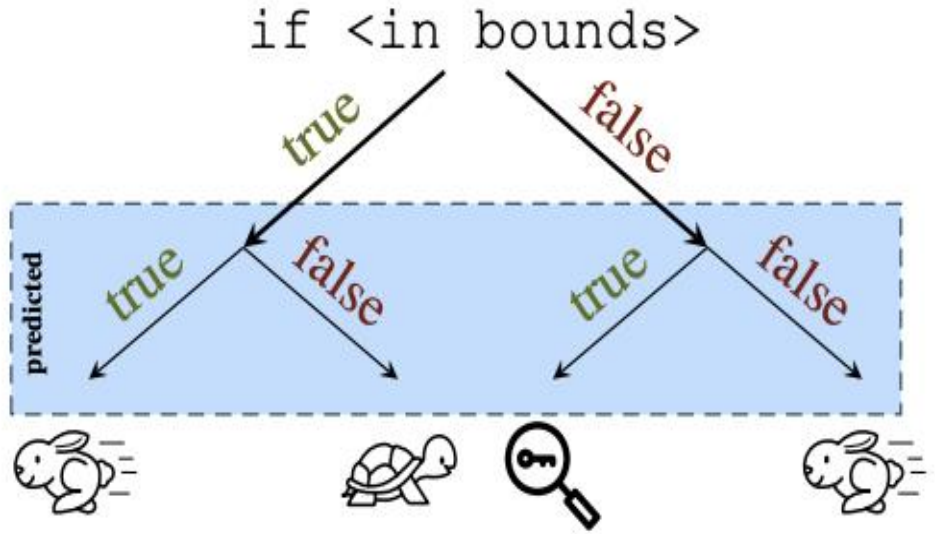
```
if (x < array1_size):
    y = array2[array1[x] * 4096]]
```

Key steps in a Spectre Attack:

1. Mistrain branch predictor to predict a path that will leak info over a covert channel
2. Invoke code with malicious inputs
3. Extract information from covert channel (AKA the cache!!)

Mistrain branch predictor to assume x is in bounds (invoke code many times with valid x inputs)

Call with x = (addr of some secret) - (array1 base)



Some Specific Security Goals

1. Code and memory specific to an application are **isolated** from the rest of the system

2. When running this isolated code, having a means of **attesting** to a local or remote system what code is running **without exposing any sensitive information**

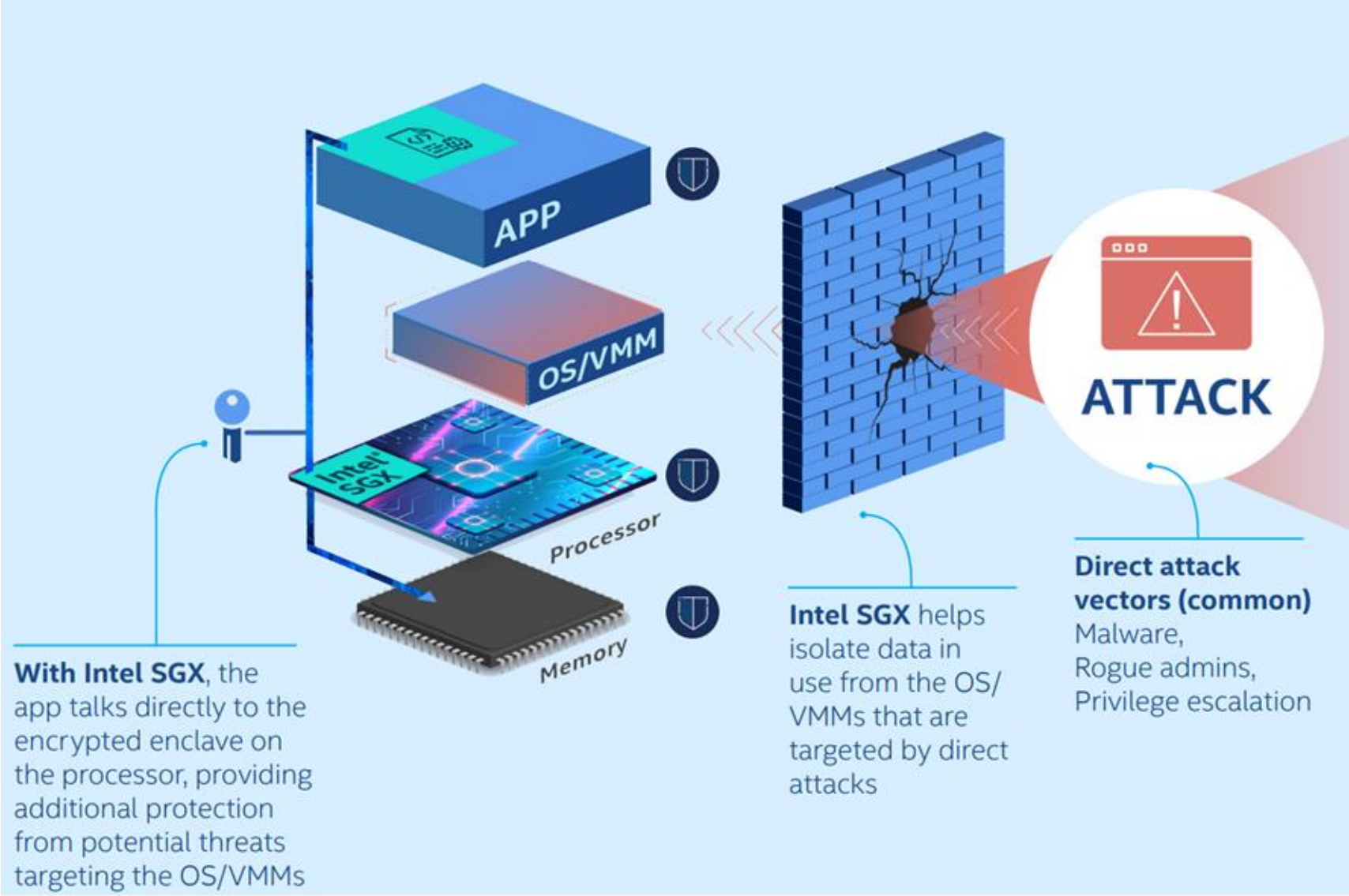
3. Minimizing the size of the **TCB** so that only the processor has to be trusted.

Intel came up with a solution that meets all of these…

Intel SGX

- **Intel Software Guard Extension** (SGX) is hardware added to Intel processors designed to protect sensitive data from malicious actors, specifically from software with a higher privilege level or when a malicious actor has control over the system
- Furthermore, SGX is meant to be non-disruptive for legitimate systems while also providing a secure way to scale applications to unknown remote servers

High Level Visual



SGX Overview

- 17 new instructions that allow developers to generate hardware enforceable, secure containers, called **enclaves**
- Enclaves are protected areas in the application's address space
- SGX allows the protected application data to be distributed in the clear

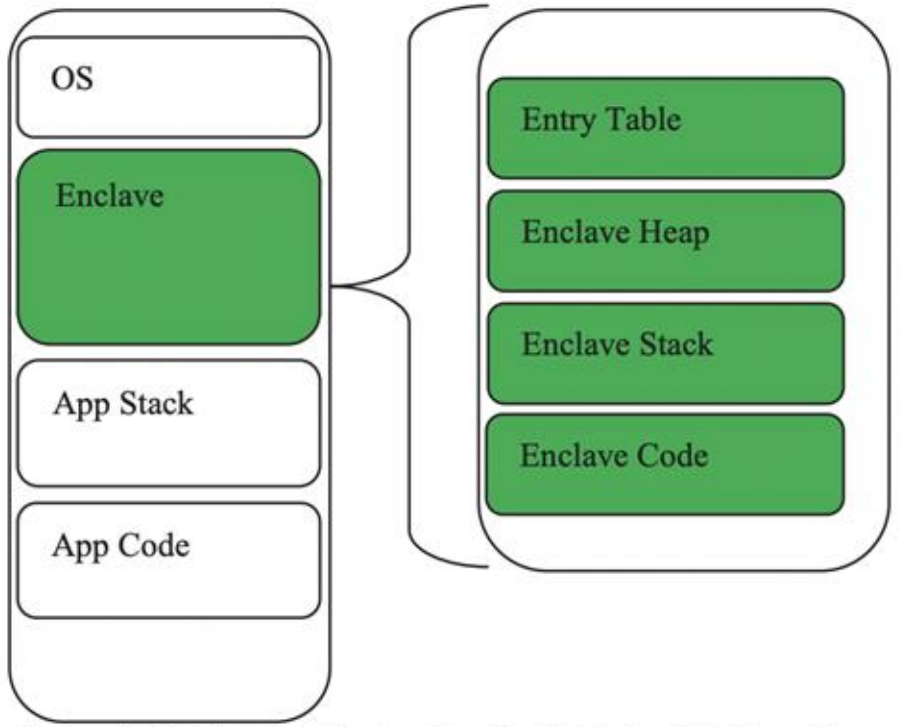
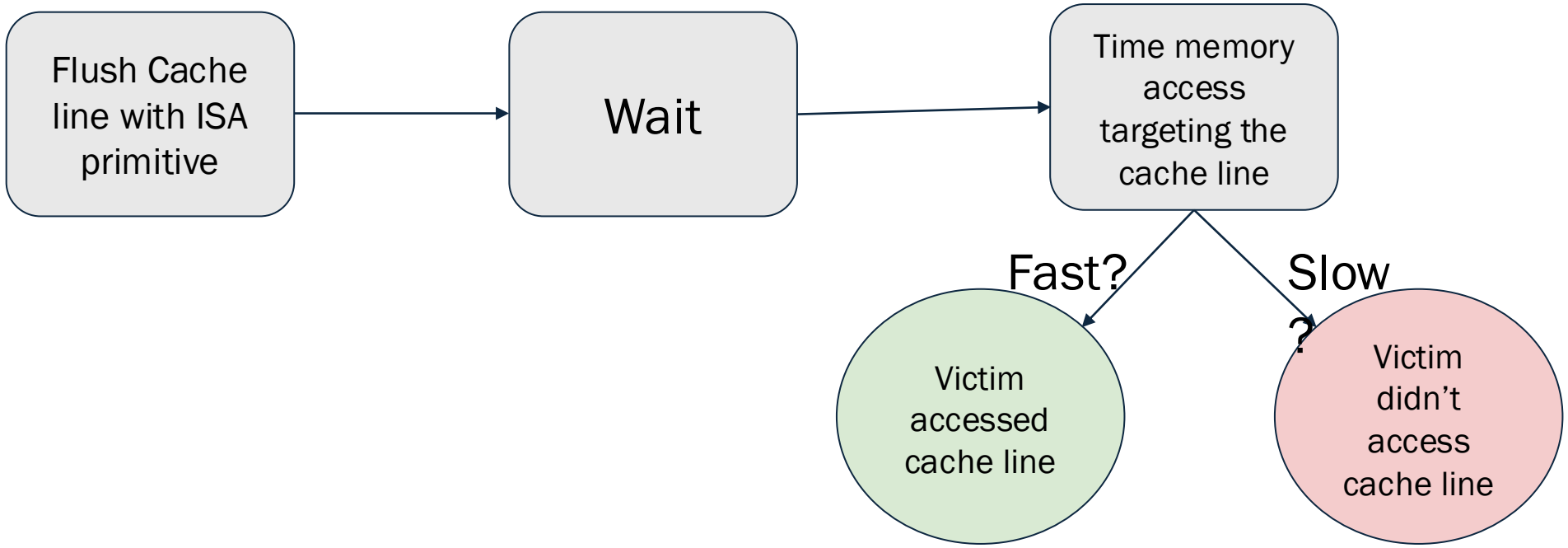


Figure 1: Enclave within Application's Virtual Address Space

Example: Flush and Reload Mechanics

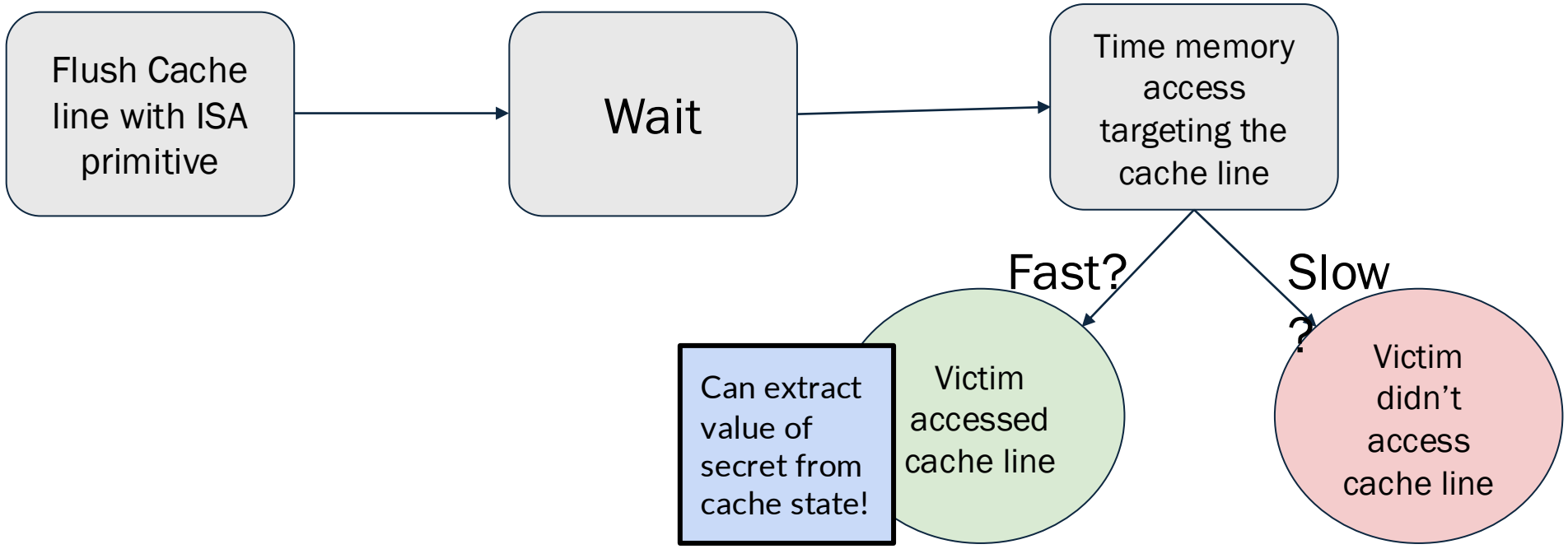
- In the threat model for SGX, the **attacker controls the OS → enclave data can be leaked.** How? Through a side-channel like a cache.



Flush & Reload Mechanics

```
if (x < array1_size):
    y = array2[array1[x] * 4096]]
```

- In the threat model for SGX, the **attacker controls the OS → enclave data can be leaked.** How? Through a side-channel like a cache.



If you enjoyed this class…

- **Yay! Me too! Thank you all for taking it with me!**
- **Consider applying to be an LA: https://csxl.unc.edu/article/apply-to-be-a-fall-2026-uta-gta**
- Consider taking…
  - COMP541! (Digital Logic & Design!)
    - Build a single-cycle CPU using a hardware description language (Verilog)
    - End of course project is programming it in assembly!
  - COMP520 (Compilers!)
    - Learn about the SW programs that take high-level code to machine code
    - Implement a compiler for a subset of C. Compiling to RISC-V!
  - COMP530 (Operating Systems!)
    - Learn more about the memory hierarchy, computer performance & the HW/SW interface
- Consider joining the undergrad systems reading group!
  - Info is on my website, or you can email me!