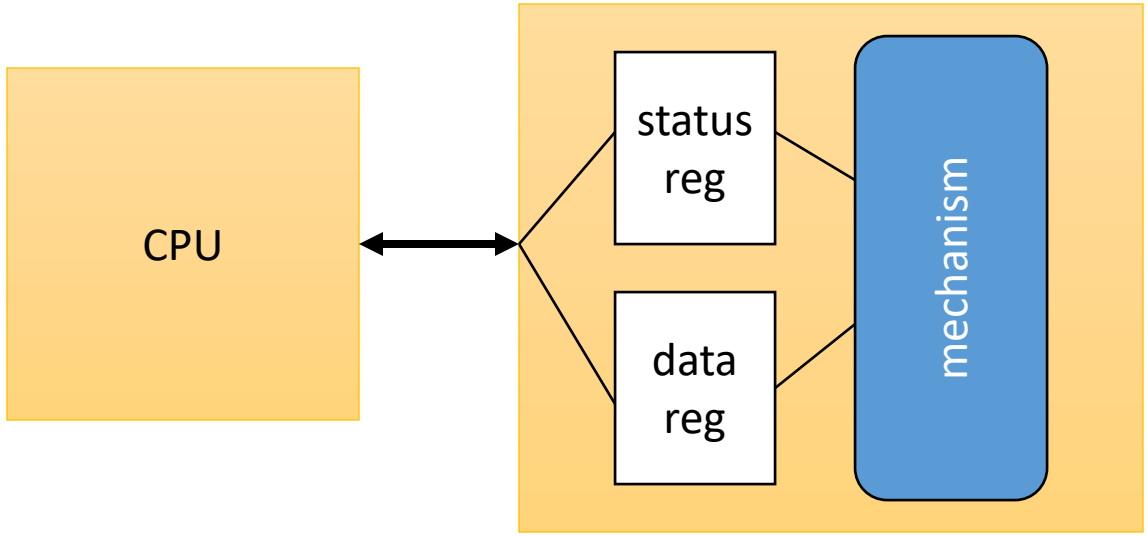
CPUs

- Input and output.
- Supervisor mode, exceptions, traps.
- Co-processors.

I/O devices

- Usually includes some non-digital component.
- Typical digital interface to CPU:

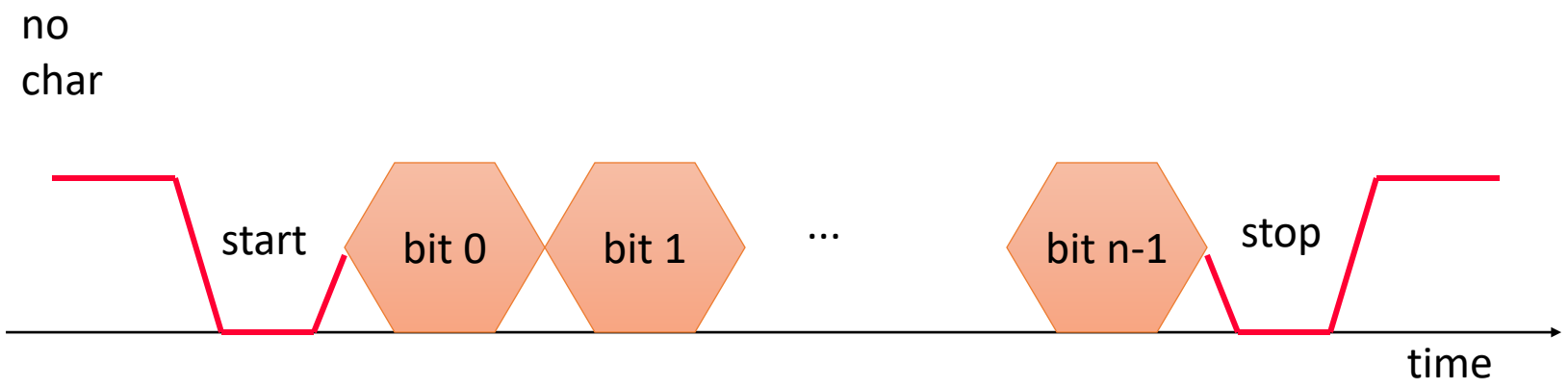


Application: 8251 UART

- Universal asynchronous receiver transmitter (UART) : provides serial communication.
- 8251 functions are integrated into standard PC interface chip.
- Allows many communication parameters to be programmed.

Serial communication

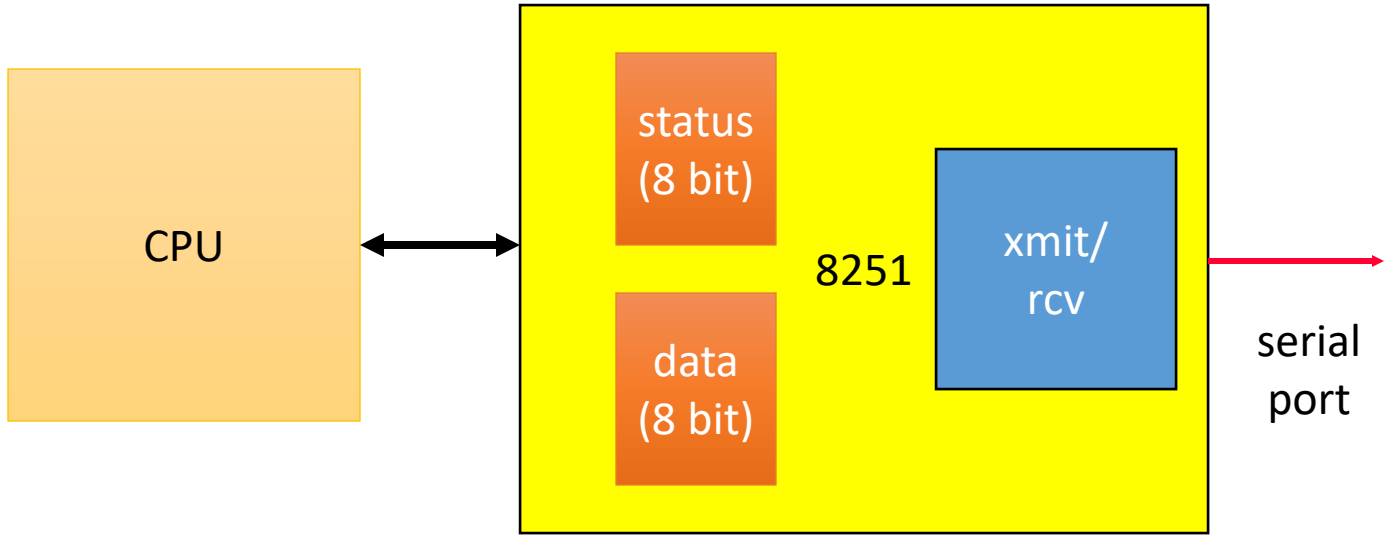
- Characters are transmitted separately:



Serial communication parameters

- Baud (bit) rate.
- Number of bits per character.
- Parity/no parity.
- Even/odd parity.
- Length of stop bit (1, 1.5, 2 bits).

8251 CPU interface



Serial port mode bits

- mode[1:0]: baud rate
  - 00: synchronous
  - 01: async, no clock prescaler
  - 10: async, 16x prescaler
  - 11: async, 64x prescaler
- mode[3:2]: bits/char
  - 00: 5 bits
  - 01: 6 bits
  - 10: 7 bits
  - 11: 8 bits.
- mode[5:4]: parity
  - 00, 10: no parity
  - 01: odd parity
  - 11: even parity
- mode[7:6]: stop bit
  - 00: invalid
  - 01: 1 stop bit
  - 10: 1.5 stop bits
  - 11: 2 stop bits

Serial port command register

- mode[0]: transmit enable
- mode[1]: set nDTR output
- mode[2]: enable receiver
- mode[3]: send break
- mode[4]: reset error flags
- mode[5]: set nRTS
- mode[6]: internal reset
- mode[7]: hunt mode

Serial port status register

- status[0]: transmitter ready
- status[1]: receiver ready
- status[2]: transmission complete
- status[3]: parity
- status[4]: overrun
- status[5]: frame error
- status[6]: sync char deleted
- status[7]: nDSR value

Programming I/O

- Two types of instructions can support I/O:
  - special-purpose I/O instructions;
  - memory-mapped load/store instructions.
- Intel x86 provides `in, out` instructions. Most other CPUs use memory-mapped I/O.
- I/O instructions do not preclude memory-mapped I/O.

ARM memory-mapped I/O

- Define location for device:

```
DEV1 EQU 0x1000
```

- Read/write code:

```
LDR r1,#DEV1 ; set up device adrs
LDR r0,[r1] ; read DEV1
LDR r0,#8 ; set up value to write
STR r0,[r1] ; write value to device
```

SHARC memory mapped I/O

- Device must be in external memory space (above 0x400000).
- Use DM to control access:

```
I0 = 0x400000;
M0 = 0;
R1 = DM(I0,M0);
```

Peek and poke

- Traditional HLL interfaces:

```
int peek(char *location) {
  return *location; }

void poke(char *location, char newval) {
  (*location) = newval; }
```

Busy/wait output

- Simplest way to program device.
  - Use instructions to test when device is ready.

```
current_char = mystring;
while (*current_char != ‘\0’) {
  poke(OUT_CHAR,*current_char);
  while (peek(OUT_STATUS) != 0);
  current_char++;
}
```

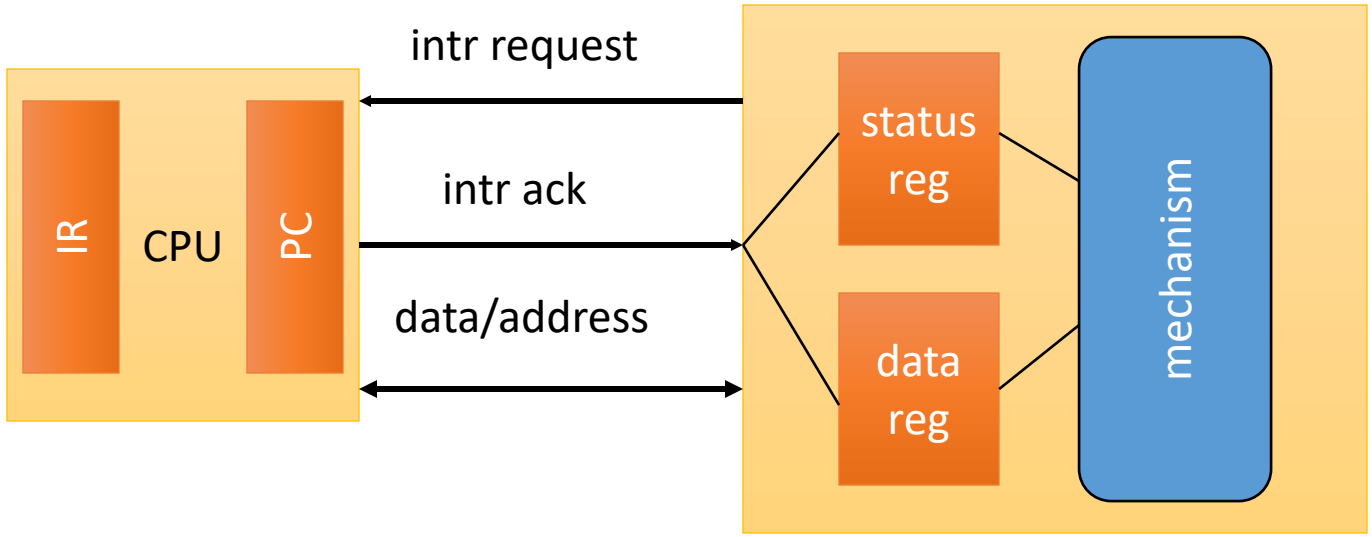
Simultaneous busy/wait input and output

```
while (TRUE) {
  /* read */
  while (peek(IN_STATUS) == 0);
  achar = (char)peek(IN_DATA);
  /* write */
  poke(OUT_DATA,achar);
  poke(OUT_STATUS,1);
  while (peek(OUT_STATUS) != 0);
  }
```

Interrupt I/O

- Busy/wait is very inefficient.
  - CPU can’t do other work while testing device.
  - Hard to do simultaneous I/O.
- Interrupts allow a device to change the flow of control in the CPU.
  - Causes subroutine call to handle device.

Interrupt interface



Interrupt behavior

- Based on subroutine call mechanism.
- Interrupt forces next instruction to be a subroutine call to a predetermined location.
  - Return address is saved to resume executing foreground program.

Interrupt physical interface

- CPU and device are connected by CPU bus.
- CPU and device handshake:
  - device asserts interrupt request;
  - CPU asserts interrupt acknowledge when it can handle the interrupt.

Example: character I/O handlers

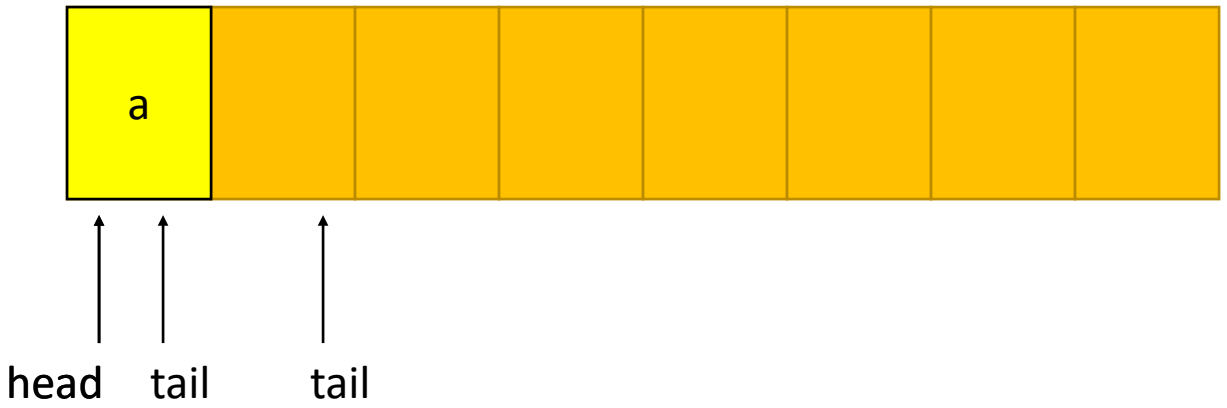
```
void input_handler() {
  achar = peek(IN_DATA);
  gotchar = TRUE;
  poke(IN_STATUS,0);
}
void output_handler() {
}
```

Example: interrupt-driven main program

```
main() {
 while (TRUE) {
     if (gotchar) {
         poke(OUT_DATA,achar);
         poke(OUT_STATUS,1);
         gotchar = FALSE;
         }
     }
 }
```

Example: interrupt I/O with buffers

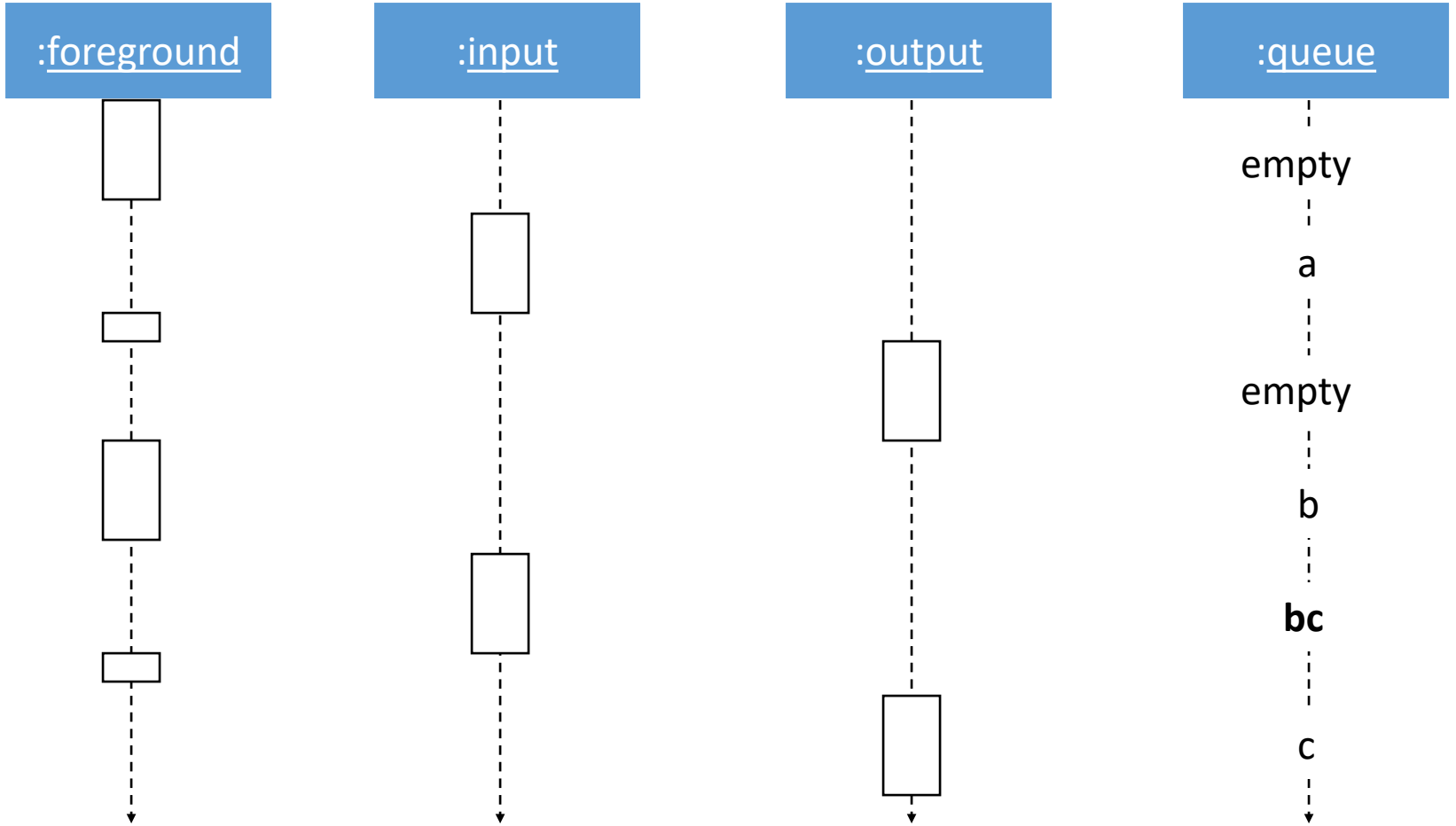
- Queue for characters:



Buffer-based input handler

```
void input_handler() {
  char achar;
  if (full_buffer()) error = 1;
  else { achar = peek(IN_DATA); add_char(achar); }
  poke(IN_STATUS,0);
  if (nchars == 1)
      { poke(OUT_DATA,remove_char()); poke(OUT_STATUS,1); }
  }
```

I/O sequence diagram



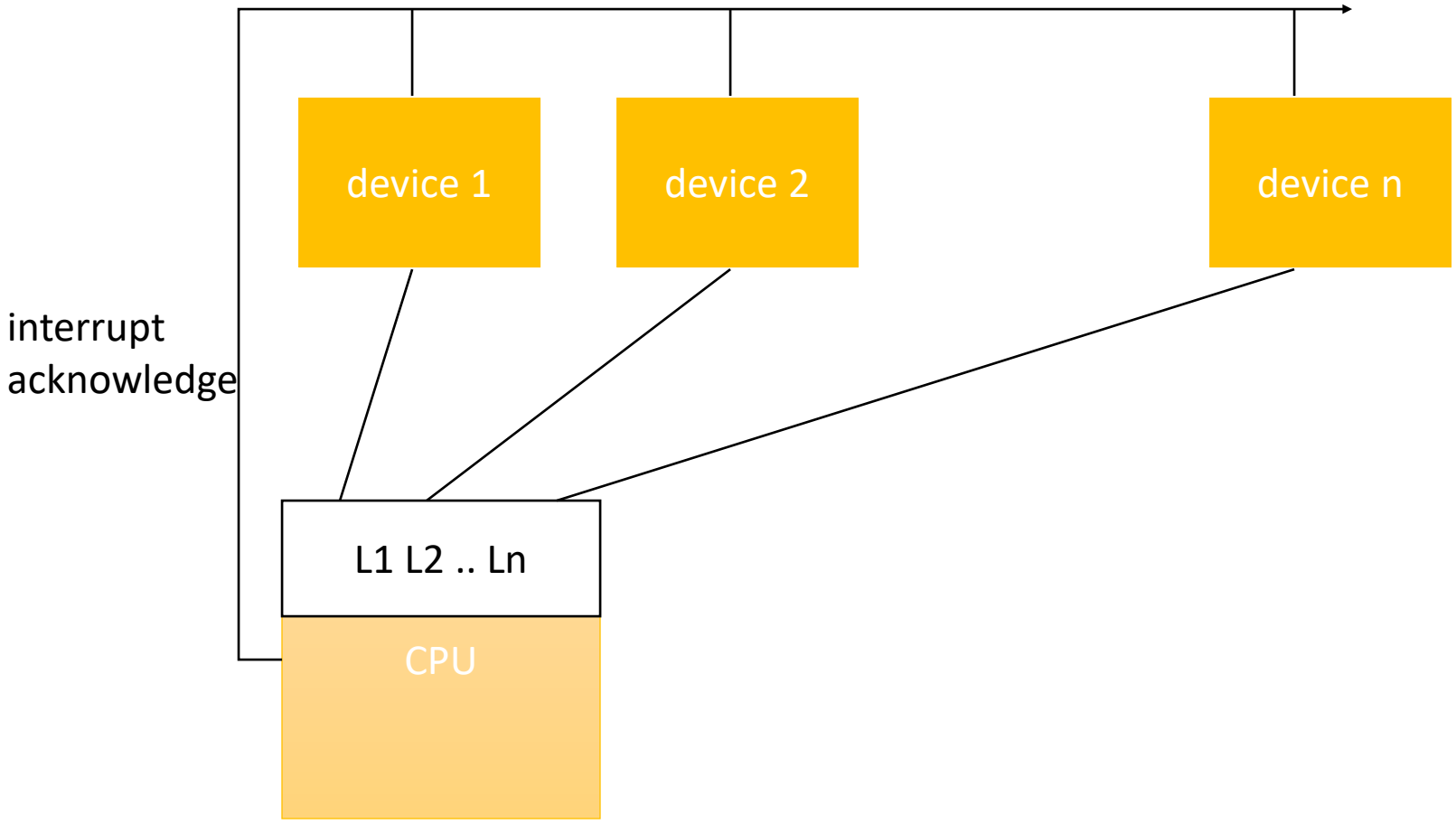
Debugging interrupt code

- What if you forget to change registers?
  - Foreground program can exhibit mysterious bugs.
  - Bugs will be hard to repeat---depend on interrupt timing.

Priorities and vectors

- Two mechanisms allow us to make interrupts more specific:
  - Priorities determine what interrupt gets CPU first.
  - Vectors determine what code is called for each type of interrupt.
- Mechanisms are orthogonal: most CPUs provide both.

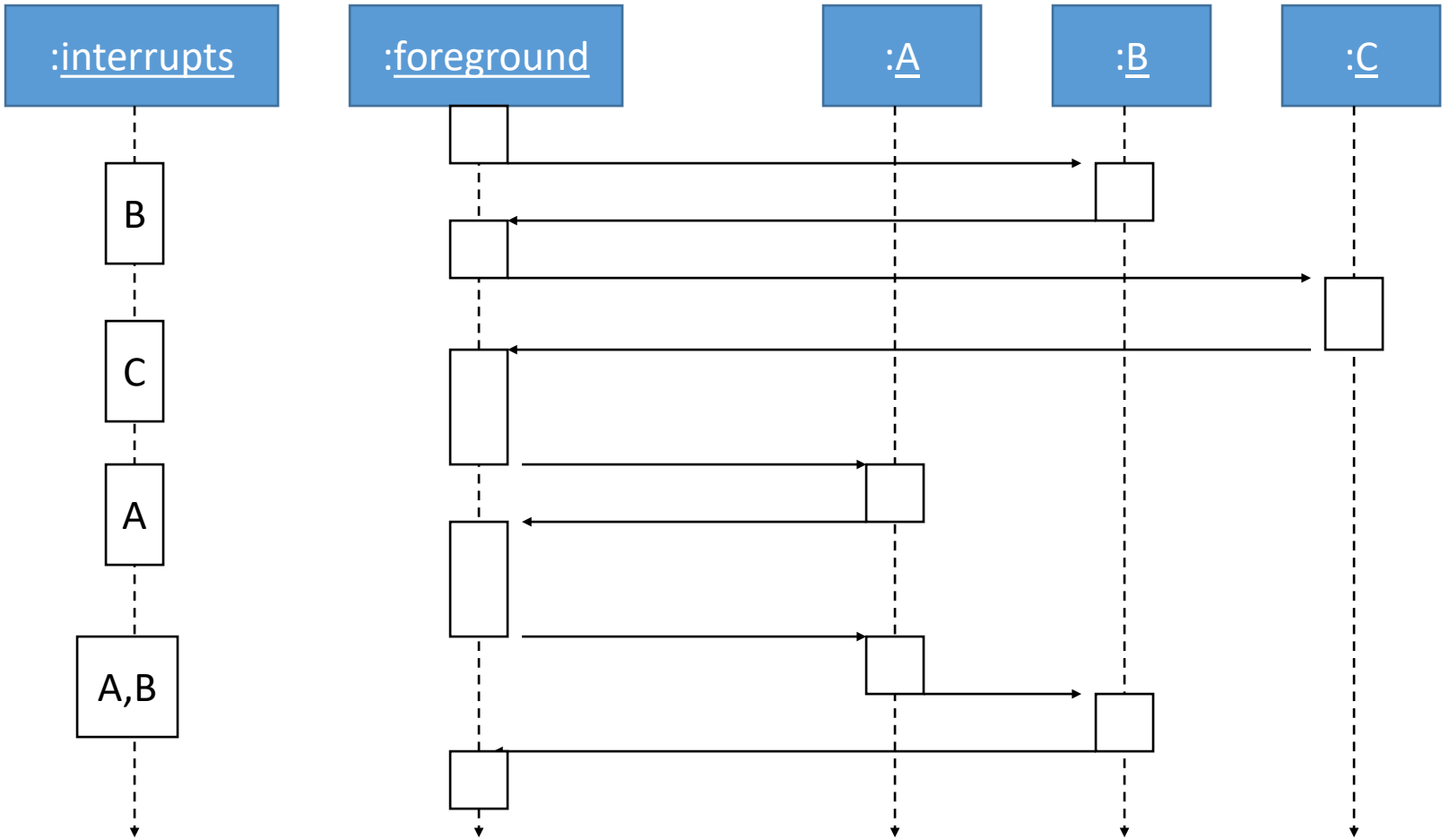
Prioritized interrupts



Interrupt prioritization

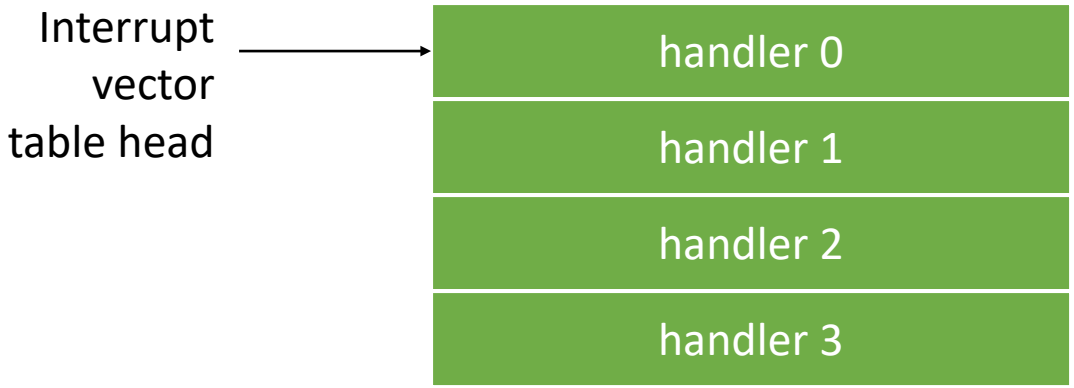
- Masking: interrupt with priority lower than current priority is not recognized until pending interrupt is complete.
- Non-maskable interrupt (NMI): highest-priority, never masked.
  - Often used for power-down.

Example: Prioritized I/O

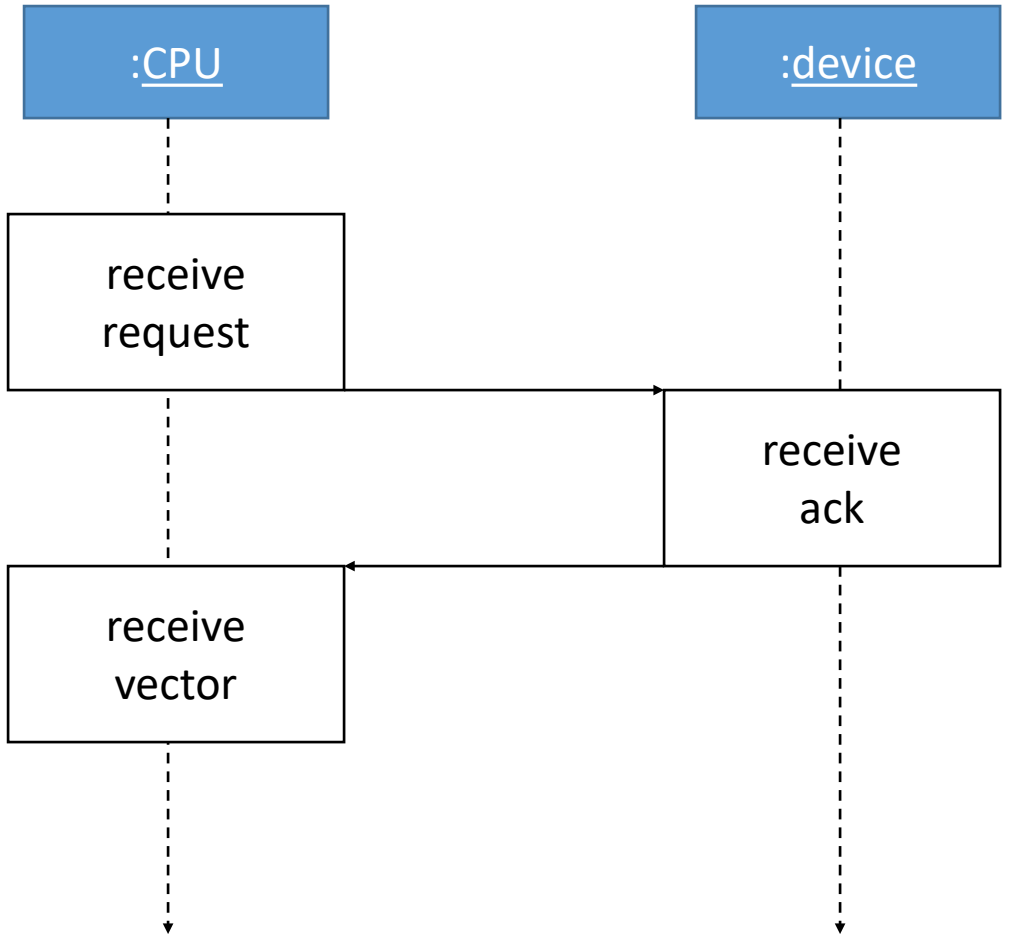


Interrupt vectors

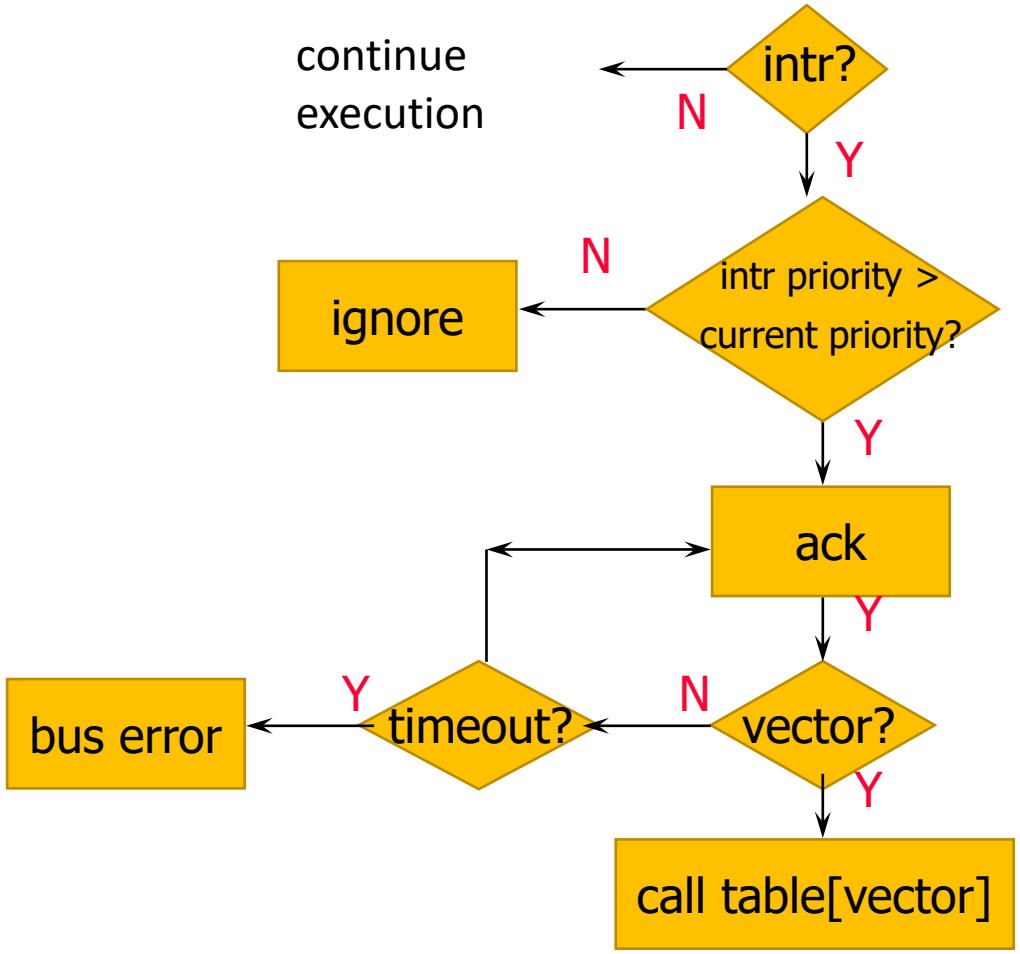
- Allow different devices to be handled by different code.
- Interrupt vector table:



Interrupt vector acquisition



Generic interrupt mechanism



Assume priority selection is handled before this point.

Interrupt sequence

- CPU acknowledges request.
- Device sends vector.
- CPU calls handler.
- Software processes request.
- CPU restores state to foreground program.

Sources of interrupt overhead

- Handler execution time.
- Interrupt mechanism overhead.
- Register save/restore.
- Pipeline-related penalties.
- Cache-related penalties.

ARM interrupts

- ARM7 supports two types of interrupts:
  - Fast interrupt requests (FIQs).
  - Interrupt requests (IRQs).
- Interrupt table starts at location 0.

ARM interrupt procedure

- CPU actions:
  - Save PC. Copy CPSR to SPSR.
  - Force bits in CPSR to record interrupt.
  - Force PC to vector.
- Handler responsibilities:
  - Restore proper PC.
  - Restore CPSR from SPSR.
  - Clear interrupt disable flags.

ARM interrupt latency

- Worst-case latency to respond to interrupt is 27 cycles:
  - Two cycles to synchronize external request.
  - Up to 20 cycles to complete current instruction.
  - Three cycles for data abort.
  - Two cycles to enter interrupt handling state.

C55x interrupts

- Latency is between 7 and 13 cycles.
- Maskable interrupt sequence:
  - Interrupt flag register is set.
  - Interrupt enable register is checked.
  - Interrupt mask register is checked.
  - Interrupt flag register is cleared.
  - Appropriate registers are saved.
  - INTM set to 1, DBGM set to 1, EALLOW set to 0.
  - Branch to ISR.
- Two styles of return: fast and slow.

PIC16F interrupts

- Synchronous interrupts come from inside CPU.
- Asynchronous interrupts come from outside CPU.
- INTCON register holds interrupt control bits.
  - GIE allows unmasked interrupts.
  - PIE allows peripheral interrupts.
  - TMR0 enables timer 0.
  - INT enables external interrupts.
  - PIR1/PIR2 control peripheral interrupts.
- RETFIE used to return from interrupts.
- Latency:
  - Synchronous: 3Tcy
  - Asynchronous: 3 to 3.75 Tcy

Supervisor mode

- May want to provide protective barriers between programs.
  - Avoid memory corruption.
- Need supervisor mode to manage the various programs.
- SHARC does not have a supervisor mode.

ARM supervisor mode

- Use SWI instruction to enter supervisor mode, similar to subroutine:

```
SWI CODE_1
```

- Sets PC to 0x08.
- Argument to SWI is passed to supervisor mode code.
- Saves CPSR in SPSR.

Exception

- Exception: internally detected error.
- Exceptions are synchronous with instructions but unpredictable.
- Build exception mechanism on top of interrupt mechanism.
- Exceptions are usually prioritized and vectorized.

Trap

- Trap (software interrupt): an exception generated by an instruction.
  - Call supervisor mode.
- ARM uses SWI instruction for traps.
- SHARC offers three levels of software interrupts.
  - Called by setting bits in IRPTL register.

Co-processor

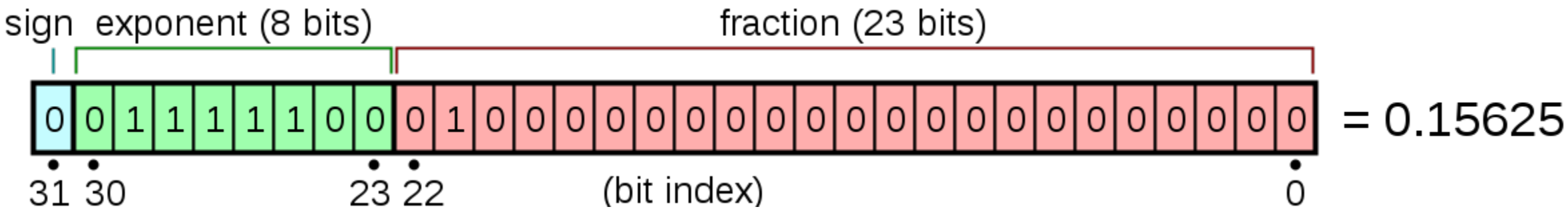
- Co-processor: added function unit that is called by instruction.
  - Floating-point units are often structured as co-processors.
- ARM allows up to 16 designer-selected co-processors.
  - Floating-point co-processor uses units 1, 2.
- C55x uses co-processors as well.

Fixed-point

- 0100.0110 = 4 + 3/8
- 0001.0000 = 1
- 0000.1000 = 1/2
- 0000.0101 = 5/16
- 0000.0100 = 1/4

Floating-point

$$12.345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10^{\overbrace{-3}^{\text{exponent}}}}_{\text{base}}$$

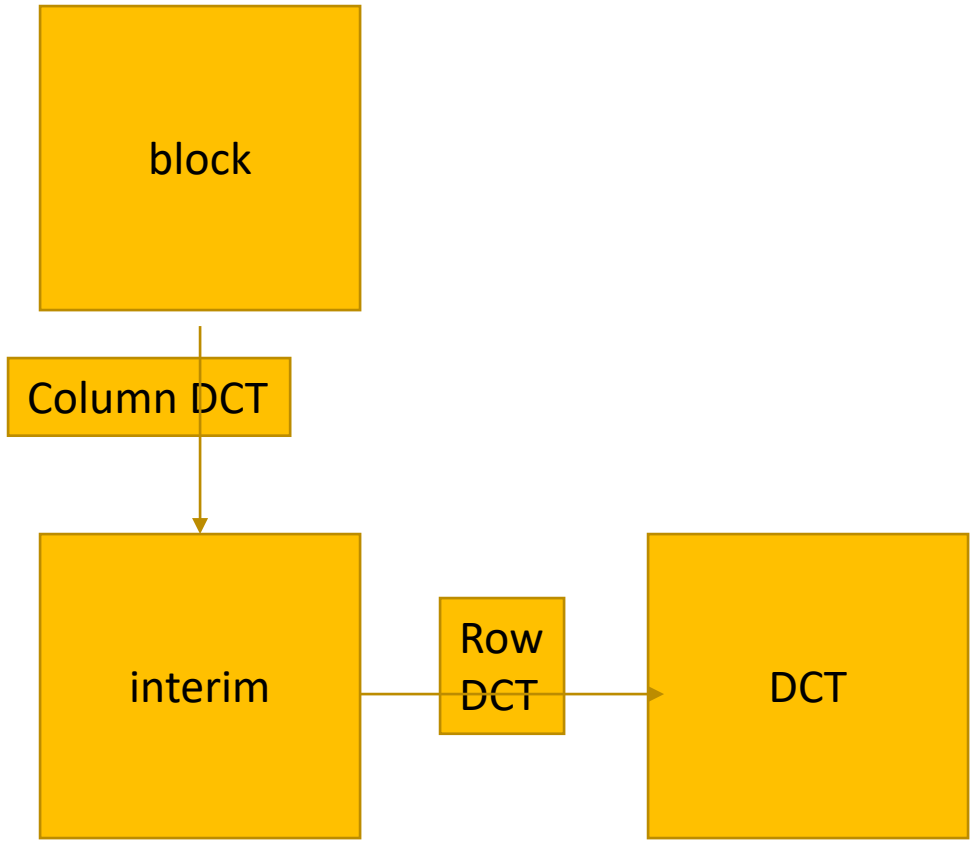


C55x image/video hardware extensions

- Available in 5509 and 5510.
  - Equivalent C-callable functions for other devices.
- Available extensions:
  - DCT/IDCT.
  - Pixel interpolation
  - Motion estimation.

DCT/IDCT

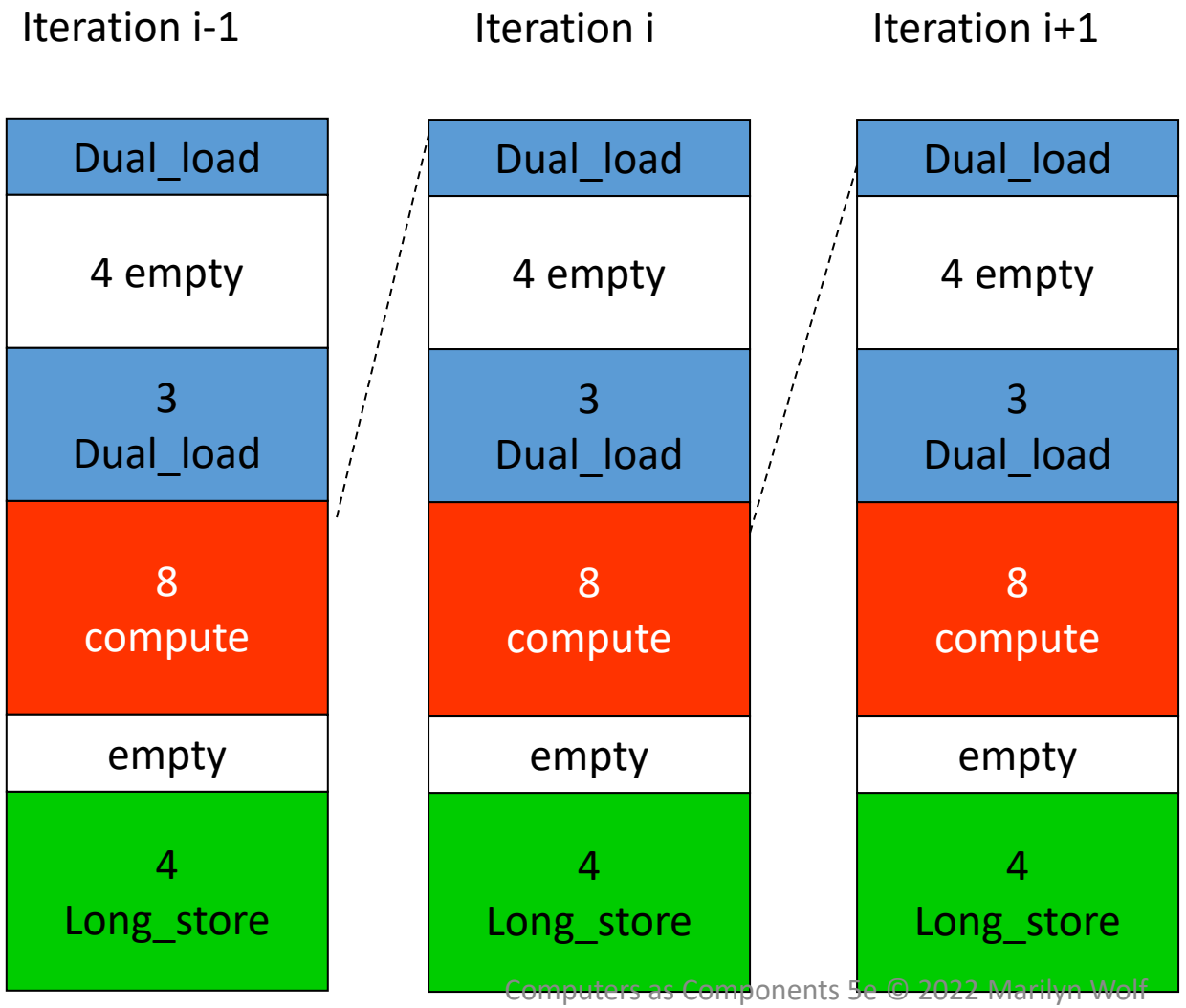
- 2-D DCT/IDCT is computed from two 1-D DCT/IDCT.
  - Put data in different banks to maximize throughput.



C55 DCT/IDCT coprocessor extensions

- Load, compute, transfer to accumulators:
  - ACy=copr(k8,ACx,Xmem,Ymem)
- Compute, transfer, mem write:
  - ACy=copr(k8,ACx,ACy), Lmem=ACz
- Special:
  - ACy=copr(k8,ACx,ACy)

Software pipelined load/compute/store for DCT



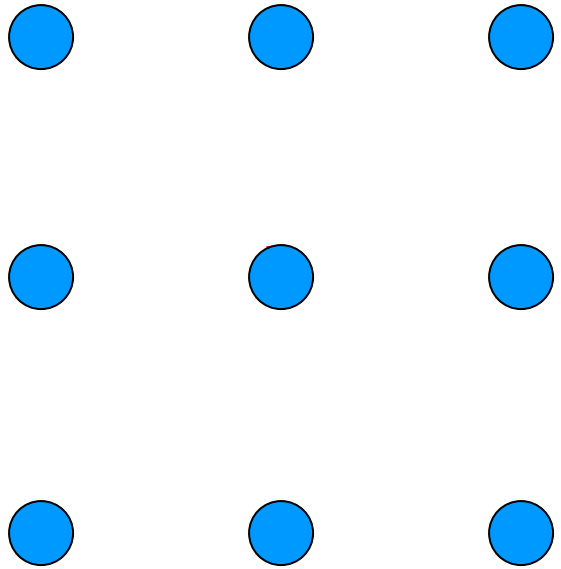
op_i(0), load_i+1(0,1)
op_i(1), store_i-1(0,1)
op_i(2), store_i-1(2,3)
op_i(2), store_i-1(4,5)
op_i(2), store_i-1(6,7)
op_i(2), load_i+1(2,3)
…

C55 motion estimation

- Search strategy:
  - Full vs. non-full.
- Accuracy:
  - Full-pixel vs. half-pixel.
- Number of returned motion vectors:
  - 1 (one 16x16) vs. 4 (four 8x8).
- Algorithms:
  - 3-step algorithm (distance 4,2,1).
  - 4-step algorithm (distance 8,4,2,1).
  - 4-step with half-pixel refinement.

Four-step motion estimation breakdown

```
d = {8,4,2,1};
for (i=0; i<4; i++) {
    compute 3 upper differences
    for d[i];
    compute 3 middle differences
    for d[i];
    compute 3 lower differences for
    d[i];
    compute minimum value;
    move to next d;
}
```

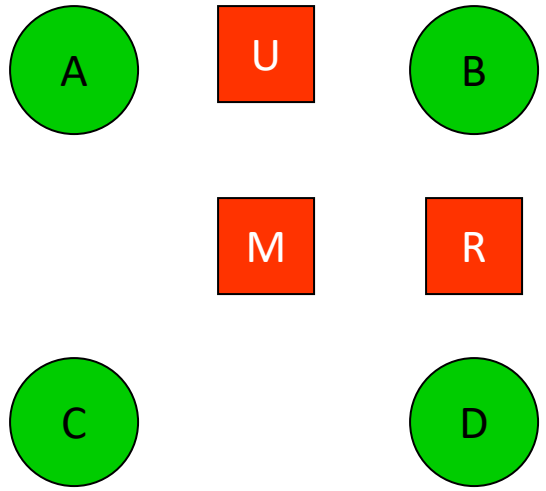


C55 motion estimation accelerator

- Includes 3 16-bit pixel data paths, 3 16-bit absolute differences (ADs).
- Basic operation:
  - [ACx,ACy] = copr(k8,ACx,ACy,Xmem,Ymem,Coeff)
  - K8 = control bits (enable AD units, etc.)
  - ACx, ACy = accumulated absolute differences
  - Xmem, Ymem = pointers to odd, even lines of the search window
  - Pointer to two adjacent pixels from reference window

C55 pixel interpolation

- Given four pixels A, B, C, D, interpolate three half-pixels:



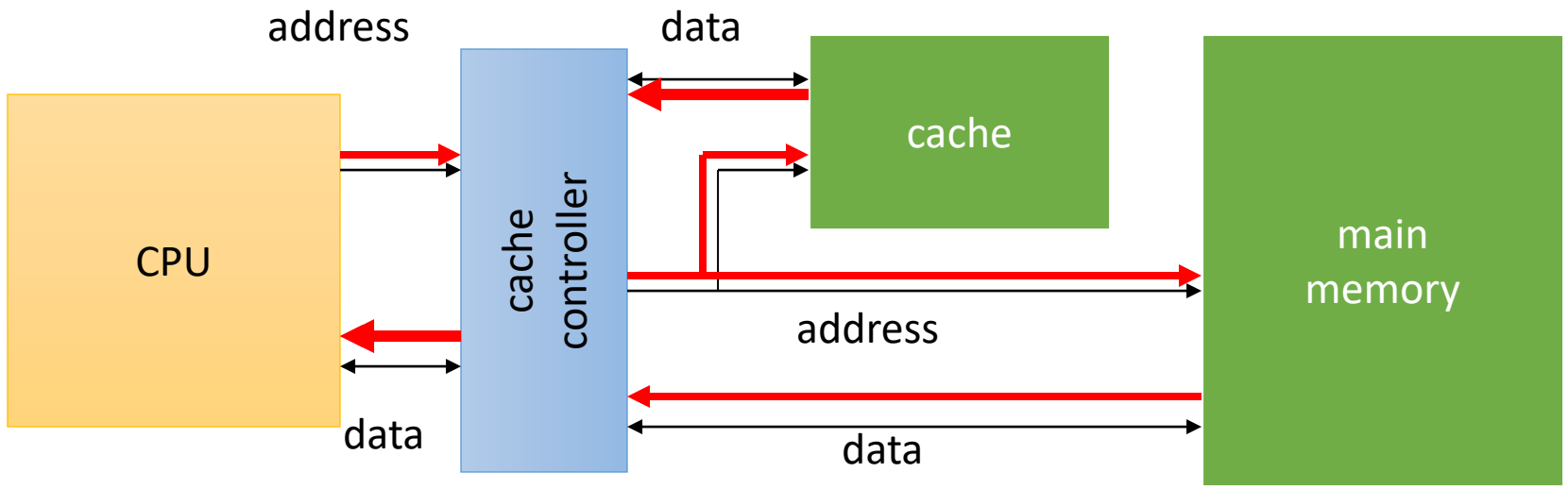
Pixel interpolation coprocessor operations

- Load pixels and compute:
  - ACy=copr(k8,AC,Lmem)
- Load pixels, compute, and store:
  - ACy=copr(k8,AACx,Lmem) || Lmem=ACz

CPUs

- Caches.
- Memory management.

Caches and CPUs



Cache operation

- Many main memory locations are mapped onto one cache entry.
- May have caches for:
  - instructions;
  - data;
  - data + instructions (unified).
- Memory access time is no longer deterministic.

Terms

- Cache hit: required location is in cache.
- Cache miss: required location is not in cache.
- Working set: set of locations used by program in a time interval.

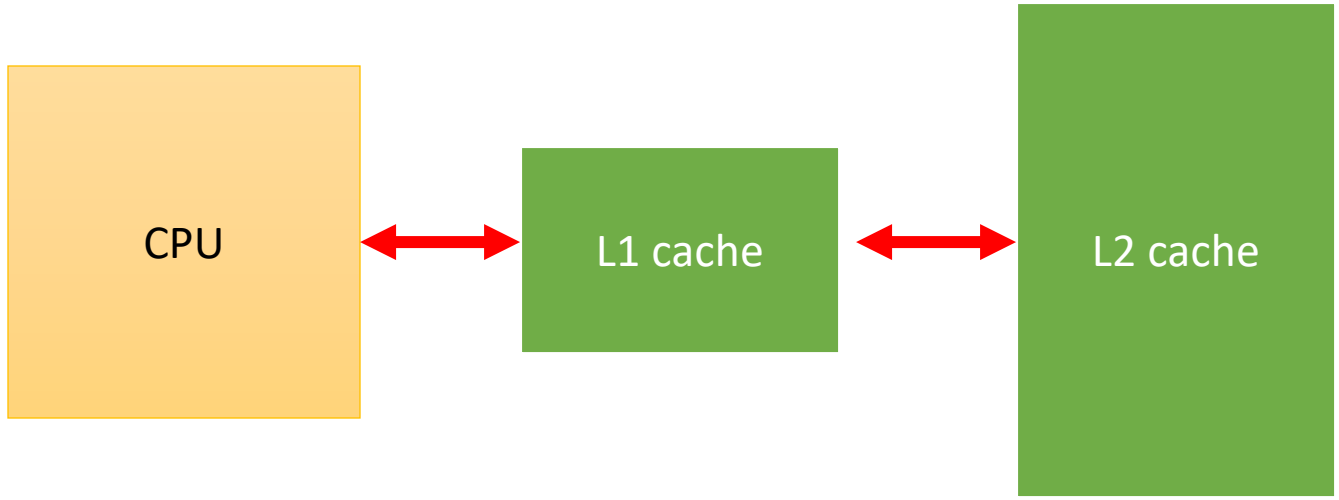
Types of misses

- Compulsory (cold): location has never been accessed.
- Capacity: working set is too large.
- Conflict: multiple locations in working set map to same cache entry.

Memory system performance

- h = cache hit rate.
- t_{cache} = cache access time, t_{main} = main memory access time.
- Average memory access time:
  - $t_{av} = ht_{cache} + (1-h)t_{main}$

Multiple levels of cache



Multi-level cache access time

- h_1 = cache hit rate.
- h_2 = rate for miss on L1, hit on L2.
- Average memory access time:
  - $t_{av} = h_1 t_{L1} + (h_2 - h_1) t_{L2} + (1 - h_2 - h_1) t_{main}$

Replacement policies

- Replacement policy: strategy for choosing which cache entry to throw out to make room for a new memory location.
- Two popular strategies:
  - Random.
  - Least-recently used (LRU).

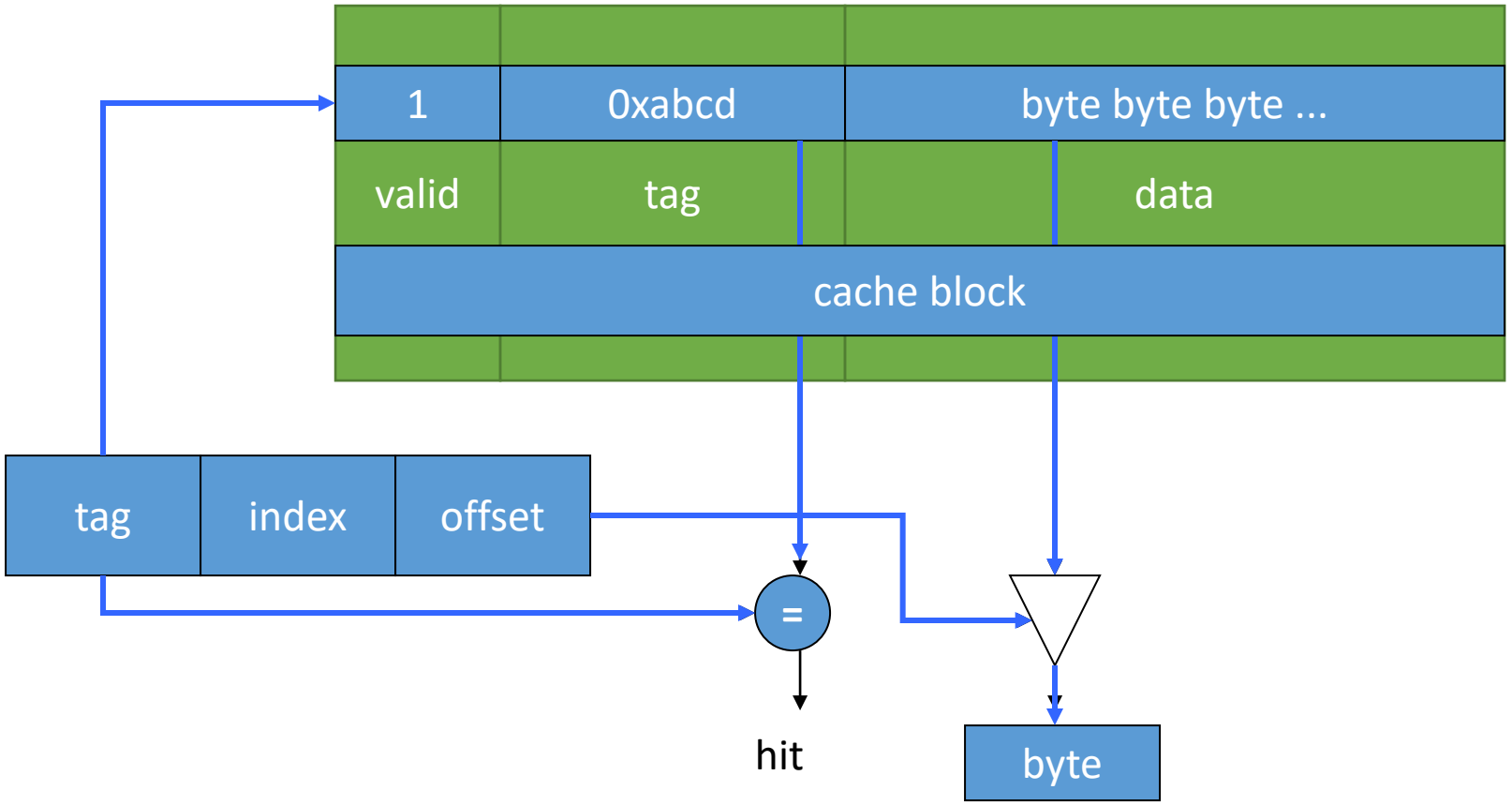
Cache organizations

- Fully-associative: any memory location can be stored anywhere in the cache (almost never implemented).
- Direct-mapped: each memory location maps onto exactly one cache entry.
- N-way set-associative: each memory location can go into one of n sets.

Cache performance benefits

- Keep frequently-accessed locations in fast cache.
- Cache retrieves more than one word at a time.
  - Sequential accesses are faster after first access.

Direct-mapped cache



Write operations

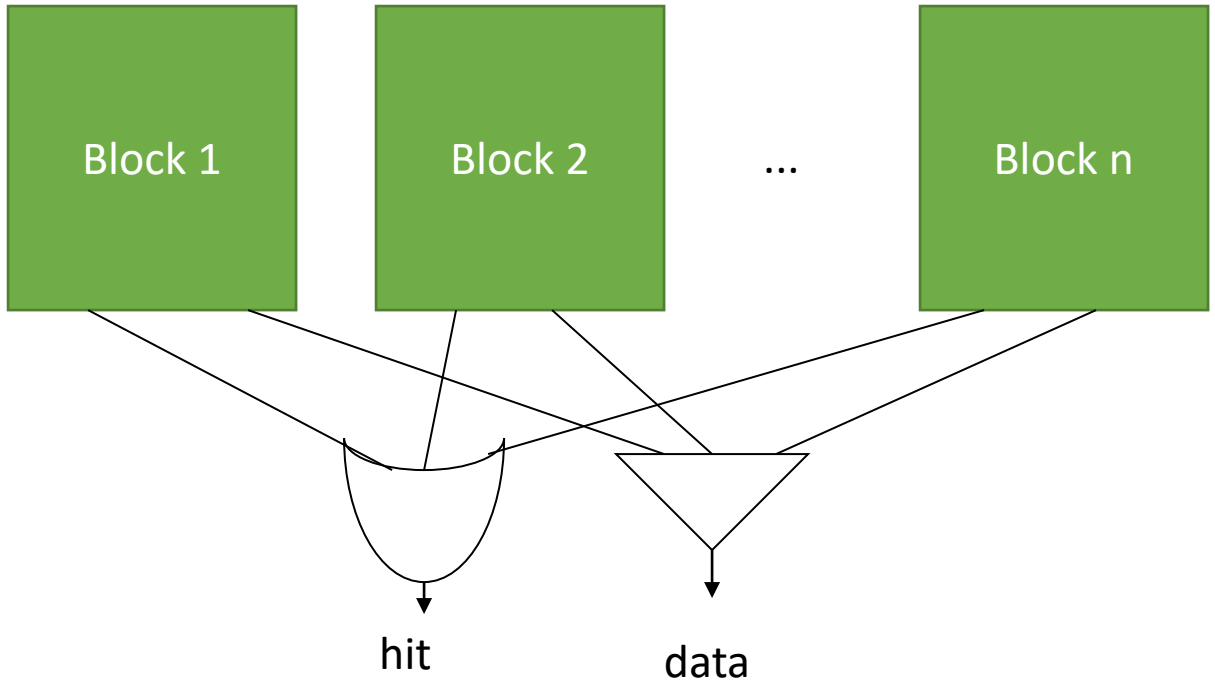
- Write-through: immediately copy write to main memory.
- Write-back: write to main memory only when location is removed from cache.

Direct-mapped cache locations

- Many locations map onto the same cache block.
- Conflict misses are easy to generate:
  - Array a[] uses locations 0, 1, 2, …
  - Array b[] uses locations 1024, 1025, 1026, …
  - Operation a[i] + b[i] generates conflict misses.

Set-associative cache

- A set of direct-mapped caches:



Example: direct-mapped vs. set-associative

address	data
000	0101
001	1111
010	0000
011	0110
100	1000
101	0001
110	1010
111	0100

Direct-mapped cache behavior

- After 001 access:

block	tag	data
00	-	-
01	0	1111
10	-	-
11	-	-

- After 010 access:

block	tag	data
00	-	-
01	0	1111
10	0	0000
11	-	-

Direct-mapped cache behavior, cont'd.

- After 011 access:

block	tag	data
00	-	-
01	0	1111
10	0	0000
11	0	0110

- After 100 access:

block	tag	data
00	1	1000
01	0	1111
10	0	0000
11	0	0110

Direct-mapped cache behavior, cont'd.

- After 101 access:

block	tag	data
00	1	1000
01	1	0001
10	0	0000
11	0	0110

- After 111 access:

block	tag	data
00	1	1000
01	1	0001
10	0	0000
11	1	0100

2-way set-associtive cache behavior

- Final state of cache (twice as big as direct-mapped):

blk	set 0 tag	set 0 data	set 1 tag	set 1 data
00	1	1000	-	-
01	0	1111	1	0001
10	0	0000	-	-
11	0	0110	1	0100

2-way set-associative cache behavior

- Final state of cache (same size as direct-mapped):

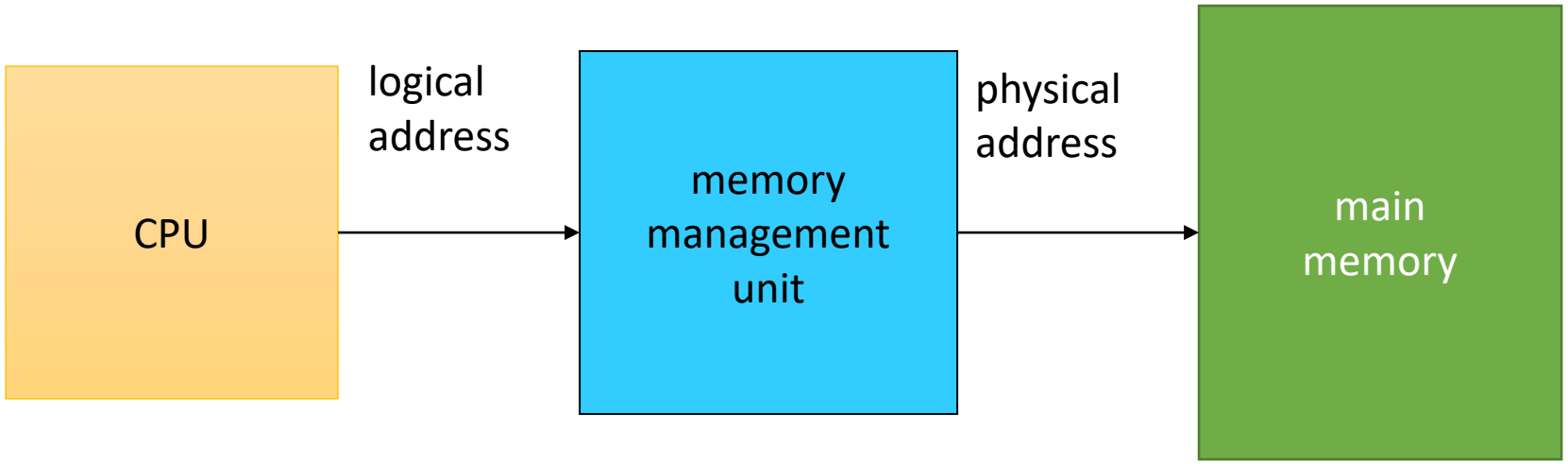
blk	set 0 tag	set 0 data	set 1 tag	set 1 data
0	01	0000	10	1000
1	10	0001	11	0100

Example caches

- StrongARM:
  - 16 Kbyte, 32-way, 32-byte block instruction cache.
  - 16 Kbyte, 32-way, 32-byte block data cache (write-back).
- SHARC:
  - 32-instruction, 2-way instruction cache.

Memory management units

- Memory management unit (MMU) translates addresses:



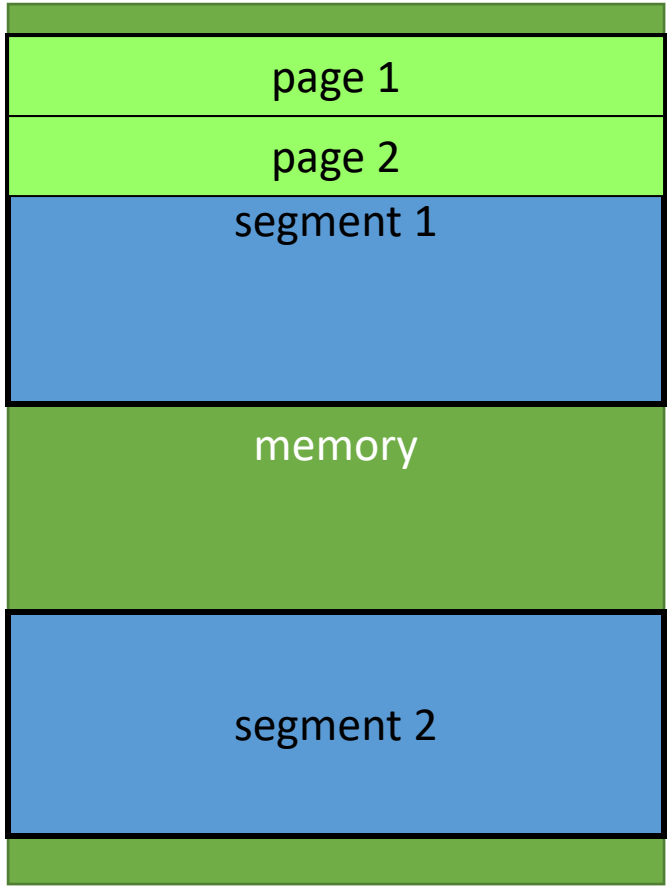
Memory management tasks

- Allows programs to move in physical memory during execution.
- Allows virtual memory:
  - memory images kept in secondary storage;
  - images returned to main memory on demand during execution.
- Page fault: request for location not resident in memory.

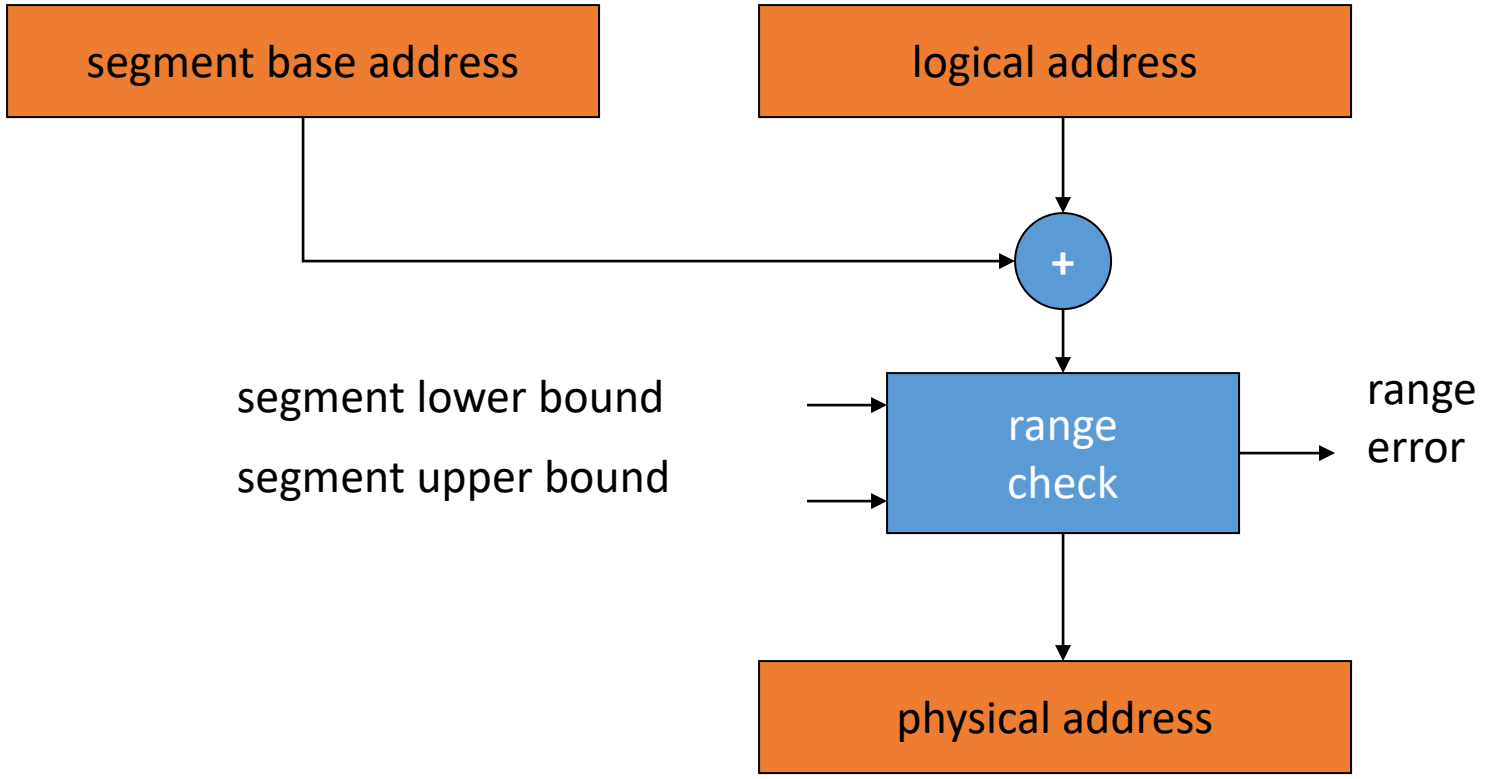
Address translation

- Requires some sort of register/table to allow arbitrary mappings of logical to physical addresses.
- Two basic schemes:
  - segmented;
  - paged.
- Segmentation and paging can be combined (x86).

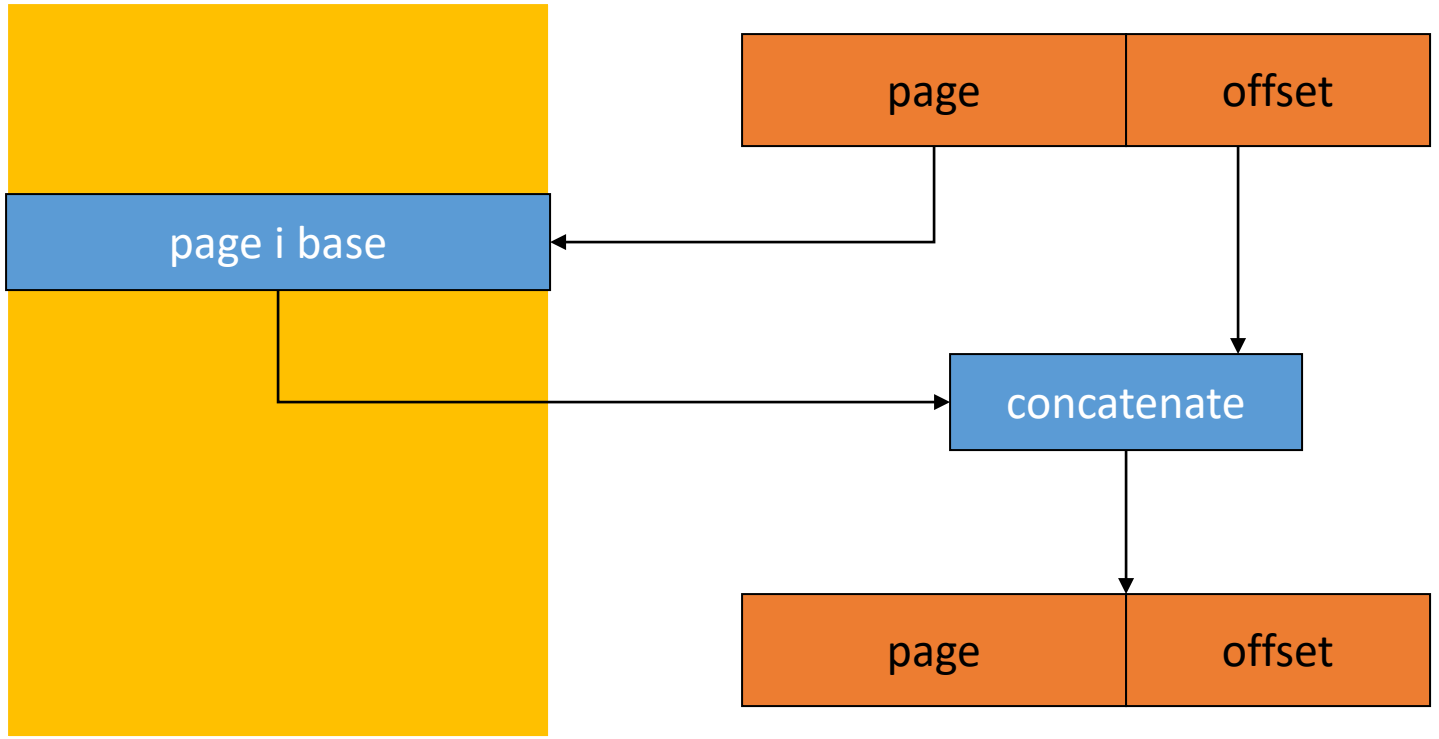
Segments and pages



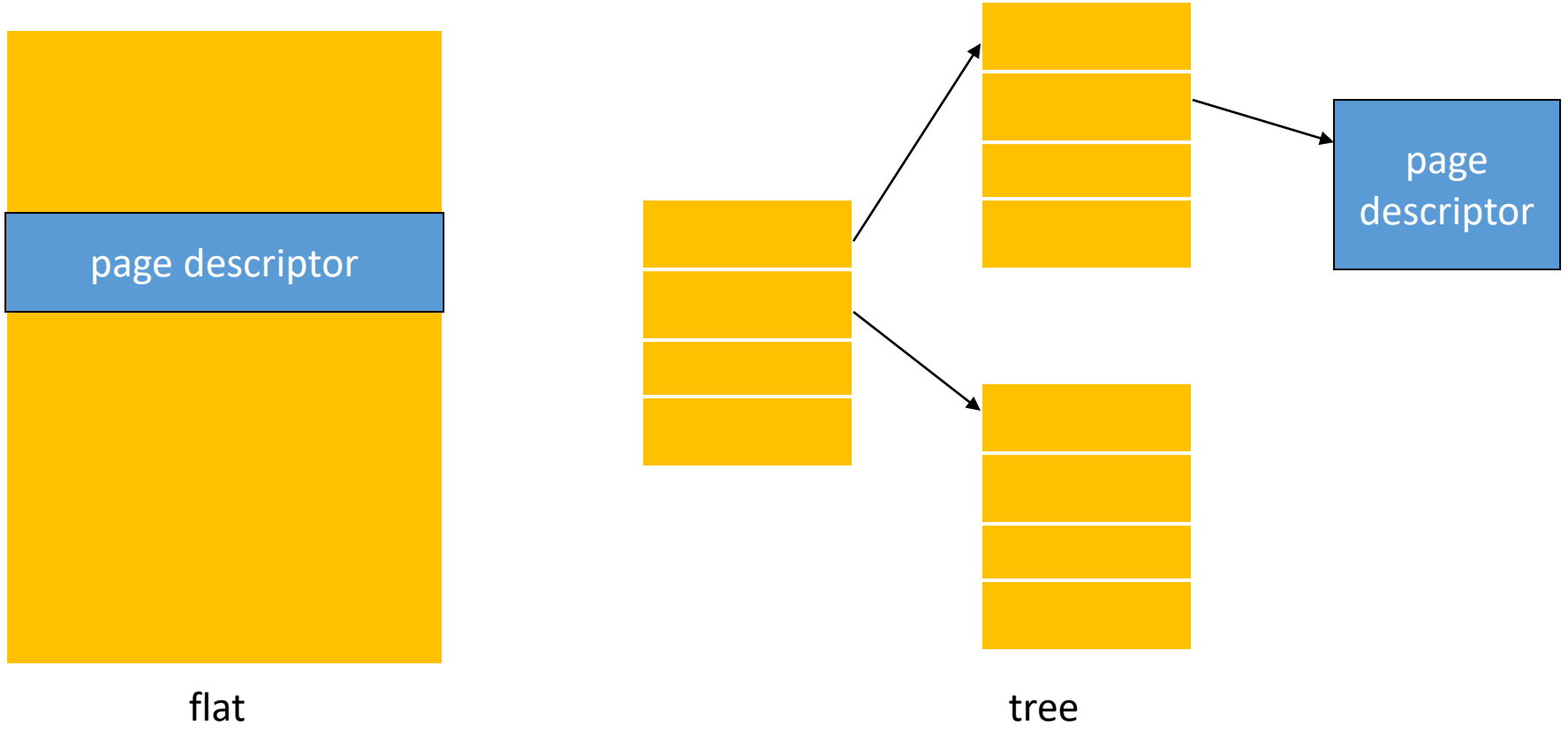
Segment address translation



Page address translation



Page table organizations



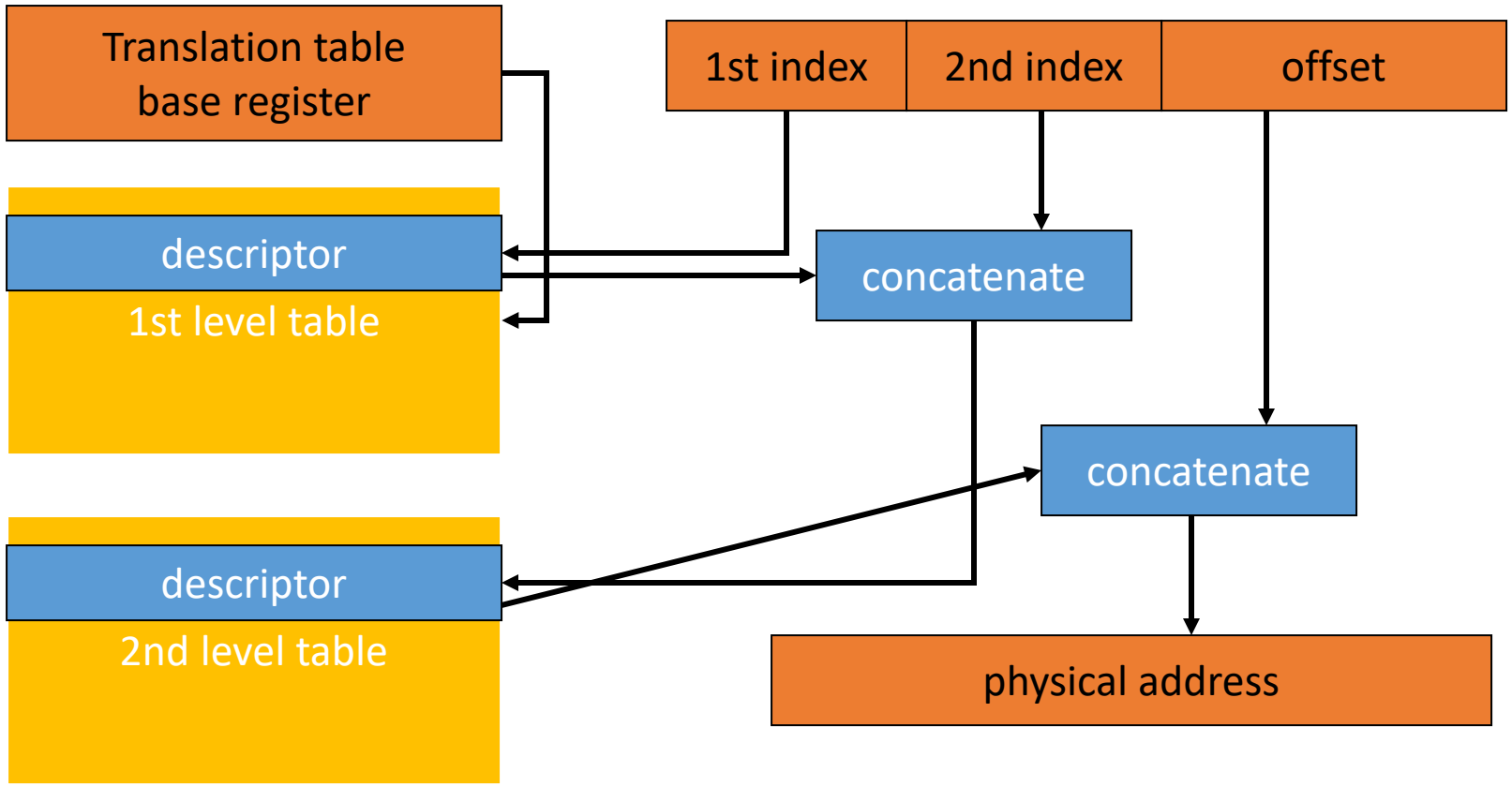
Caching address translations

- Large translation tables require main memory access.
- TLB: cache for address translation.
  - Typically small.

ARM memory management

- Memory region types:
  - section: 1 Mbyte block;
  - large page: 64 kbytes;
  - small page: 4 kbytes.
- An address is marked as section-mapped or page-mapped.
- Two-level translation scheme.

ARM address translation



CPUs

- CPU performance.
- CPU power consumption and power management.
- Safety and security.

Elements of CPU performance

- Cycle time.
- CPU pipeline.
- Memory system.

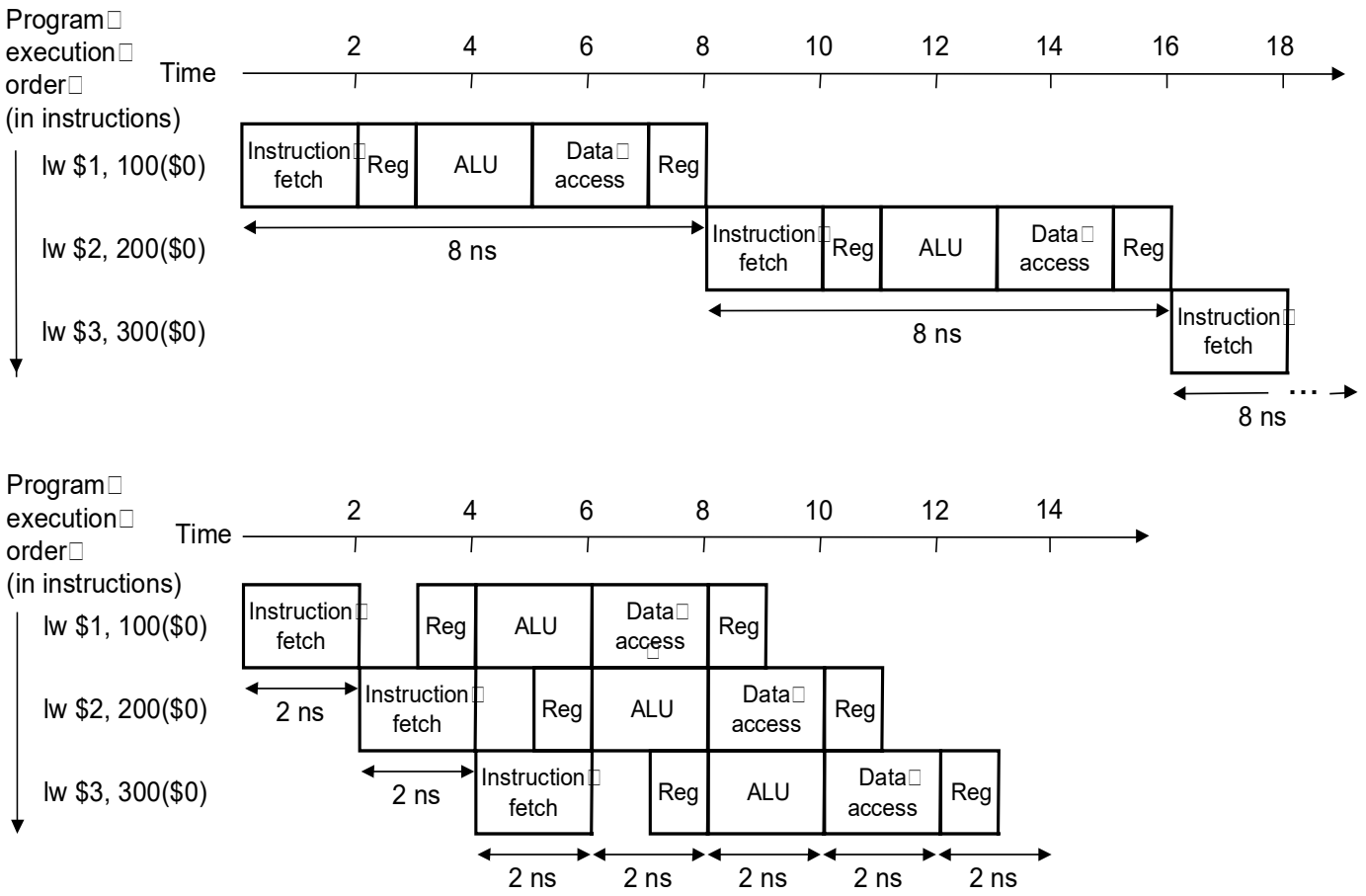
Pipelining

- Several instructions are executed simultaneously at different stages of completion.
- Various conditions can cause pipeline bubbles that reduce utilization:
  - branches;
  - memory system delays;
  - etc.

Performance measures

- Latency: time it takes for an instruction to get through the pipeline.
- Throughput: number of instructions executed per time period.
- Pipelining increases throughput without reducing latency.

Single-cycle,nonpipelining & Pipelining



Pipelining is an implementation technique in which multiple instructions are overlapped in execution.

Feature

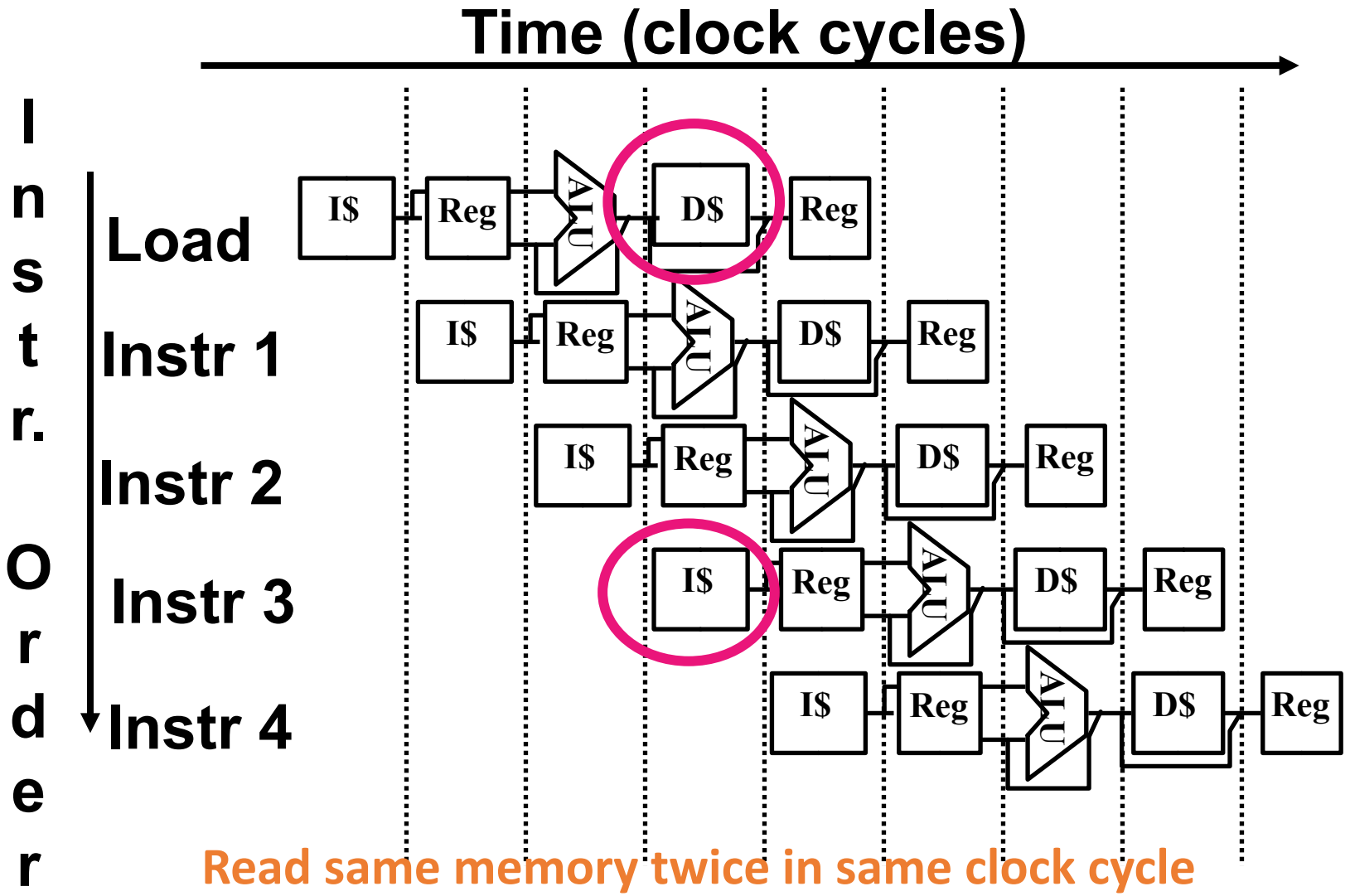
- Optimal Pipeline
  - One phase of instruction execution per cycle
  - At each clock cycle, one instruction is completed
  - On average, the execution is much faster
- What makes this work?
  - The similarity between instructions makes it possible for all instructions to use the same stage (in general)
  - Each stage takes about the same amount of time: less time wasted

Problems for Pipelining CPUs

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support some combination of instructions that we want to execute in the same clock cycle.
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline.
  - Control hazards: Pipelining of branches causes later instruction fetches to wait for the result of the branch.

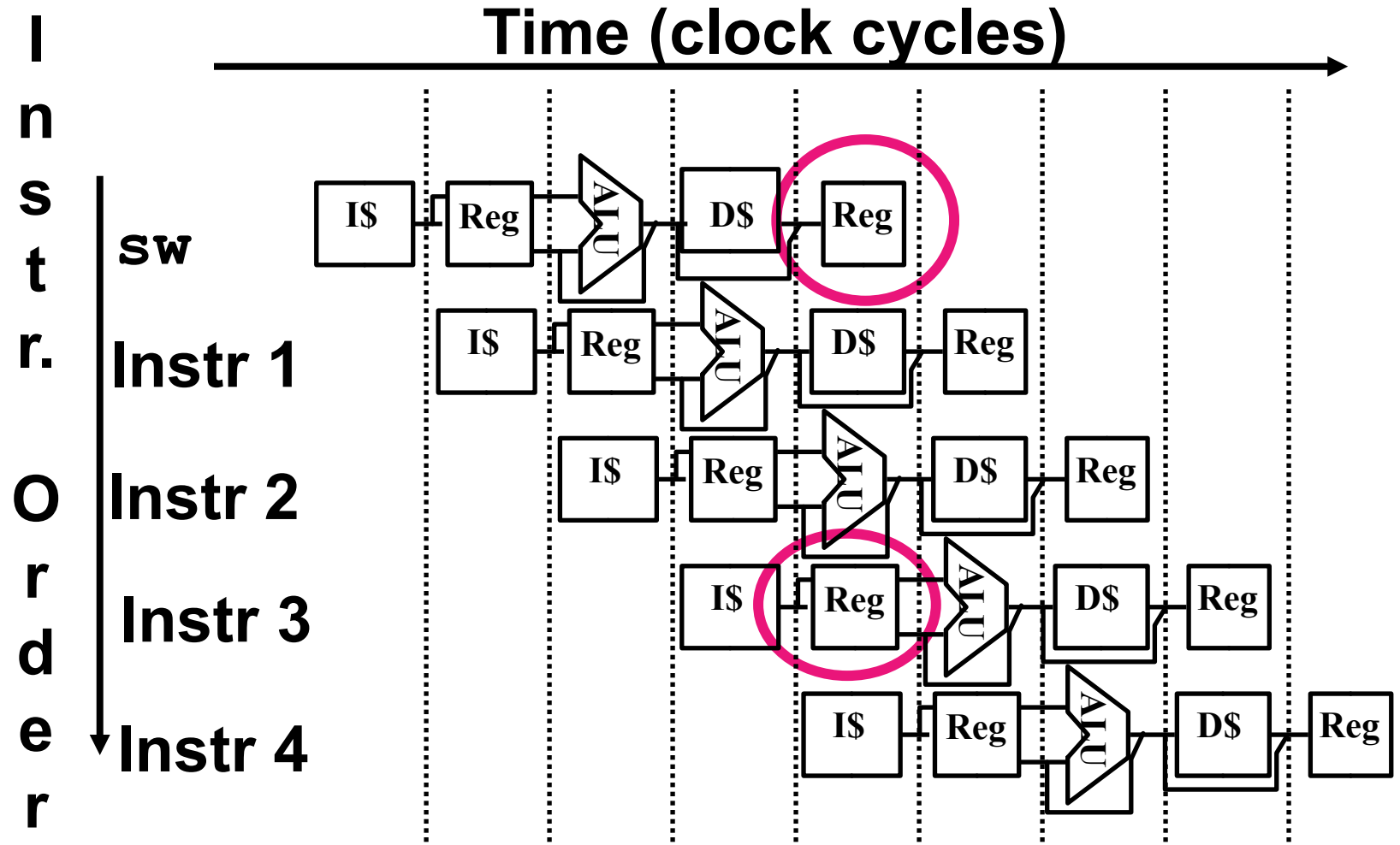
- So we need pipeline stalls or "bubbles".

Structural Hazard #1: Single Memory



Read same memory twice in same clock cycle

Structural Hazard #2: Registers

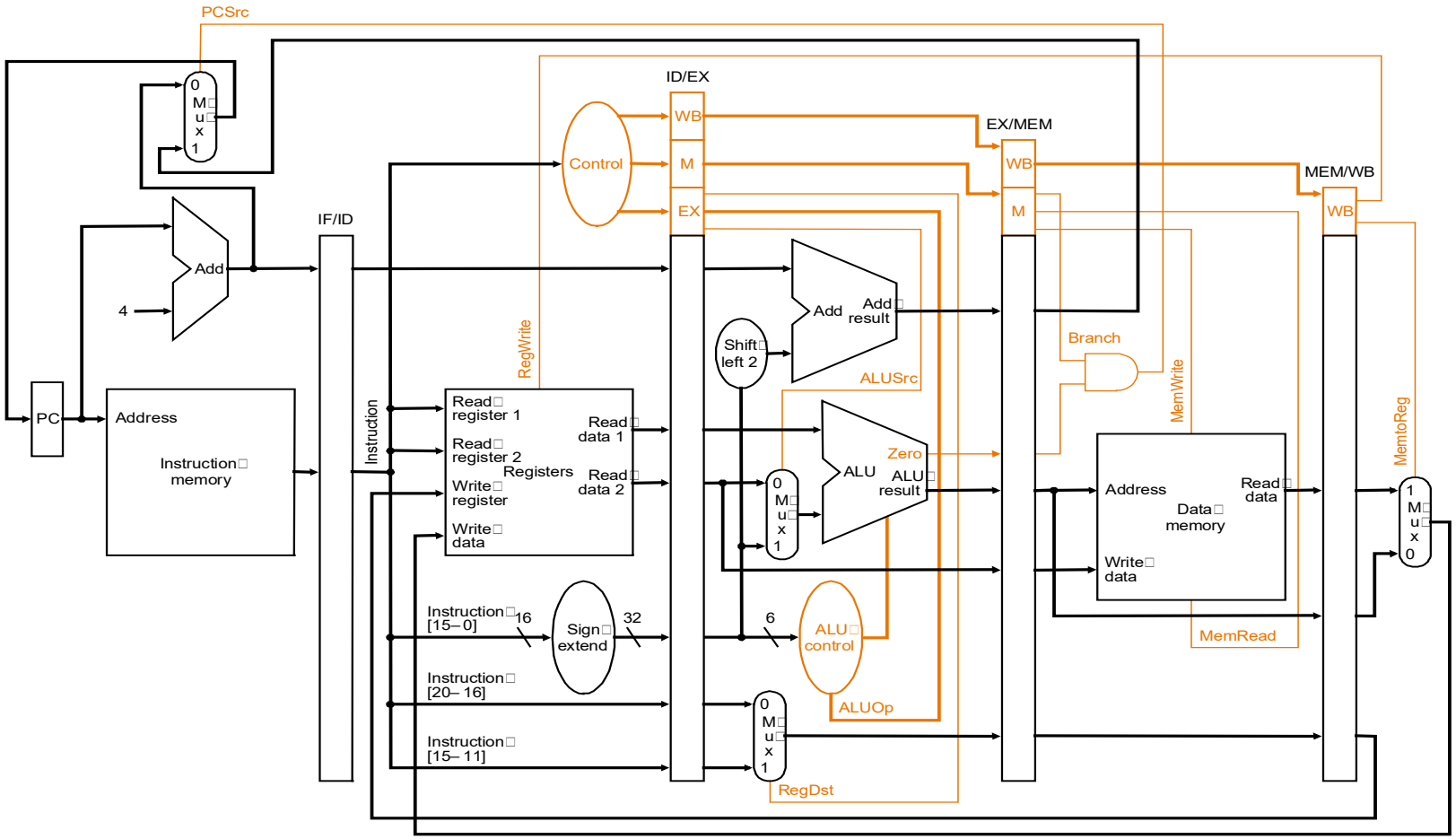


Can we read and write to registers simultaneously?

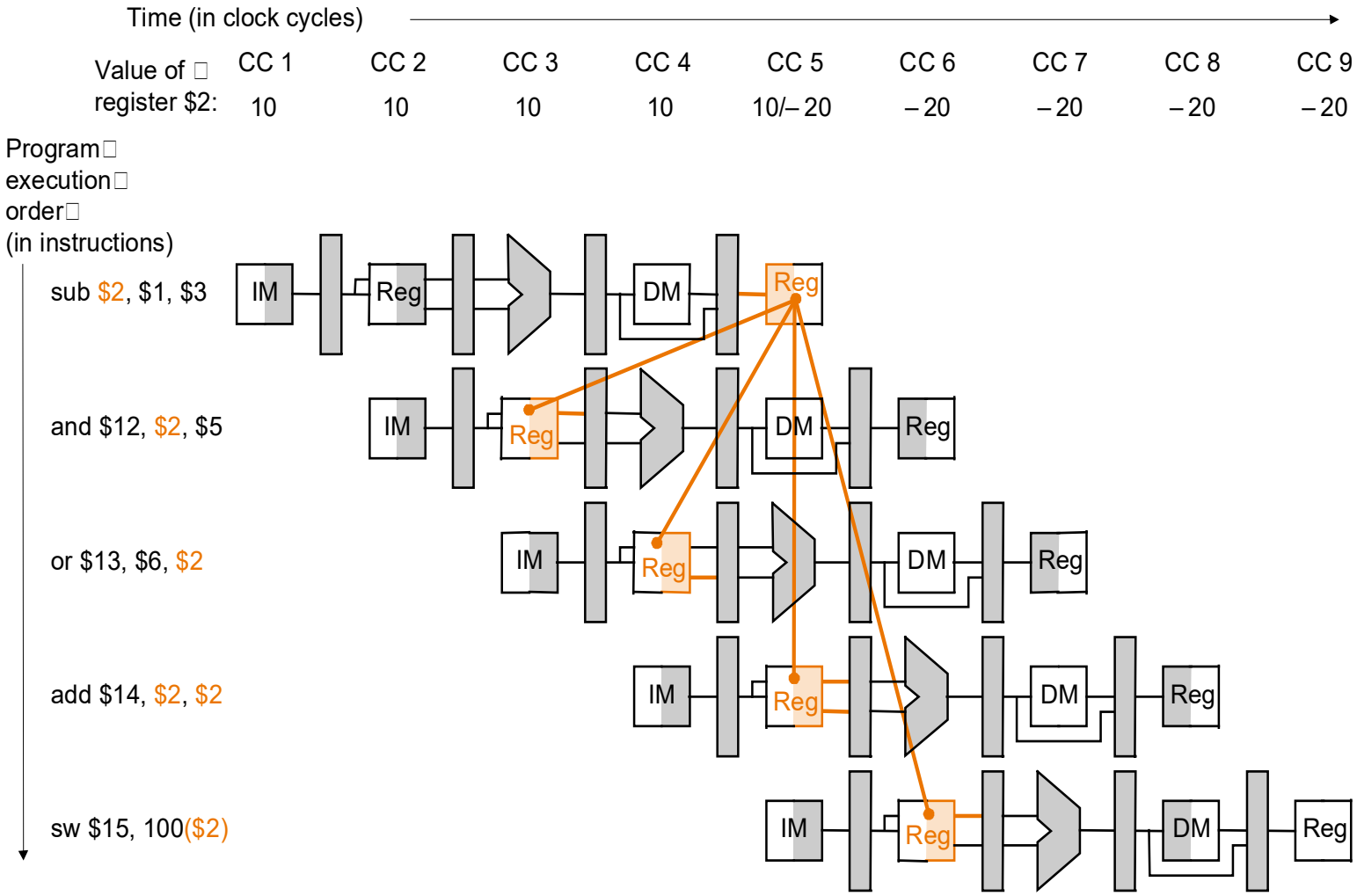
Resolving Structural Hazards

- Structural hazards occurs when two instruction need same hardware resource at same time
  - Can resolve in hardware by stalling newer instruction till older instruction finished with resource
- A structural hazard can always be avoided by adding more hardware to design
  - E.g., if two instructions both need a port to memory at same time, could avoid hazard by adding second port to memory

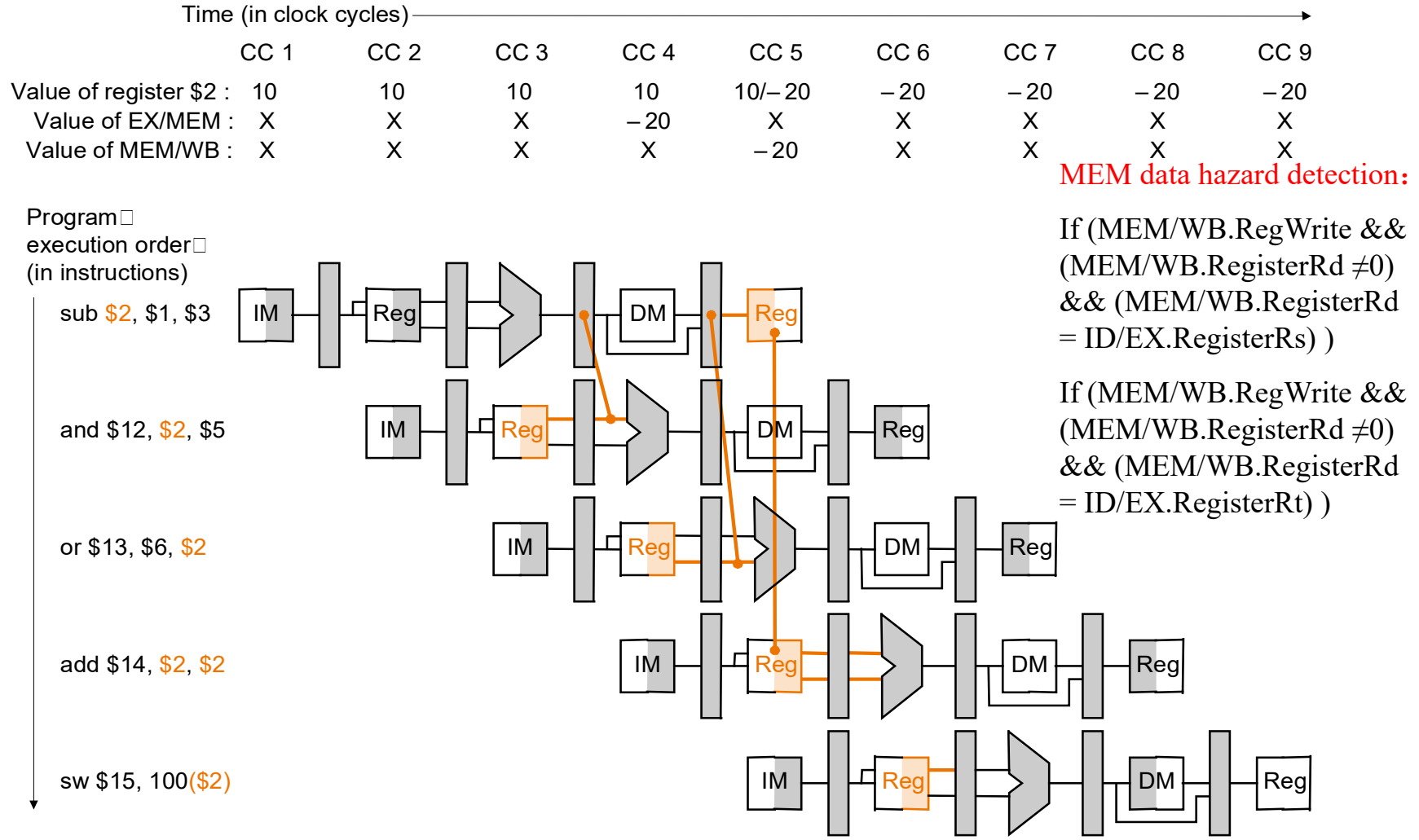
Datapath with Control



Pipelined Dependencies



Forwarding

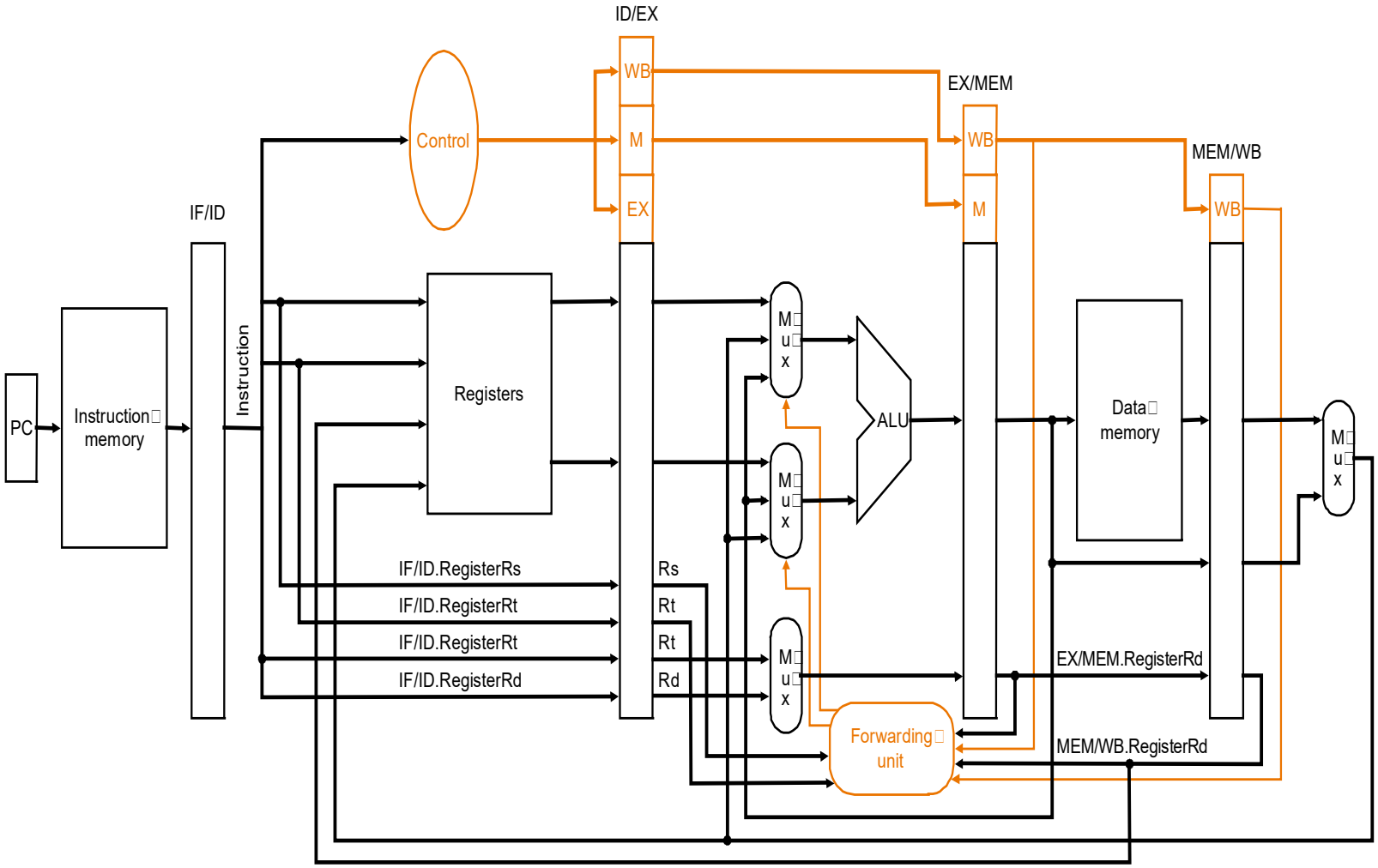


MEM data hazard detection:

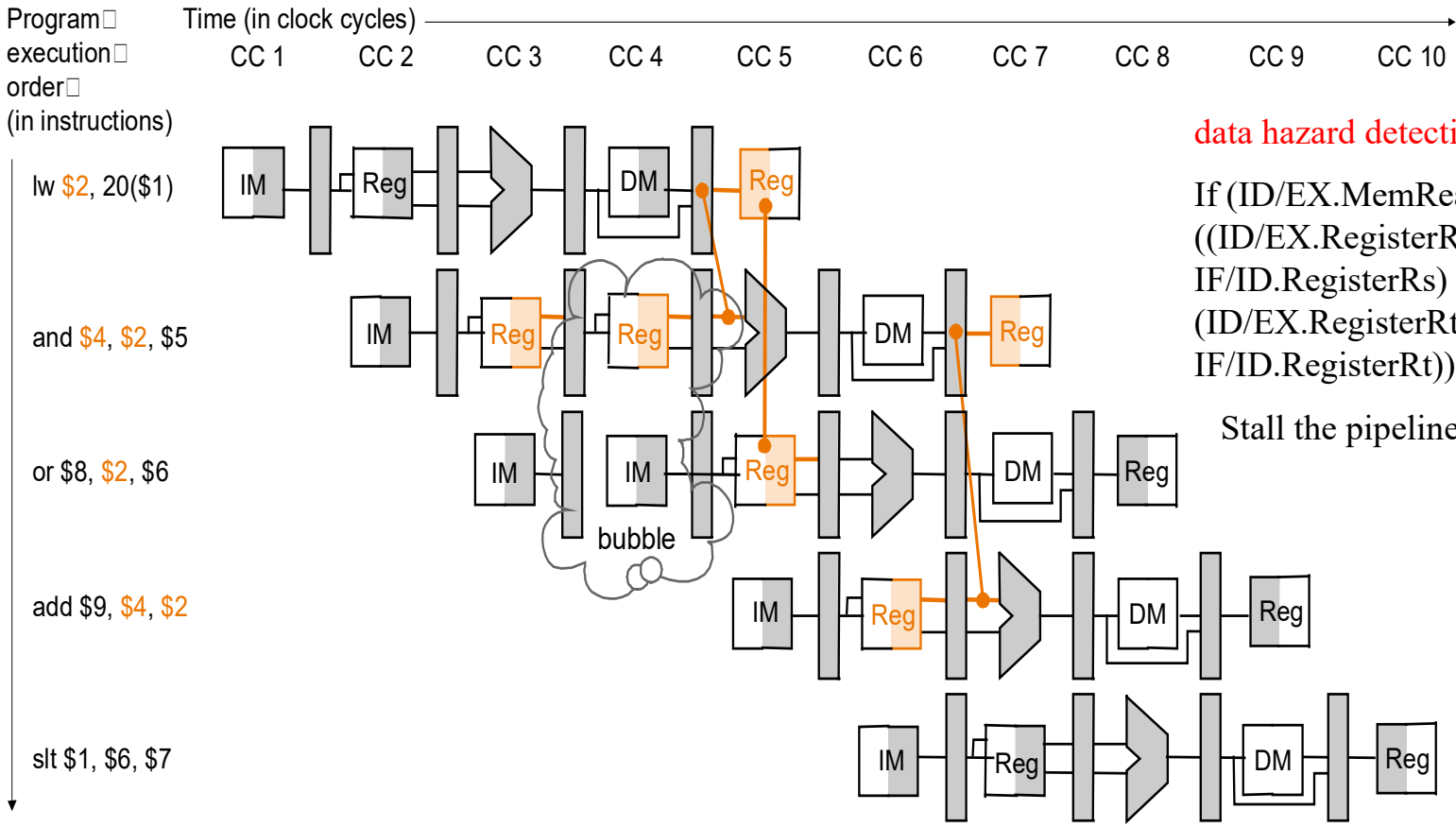
If (MEM/WB.RegWrite && (MEM/WB.RegisterRd ≠0) && (MEM/WB.RegisterRd = ID/EX.RegisterRs))

If (MEM/WB.RegWrite && (MEM/WB.RegisterRd ≠0) && (MEM/WB.RegisterRd = ID/EX.RegisterRt))

Forwarding


ID/EX
WB
M
EX
Control
EX/MEM
WB
M
MEM/WB
WB
IF/ID
PC
Instruction memory
Instruction
Registers
ALU
Data memory
M u x
M u x
M u x
M u x
IF/ID.RegisterRs
IF/ID.RegisterRt
IF/ID.RegisterRt
IF/ID.RegisterRd
Rs
Rt
Rt
Rd
EX/MEM.RegisterRd
MEM/WB.RegisterRd
Forwarding unit

Stalling

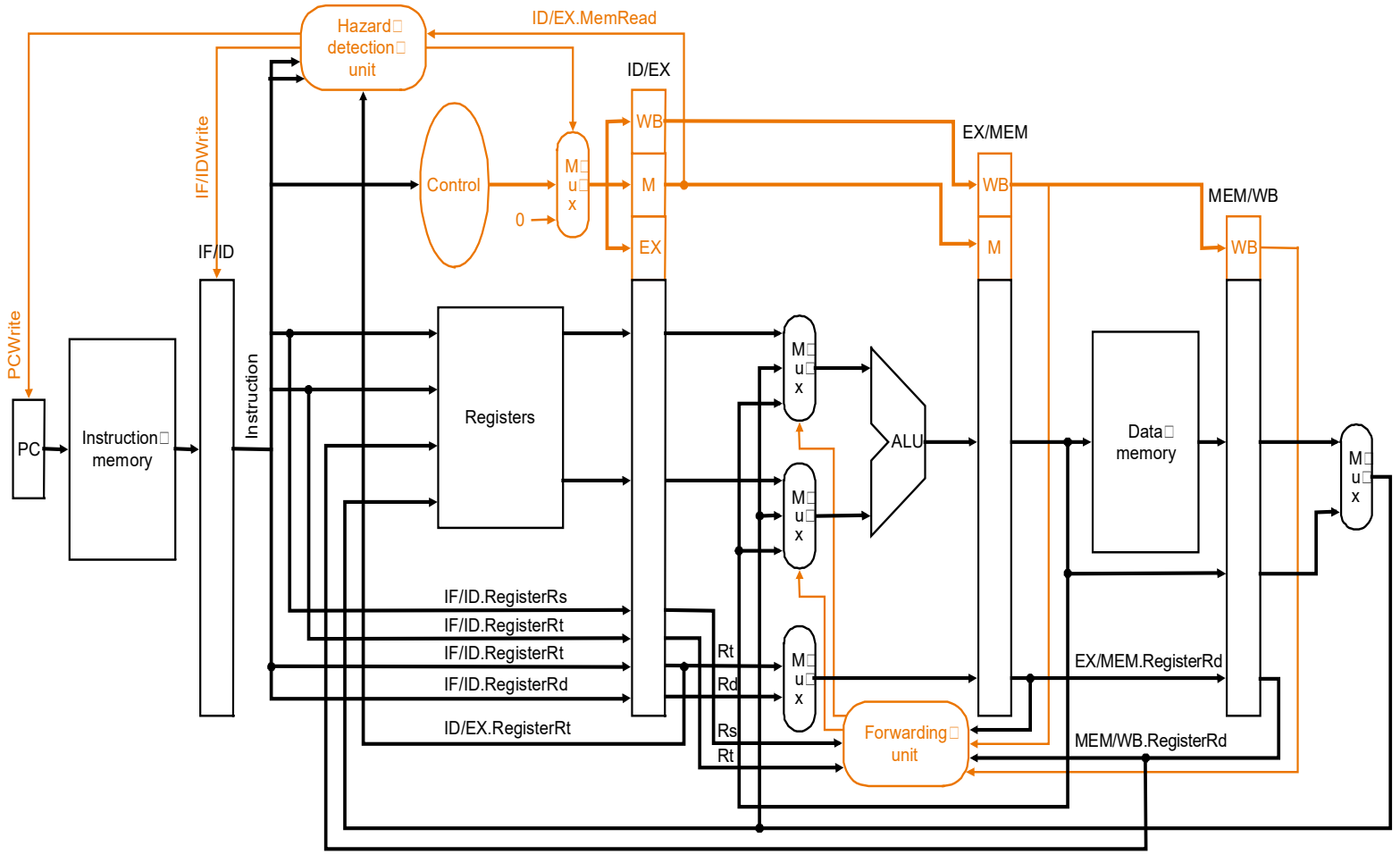


data hazard detection：

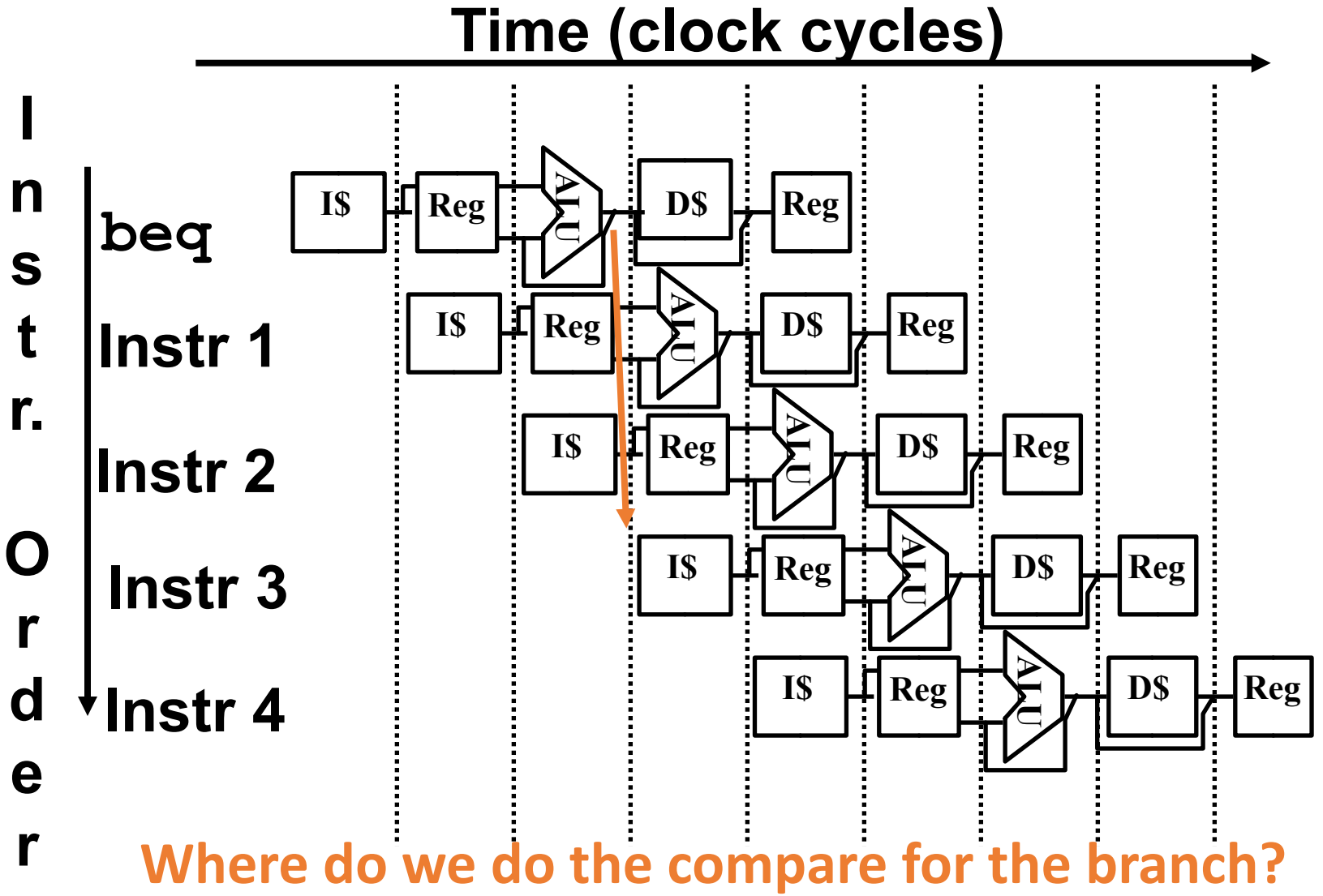
If (ID/EX.MemRead && ((ID/EX.RegisterRt = IF/ID.RegisterRs) || (ID/EX.RegisterRt = IF/ID.RegisterRt)))

Stall the pipeline

Hazard Detection Unit



Control Hazard: Branching



Where do we do the compare for the branch?

Control Hazard: Branching

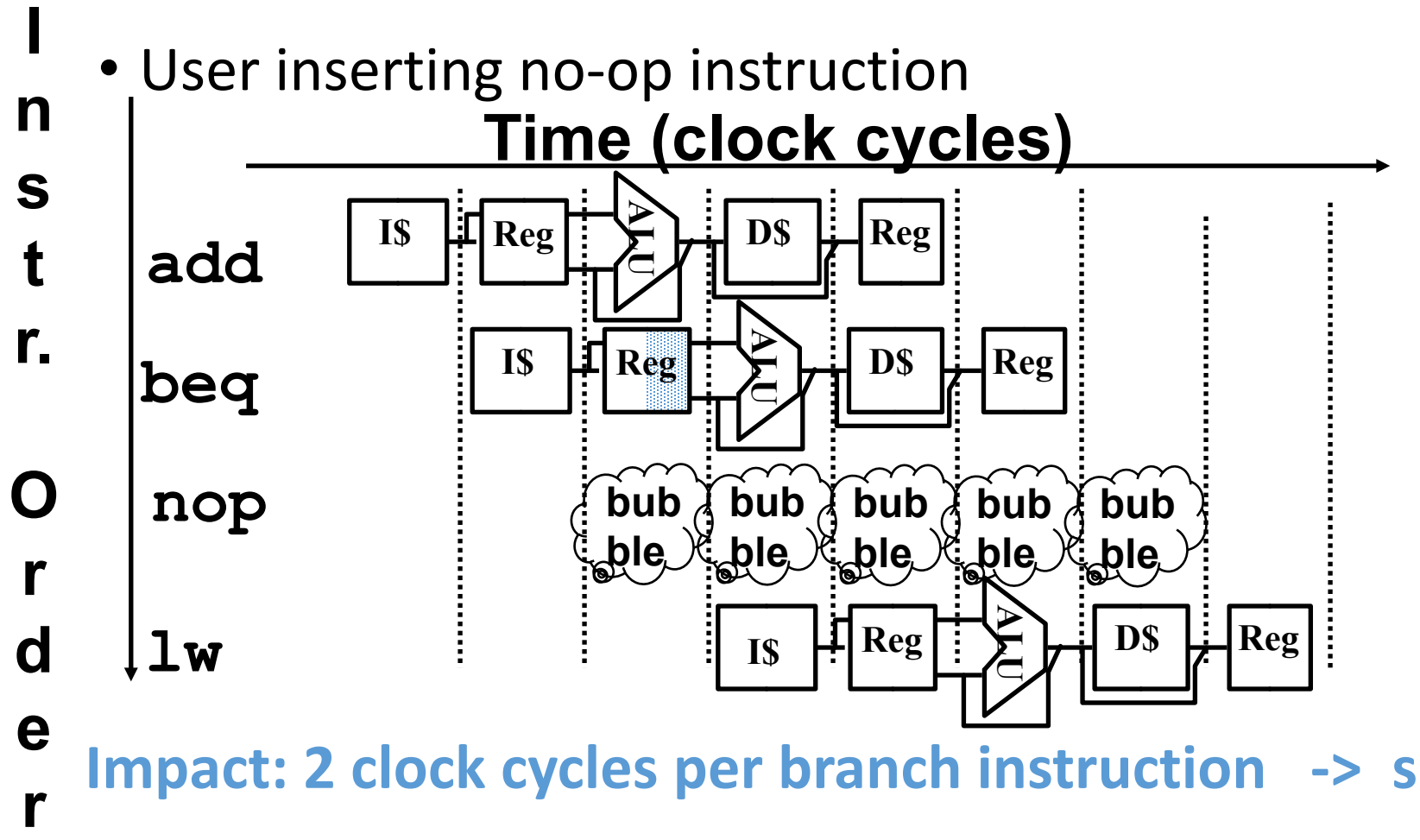
- Stall until decision is made
  - Insert the "no-op" instruction (which does nothing but consume time) or hold up the fetch of the next instruction (for 2 cycles).
  - Cons: Each branch statement takes 3 clock cycles (assuming the comparison is made in the ALU phase)

Control Hazard: Branching

- Optimization #1:
  - insert special branch comparator in Stage 2
  - as soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
  - Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
  - Side Note: This means that branches are idle in Stages 3, 4 and 5.

Control Hazard: Branching (6/8)

- User inserting no-op instruction



Impact: 2 clock cycles per branch instruction -> slow

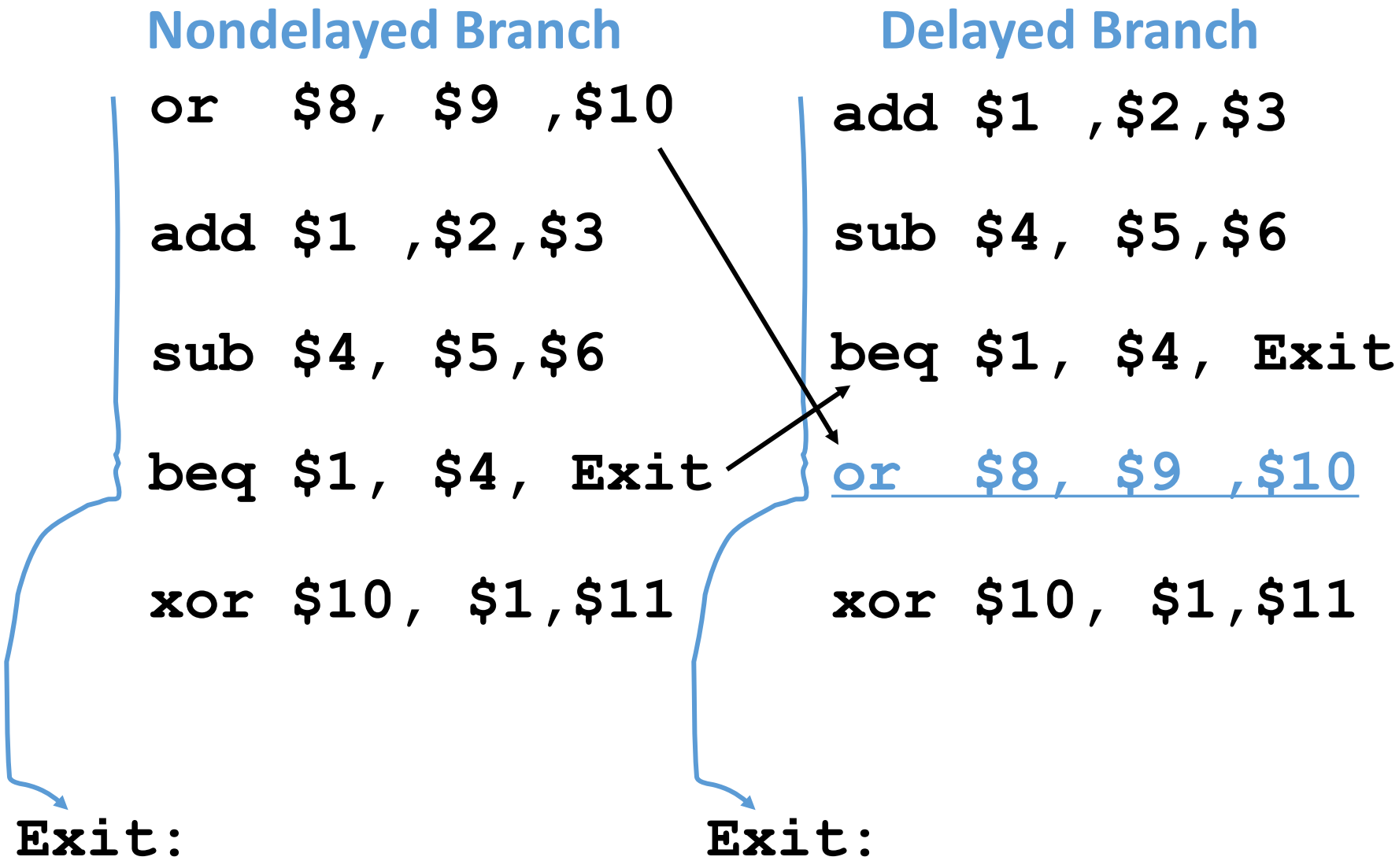
Control Hazard: Branching

- Optimization #2: Redefining branches
  - Old definition: if we take the branch, none of the instructions after the branch get executed by accident
  - New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (called the branch-delay slot)
- The term "Delayed Branch" means we always execute inst after branch

Control Hazard: Branching

- Notes on Branch-Delay Slot
  - Worst case: can always put a no-op in the branch-delay slot
  - Better case: Find an instruction before the branch, which can be put into the branch-delay slot without affecting the flow of the program
    - re-ordering instructions is a common method of speeding up programs
  - compiler must be very smart in order to find instructions to do this
  - usually can find such an instruction at least 50% of the time
  - Jumps also have a delay slot…

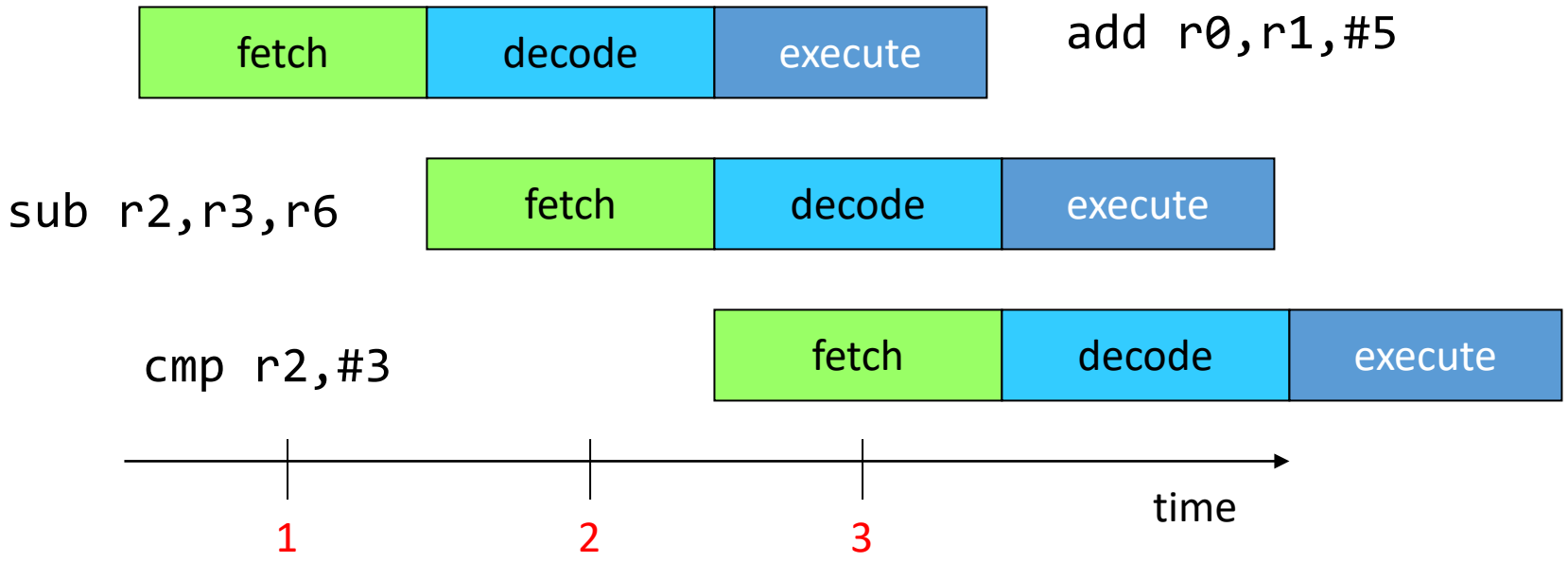
Example: Nondelayed vs. Delayed Branch



ARM7 pipeline

- ARM 7 has 3-stage pipe:
  - fetch instruction from memory;
  - decode opcode and operands;
  - execute.

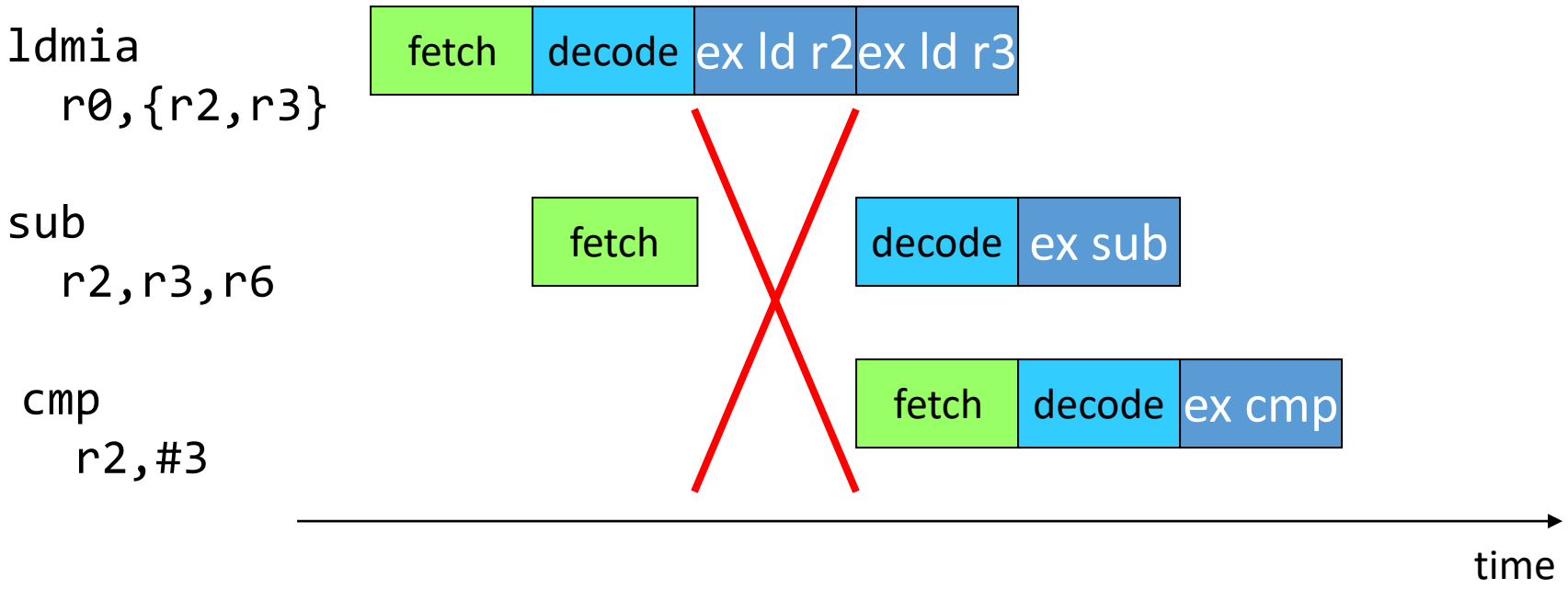
ARM pipeline execution



Pipeline stalls

- If every step cannot be completed in the same amount of time, pipeline stalls.
- Bubbles introduced by stall increase latency, reduce throughput.

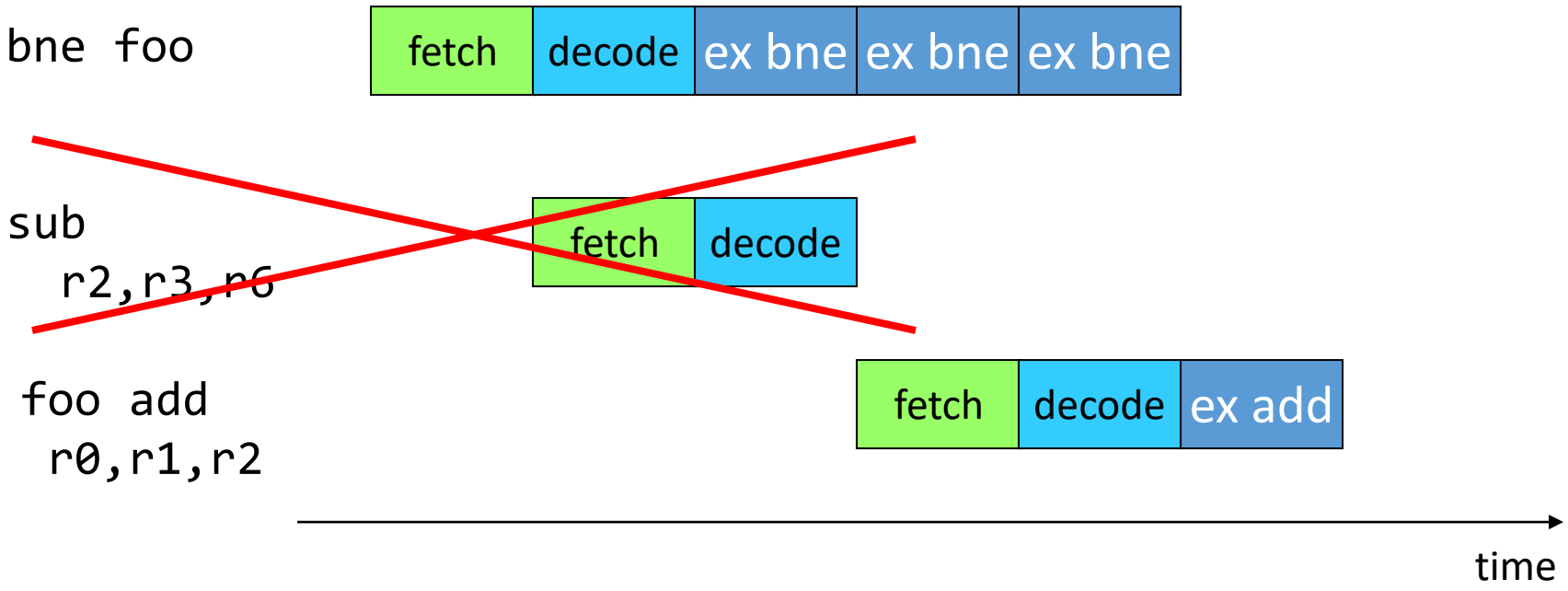
ARM multi-cycle LDMIA instruction



Control stalls

- Branches often introduce stalls (branch penalty).
  - Stall time may depend on whether branch is taken.
- May have to squash instructions that already started executing.
- Don’t know what to fetch until condition is evaluated.

ARM pipelined branch



Delayed branch

- To increase pipeline efficiency, delayed branch mechanism requires n instructions after branch always executed whether branch is executed or not.
- SHARC supports delayed and non-delayed branches.
  - Specified by bit in branch instruction.
  - 2 instruction branch delay slot.

Example: ARM execution time

- Determine execution time of FIR filter:

```
for (i=0, f=0; i<N; i++)
  f = f + c[i]*x[i];
```

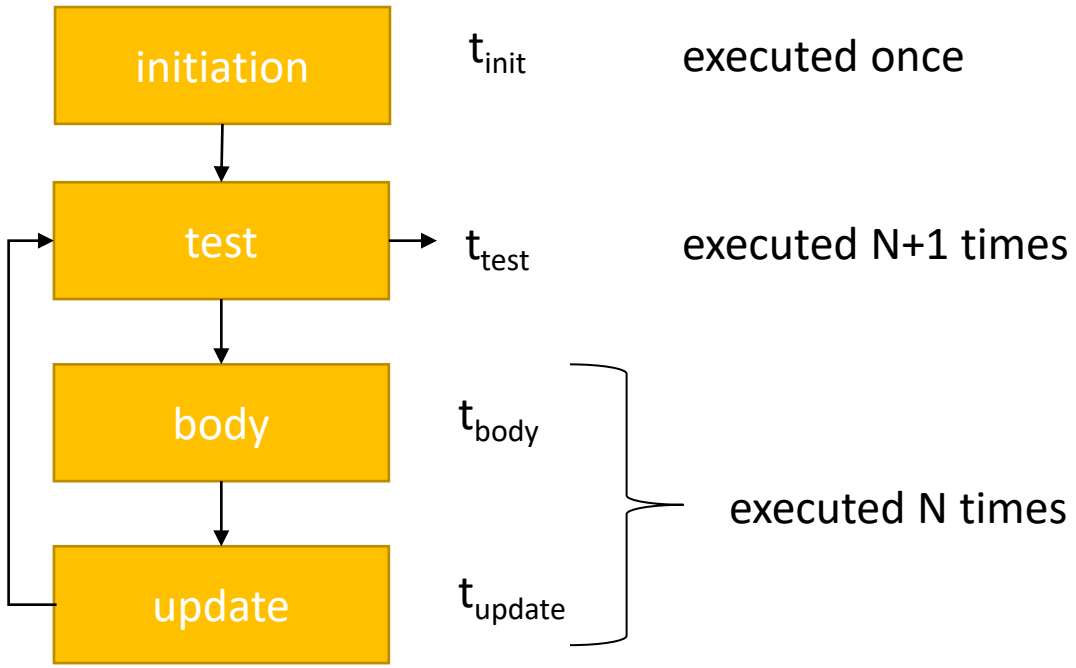
- Only branch in loop test may take more than one cycle.
  - `BLT loop` takes 1 cycle best case, 3 worst case.

FIR filter ARM code

```
; loop initiation code
MOV r0,#0 ; use r0 for i, set to 0
MOV r8,#0 ; use a separate index for arrays
ADR r2,N ; get address for N
LDR r1,[r2] ; get value of N
MOV r2,#0 ; use r2 for f, set to 0
ADR r3,c ; load r3 with address of base of c
ADR r5,x ; load r5 with address of base of x
```

```
; test for exit
loop        CMP r0,r1
 BGE loopend ; if I >= N, exit loop
; loop body
 LDR r4,[r3,r8] ; get value of c[i]
 LDR r6,[r5,r8] ; get value of x[i]
 MUL r4,r4,r6 ; compute c[i]*x[i]
 ADD r2,r2,r4 ; add into running sum
; update
 ADD r8,r8,#4 ; add one to array index
 ADD r0,r0,#1 ; add 1 to i
 B loop
loopend  ...
```

FIR filter block diagram



FIR filter performance by block

Block	Variable	# instructions	# cycles
Initiation	t_{init}	7	7
Test	t_{test}	2	[2,4]
Body	t_{body}	4	4
Update	t_{update}	3	3

$$t_{loop} = t_{init} + N(t_{body} + t_{update} + t_{test,best}) + t_{test,worst}$$

Loop exit fails is best case

Loop exit succeeds is worst case

PIC16F execution time

- Instruction divided into four Q cycles:
  - Q1 decodes.
  - Q2 reads operands.
  - Q3 performs data operations.
  - Q4 writes data.
- One Q cycle per clock cycle.
- Execution time for one instruction is Tcy.

PIC16F instruction timing

- Most instructions execute in one cycle, except:
  - CALL, GOTO, RETFIE, RETLW, RETURN require 2 cycles.
  - Skip-if instructions require two cycles if skip is taken.

PIC16F loop execution timing

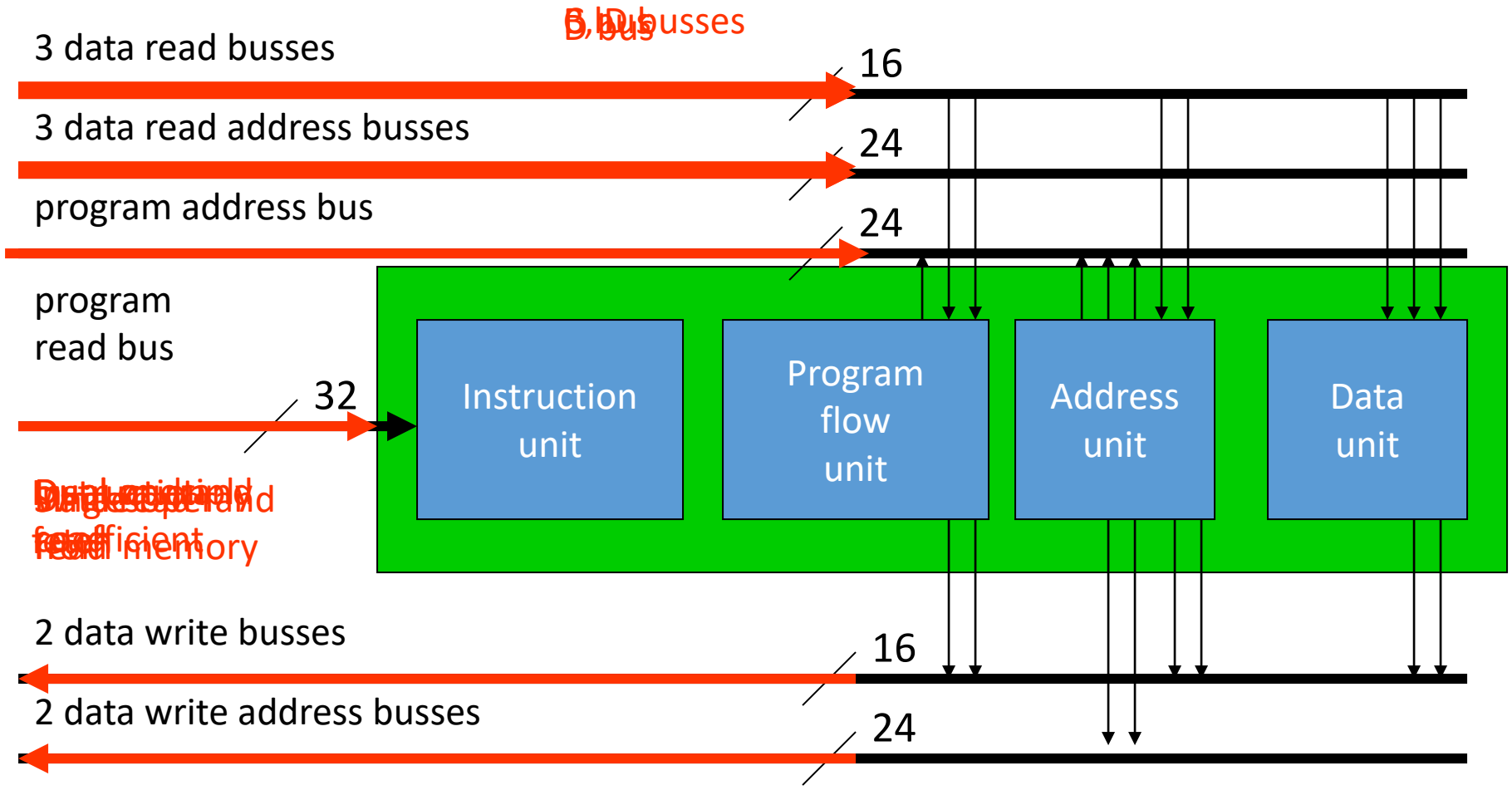
- This program sets a bit on an I/O device for a data-dependent amount of time.
- Computed goto determines number of bsfs executed.
  - len=1 -> go to len3, etc.

```
movf len, w ; get ready for computed goto
addwf pcl, f ; computed goto (PCL is low byte of
    PC)
len3: bsf x, I ; set the bit at t-3
len2: bsf x,l ; set the bit at t-2
len1: bsf x,l ; set the bit at t-1
bcf x, l ; clear the bit at t
```

C55x pipeline

- C55x has 7-stage pipe:
  - fetch;
  - decode;
  - address: computes data/branch addresses;
  - access 1: reads data;
  - access 2: finishes data read;
  - Read stage: puts operands on internal busses;
  - execute.

C55x organization



C55x pipeline hazards

- Processor structure:
  - Three computation units.
  - 14 operators.
- Can perform two operations per instruction.
- Some combinations of operators are not legal.

C55x hazards

- A-unit ALU/A-unit ALU.
- A-unit swap/A-unit swap.
- D-unit ALU,shifter,MAC/D-unit ALU,shifter,MAC
- D-unit shifter/D-unit shift, store
- D-unit shift, store/D-unit shift, store
- D-unit swap/D-unit swap
- P-unit control/P-unit control

Memory system performance

- Caches introduce indeterminacy in execution time.
  - Depends on order of execution.
- Cache miss penalty: added time due to a cache miss.

Types of cache misses

- Compulsory miss: location has not been referenced before.
- Conflict miss: two locations are fighting for the same block.
- Capacity miss: working set is too large.

CPU power consumption

- Most modern CPUs are designed with power consumption in mind to some degree.
- Power vs. energy:
  - heat depends on power consumption;
  - battery life depends on energy consumption.

CMOS power consumption

- Voltage drops: power consumption proportional to V^2.
- Toggling: more activity means more power.
- Leakage: basic circuit characteristics; can be eliminated by disconnecting power.

Power and Energy

- Power is drawn from a voltage source attached to the V_{DD} pin(s) of a chip.
- Instantaneous Power:

$$P(t) = I(t)V(t)$$

- Energy:

$$E = \int_0^T P(t)dt$$

- Average Power:

$$P_{\text{avg}} = \frac{E}{T} = \frac{1}{T}\int_0^T P(t)dt$$

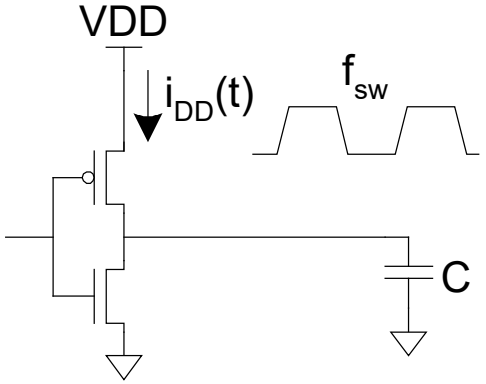
Switching Power and Activity Factor

$$P_{\text{switching}} = \frac{1}{T}\int_0^T i_{DD}(t)V_{DD}\,dt$$

$$= \frac{V_{DD}}{T}\int_0^T i_{DD}(t)\,dt$$

$$= \frac{V_{DD}}{T}\left[Tf_{\text{sw}}CV_{DD}\right]$$

$$= CV_{DD}{}^2 f_{\text{sw}}$$



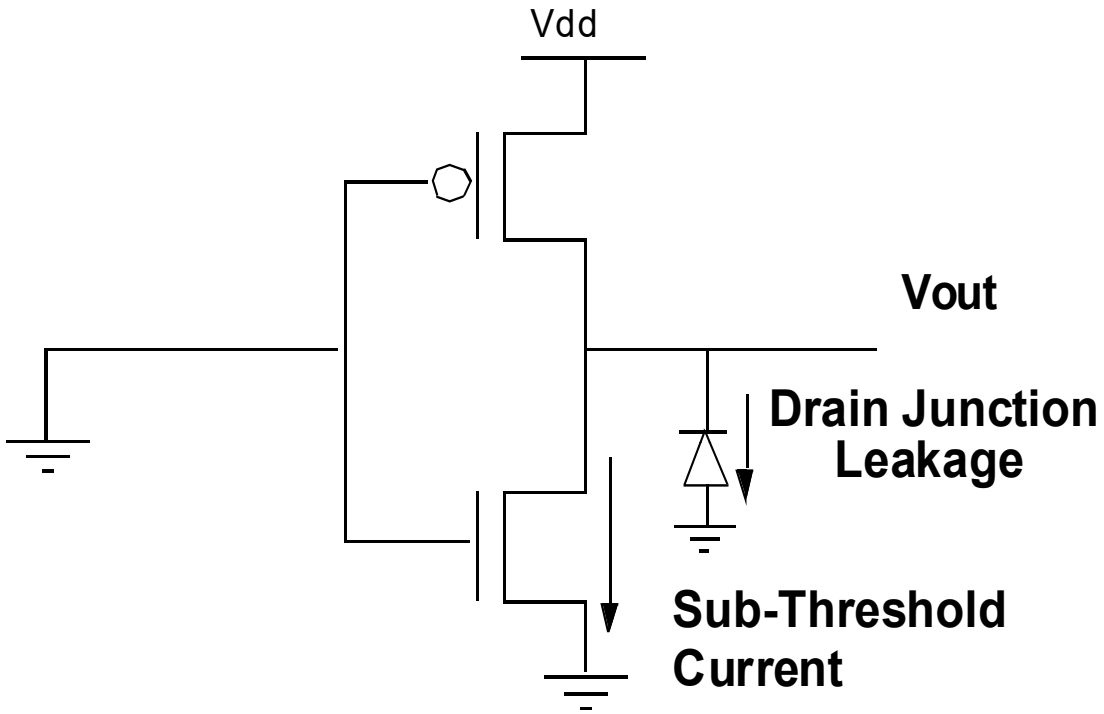
Suppose the system clock frequency = f, Let $f_{sw} = \alpha f$, where α = activity factor

If the signal is a clock, $\alpha = 1$

If the signal switches once per cycle, $\alpha = ½$

So, Dynamic power: $P_{\text{switching}} = \alpha CV_{DD}{}^2 f$

Leakage



Sub-threshold current one of most compelling issues in low-energy circuit design!

CPU power-saving strategies

- Reduce power supply voltage.
- Run at lower clock frequency.
- Disable function units with control signals when not in use.
- Disconnect parts from power supply when not in use.

C55x low power features

- Parallel execution units---longer idle shutdown times.
- Multiple data widths:
  - 16-bit ALU vs. 40-bit ALU.
- Instruction caches minimizes main memory accesses.
- Power management:
  - Function unit idle detection.
  - Memory idle detection.
  - User-configurable IDLE domains allow programmer control of what hardware is shut down.

Power management styles

- Static power management: does not depend on CPU activity.
  - Example: user-activated power-down mode.
- Dynamic power management: based on CPU activity.
  - Example: disabling off function units.

Power-down costs

- Going into a power-down mode costs:
  - time;
  - energy.
- Must determine if going into mode is worthwhile.
- Can model CPU power states with power state machine.

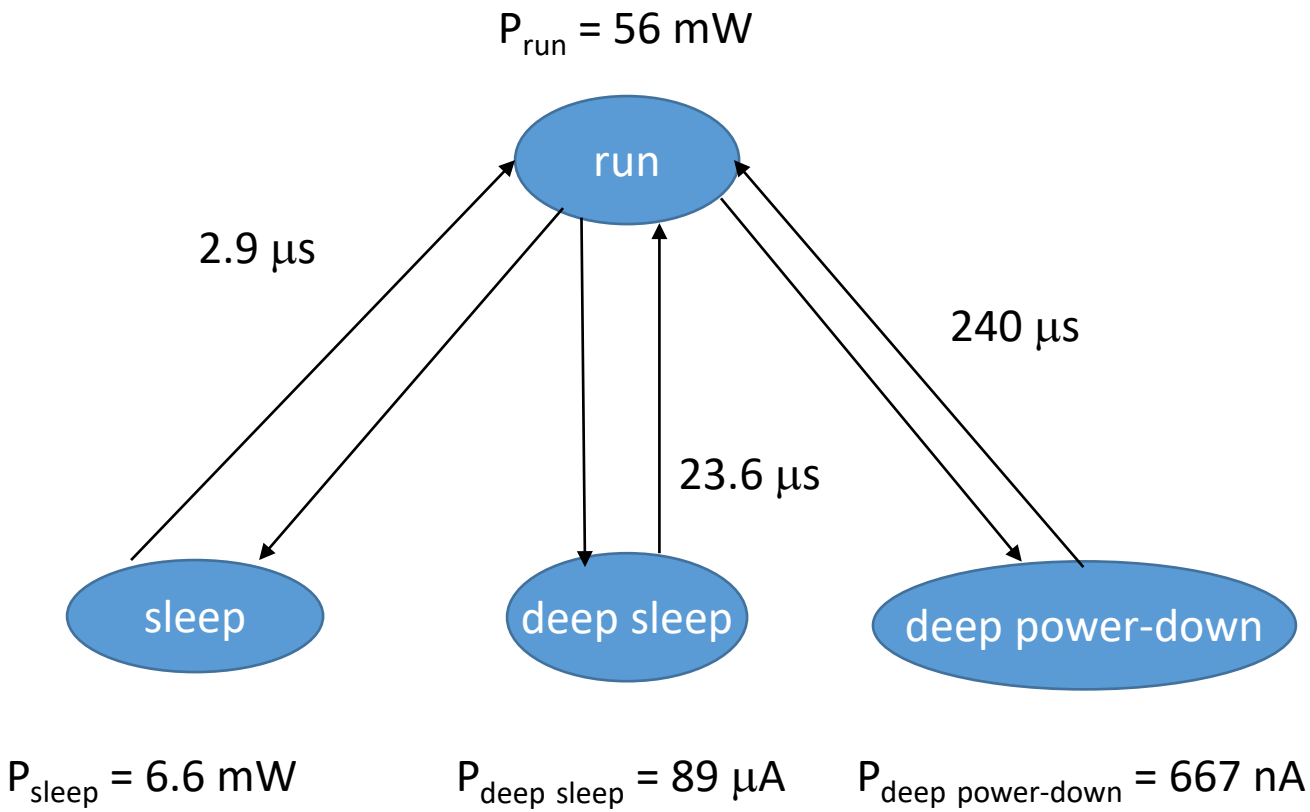
ARMv8-A standby mode

- Standby keeps power on CPU but stops or gates most clocks.
- WFI instruction waits for interrupt or debug request.
- WFE instruction waits for certain specified events.

Application: NXP LPC 1311

- Four power modes:
  - Run: normal operation.
  - Sleep: stops CPU clock, with CPU and peripheral logic still powered. Peripherals may initiate interrupts that wake up the CPU.
  - Deep sleep: stops CPU clock, powers down most analog blocks, maintains CPU and peripheral registers and SRAM values.
  - Deep power-down: power and clocks shut down to entire chip, registers and SRAM not maintained.

LPC1311 power state machine



Power management

- Different physical mechanisms require different strategies:
  - Dynamic voltage and frequency scaling (DVFS) for low leakage.
  - Race-to-dark for high leakage.
- Total energy consumption comes from dynamic and static power consumption mechanisms:
  - $E_{tot} = \int_0^T P(t)dt = \int_0^T \left[P_{dyn}(t) + P_{static}(t)\right]dt$

Two power management strategies

- DVFS used for cases when power consumption is dominated by dynamic power.
- Characteristics as a function of power supply voltage:
  - Clock frequency proportional to power supply.
  - Power consumption proportional to square of power supply.
- Reducing power supply voltage reduces performance but reduces power consumption more.
- Adjust power supply voltage to level just fast enough for current tasks.

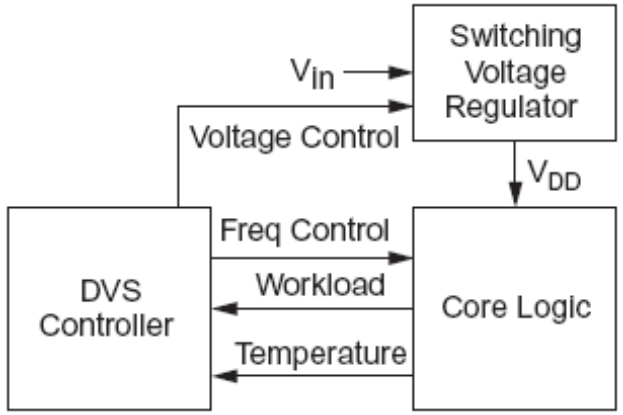
- Race-to-dark used when power consumption is dominated by static power.
- Run as fast as possible, then remove power supply from logic to eliminate static power consumption.

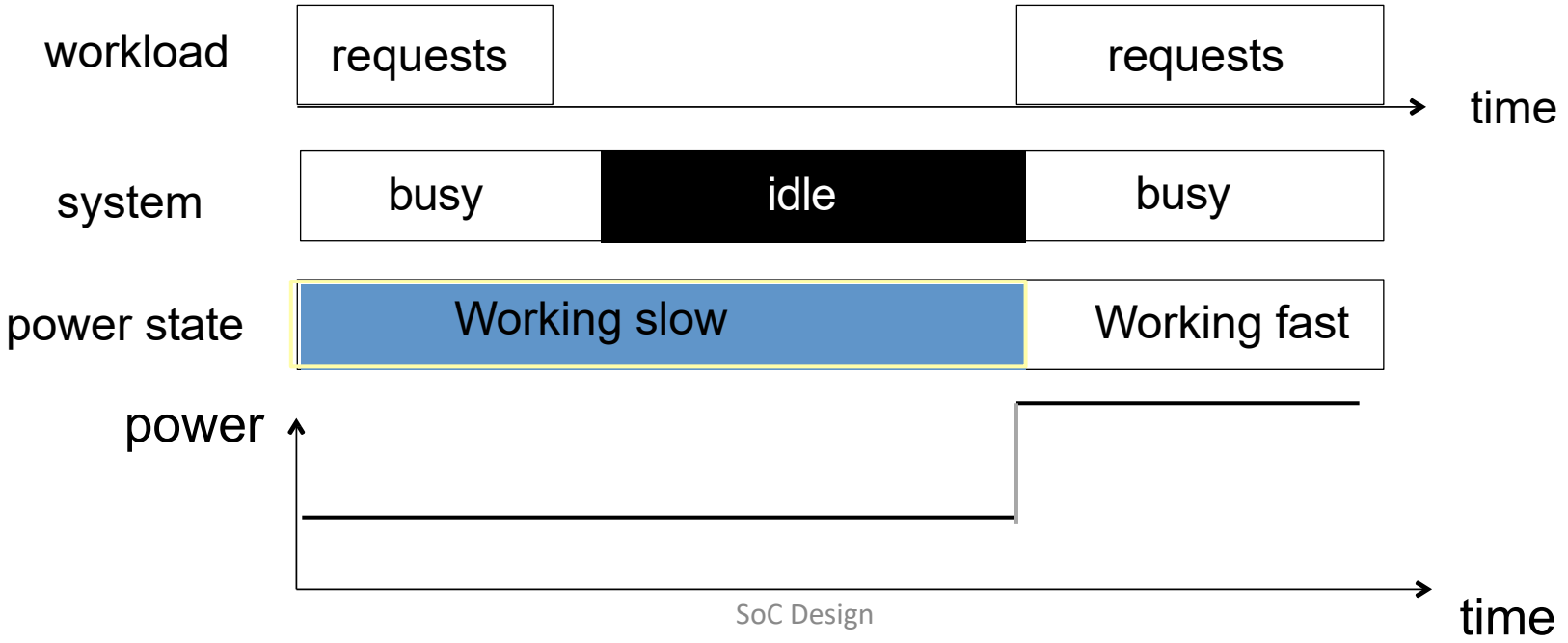
What is DVFS

- Dynamic voltage and frequency scaling
  - Circuits can work at a range of V_{dd} values
  - A given V_{dd} can support a range of clock frequencies with a
  - $F \propto (V_{dd}-V_{th})^{\chi}/V_{dd} \quad \chi \in (1.0,2.0)$
- Why DVFS saves power and energy?
  - Reduce V_{dd} to γV_{dd}
  - f_{max} reduces to roughly γ f_{max}
  - Dynamic power reduces by roughly γ^3
  - Energy reduces by roughly γ^2

Voltage / Frequency

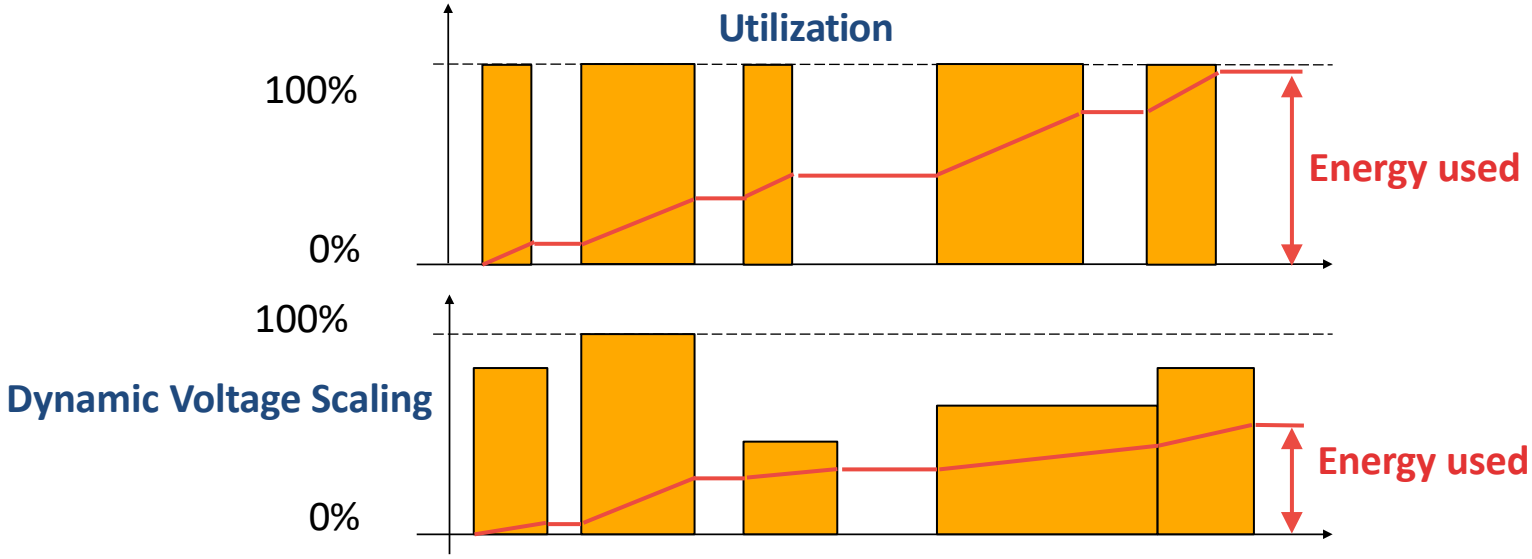
- Dynamic Voltage Scaling
  - Adjust V_{DD} and f according to workload



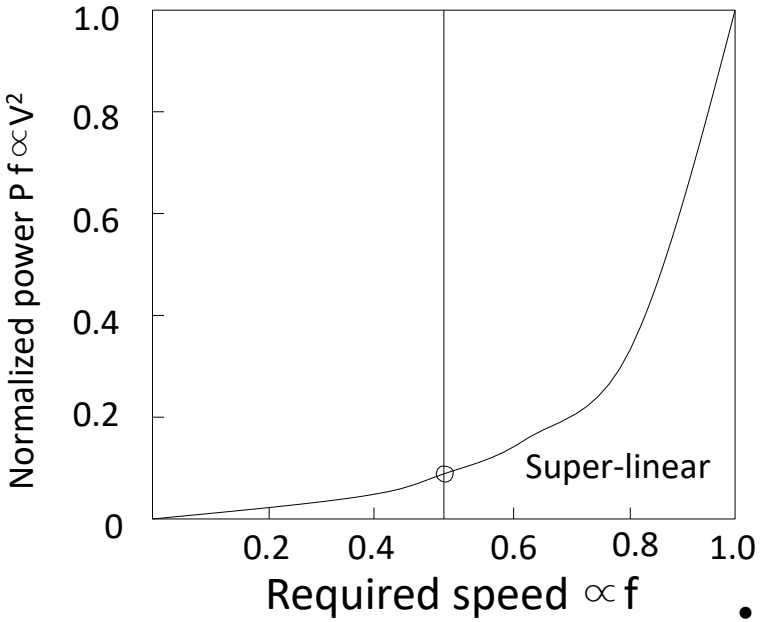


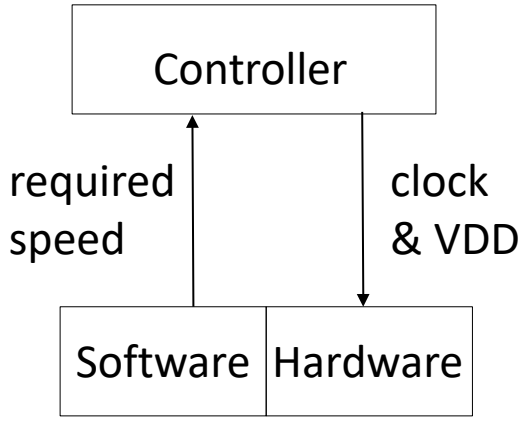
Dynamic Voltage Scaling (DVS)

- Stretch the execution by lowering the supply voltage
  - Quadratic power saving
  - Working only fast enough to meet deadlines!
- DVS algorithms can be implemented as HW or SW



Dynamic Voltage and Frequency Scaling (DVFS)



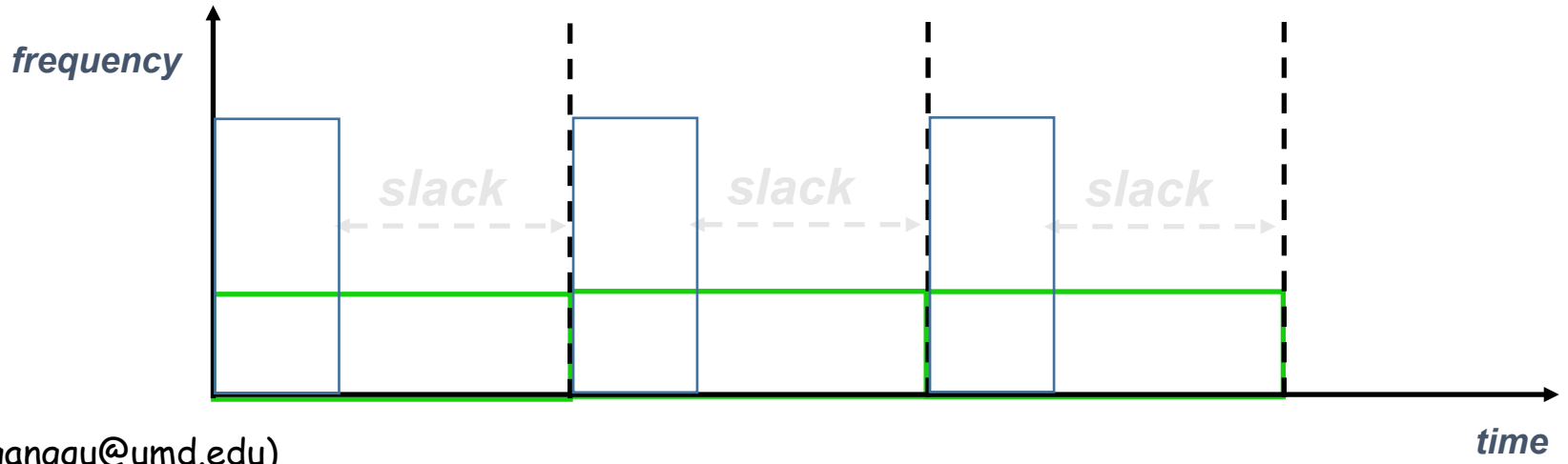


If you don't need to hustle, relax and save power

- Through SW-HW cooperation
- SW automatically predicts future workloads (by OS and application software interaction)

How DVFS Works

- Suppose that a data sample comes every 1 ms
- Requires processing time of 250 μs at 600MHz
- DVS: reduce voltage such that clock slows down to 150MHz



"Pure" research
Implementation
Applications

DVFS Roadmap

'02 Banias (Intel), 1GHz Crusoe, Subnotebook systems, ULP-PDAs, multi-processor, ultra-simple online heuristics, QoS guarantees, VLIW

'01 TM5500/5800 Crusoe, Pentium-III ULV and Celeron, μAMPS, WINS, PADS, VS and power management for PDAs, data dependent scheduling

'00 Speed Step Pentium-III, TM3120/5400 Crusoe, lpARM, speed-adaptive microprocessor, IBM, Toshiba, Fujitsu, NEC, PicoRadio, voltage hopping, rate-based, priority, real-time

'99 QoS-Energy tradeoff, application-specified scheduling

'98 Variable voltage core-based design, Toshiba's Ultra Low-Power MPEG4 Codec, multiple voltage scheduling, feasible DVS scheduling

'97 DC-DC switching regulators, multiple voltage scheduling

'96 Dataflow graph scheduling for low power

'95 Clustered VS, multiple voltage and ideal variable speed CPU scheduling

@ Gang Qu (gangqu@umd.edu)

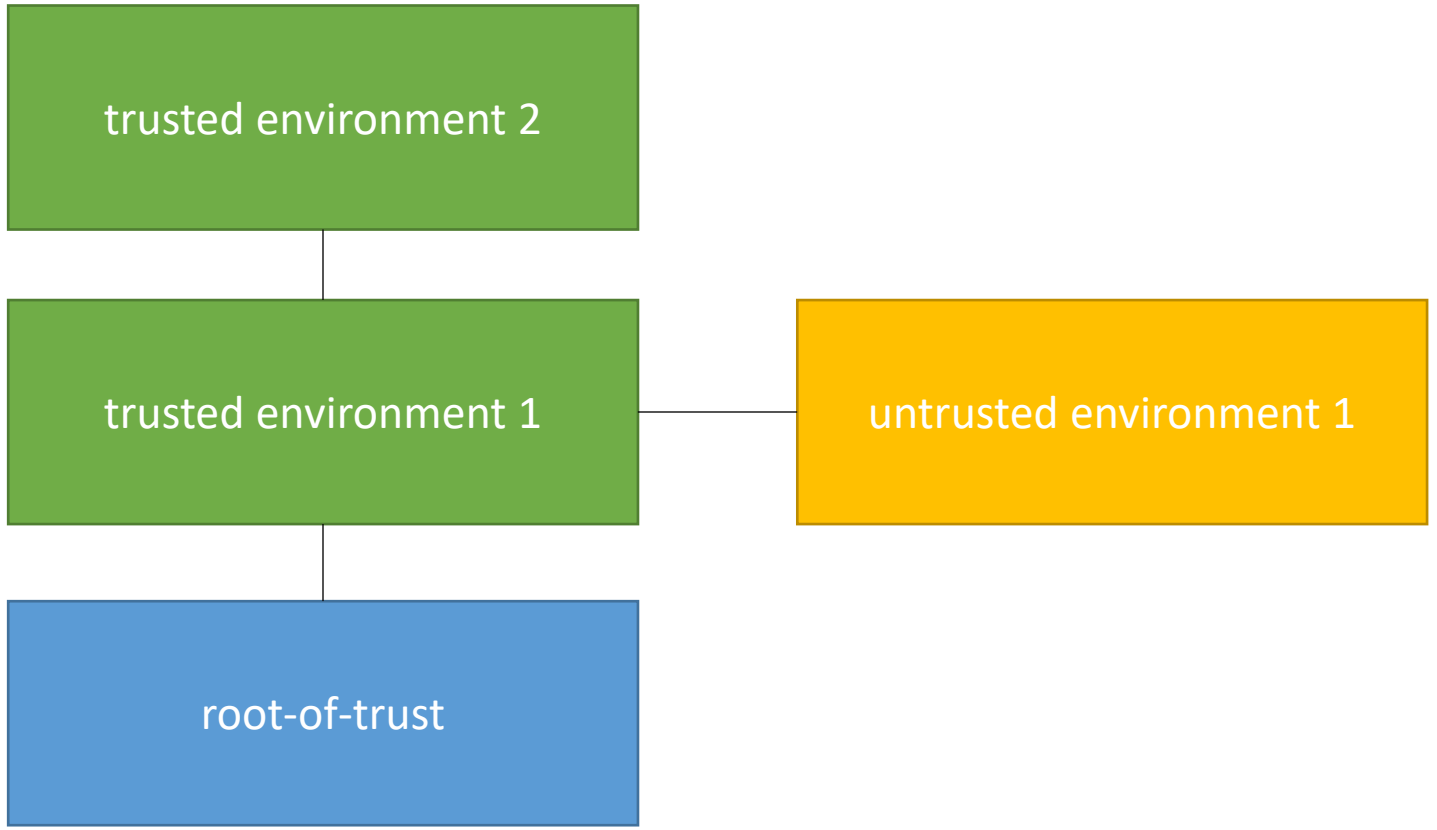
Safety and security

- Programs that perform important operations should run in trusted mode.
- Supervisor mode, memory management provide some level of trust but does not ensure that the code comes from a trusted source.
- Testing a digital signature for software before executing ensures that the software comes from a trusted source.

Trusted programs and root of trust

- Checking a digital signature for code requires a public key.
  - Public key must come from a trusted source.
- A trusted program may rely on another trusted program, etc.
  - New program's level of trustworthiness depends on the previous program's trustworthiness.
- Trust must eventually be traced back to a root of trust that provides known trust.
- Root of trust can be provided with a hardware key that cannot be altered.

Chain-of-trust



ARM Trusted Execution Environment

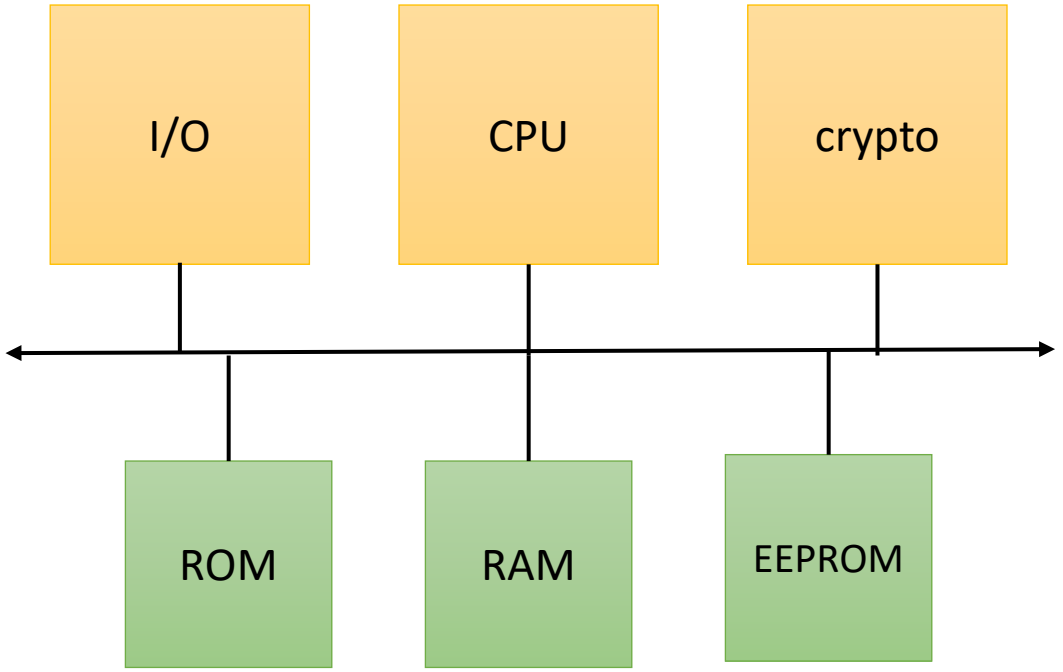
- Four-compartment model:
  - Normal world includes user and system modes.
  - Hypervisor mode supports virtual machines.
  - Trusted World partitions into secure and non-secure components.
  - SecurCore provides physically separate chips for additional protection.
- Trusted Execution Environment (TEE) provides secure execution mode.

Secure operation according to ARM

- Processor security mechanisms should limit system access.
- DMA should provide secure access for memory and I/O.
- Access to CPU from other parts of system should be limited.
- Trusted software should be able to identify and control processes that consume resources in the non-trusted world.
- JTAG etc. should not compromise security.
- I/O devices should support secure/non-secure operation.
- Platform should have a unique, non-modifiable key.
- Platform should provide secure trusted private key storage.
- Process and update should be secure and verifiable.
- Secure execution should ensure that control flow cannot be subverted.
- Platform should provide security primitives.

Smart cards

- Smart cards are used for financial, medical, other types of secure information.
  - Must operate at very low energy levels and very low cost.
- Runs from extneral power supply.
- Some code held in ROM.
- Permanent storage in non-volatile EEPROM.
- Cryptography engine provides low power operation, trusted functions.



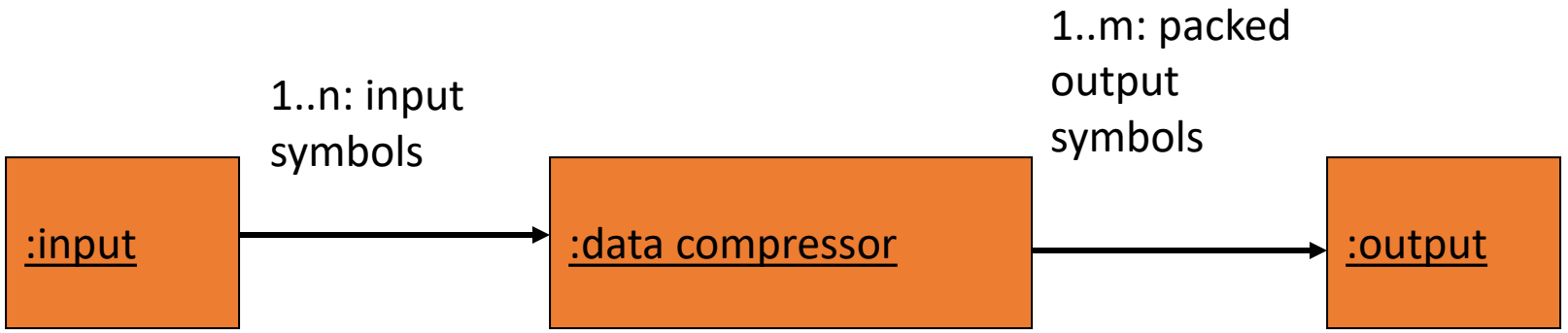
CPUs

- Example: data compressor.

Goals

- Compress data transmitted over serial line.
  - Receives byte-size input symbols.
  - Produces output symbols packed into bytes.
- Will build software module only here.

Collaboration diagram for compressor

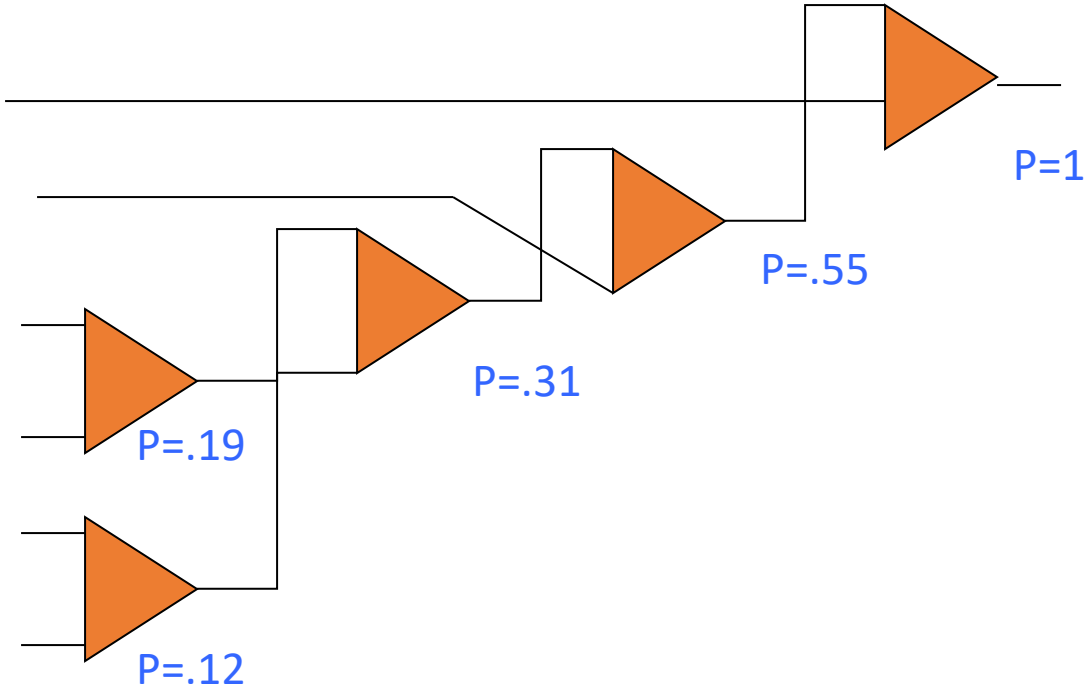


Huffman coding

- Early statistical text compression algorithm.
- Select non-uniform size codes.
  - Use shorter codes for more common symbols.
  - Use longer codes for less common symbols.
- To allow decoding, codes must have unique prefixes.
  - No code can be a prefix of a longer valid code.

Huffman example

character	P
a	.45
b	.24
c	.11
d	.08
e	.07
f	.05



Example Huffman code

- Read code from root to leaves:

a	1
b	01
c	0000
d	0001
e	0010
f	0011

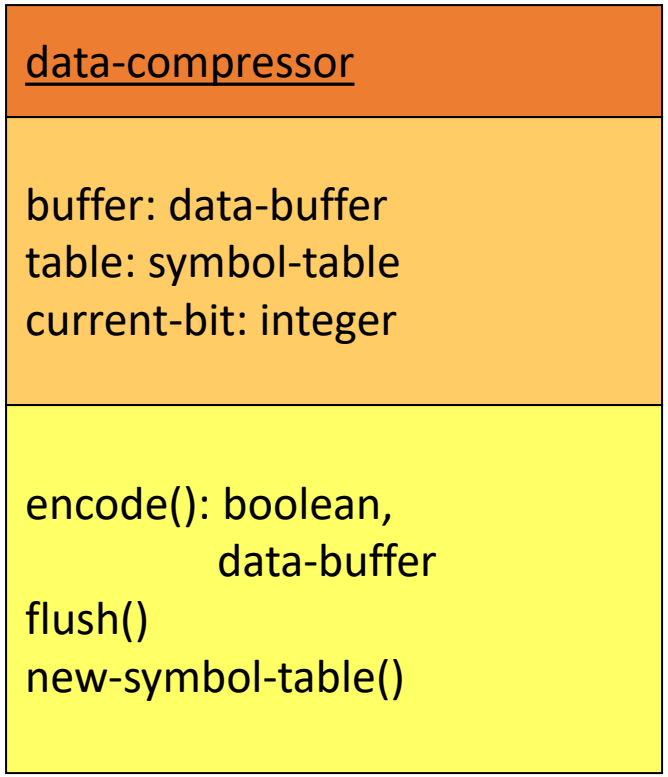
Huffman coder requirements table

name	data compression module
purpose	code module for Huffman compression
inputs	encoding table, uncoded byte-size inputs
outputs	packed compression output symbols
functions	Huffman coding
performance	fast
manufacturing cost	N/A
power	N/A
physical size/weight	N/A

Building a specification

- Collaboration diagram shows only steady-state input/output.
- A real system must:
  - Accept an encoding table.
  - Allow a system reset that flushes the compression buffer.

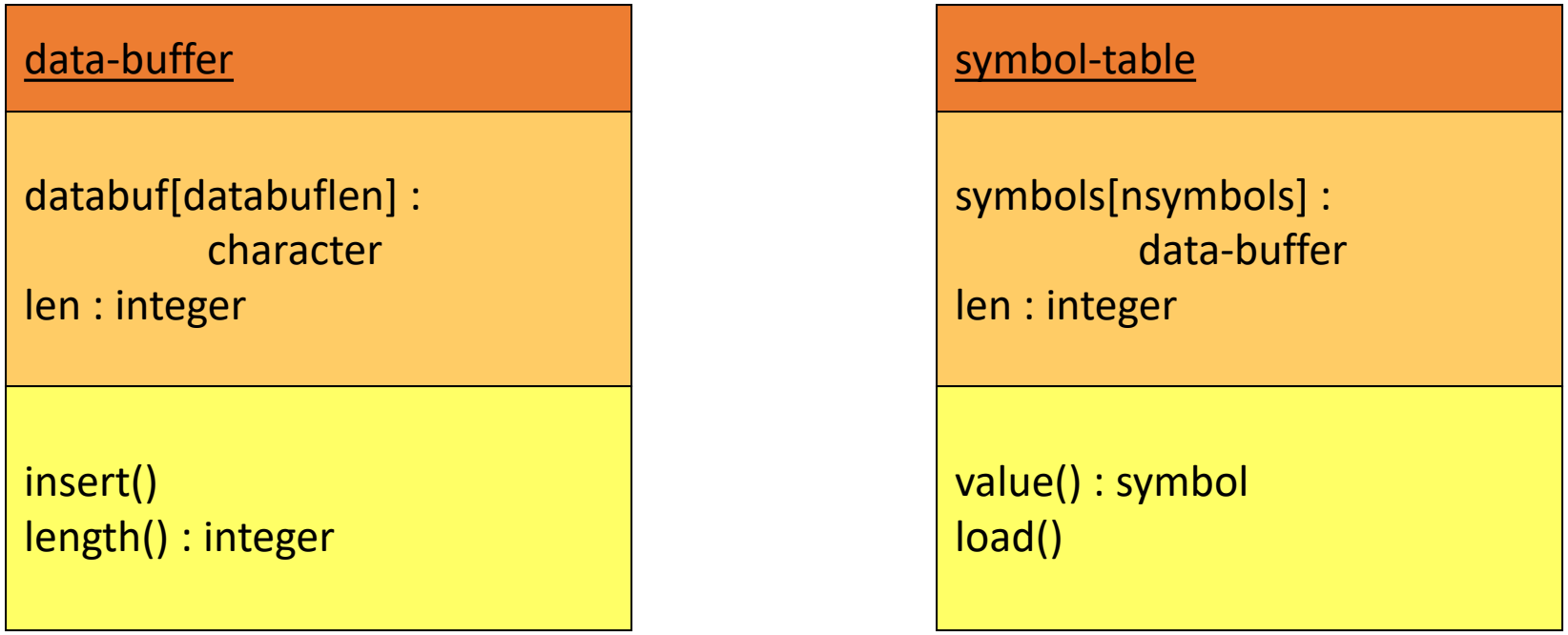
data-compressor class



data-compressor behaviors

- encode: Takes one-byte input, generates packed encoded symbols and a Boolean indicating whether the buffer is full.
- new-symbol-table: installs new symbol table in object, throws away old table.
- flush: returns current state of buffer, including number of valid bits in buffer.

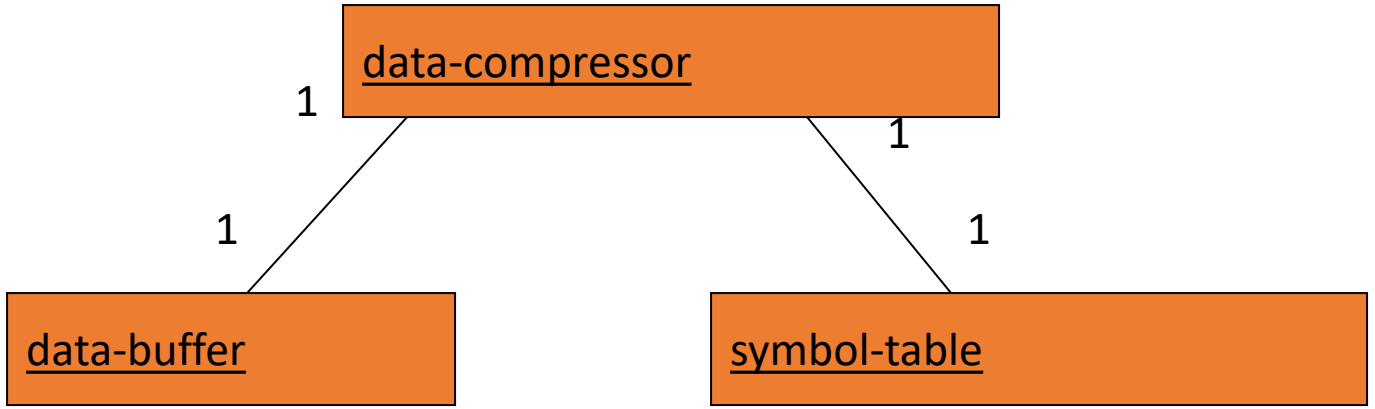
Auxiliary classes



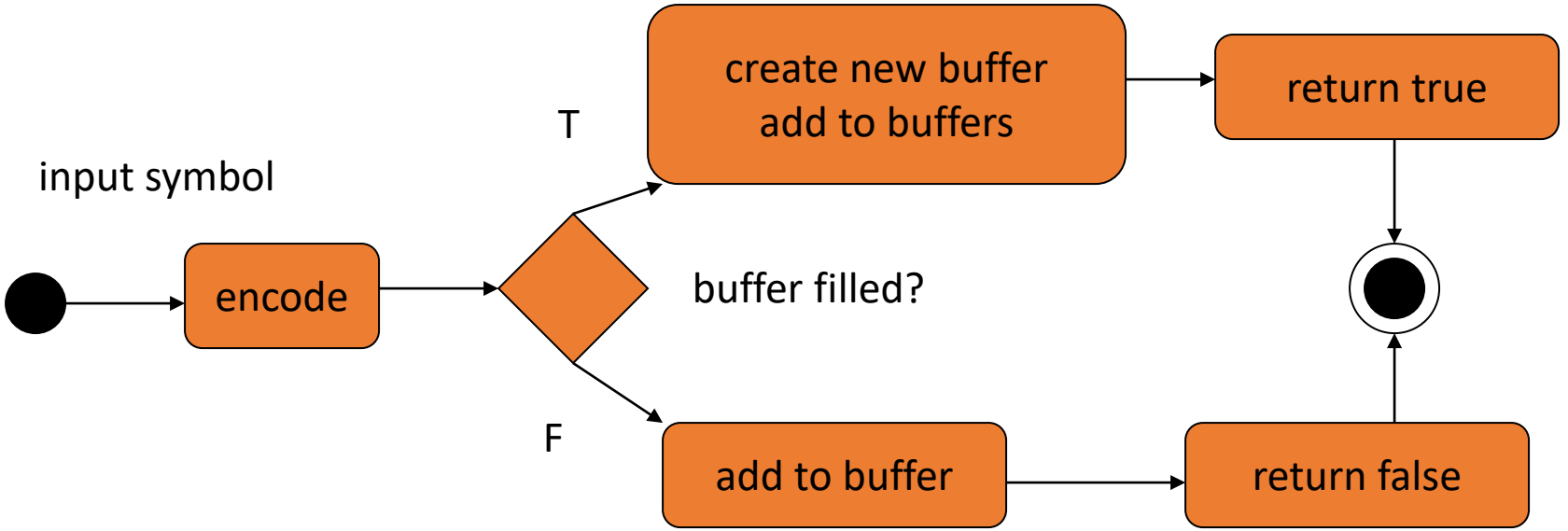
Auxiliary class roles

- data-buffer holds both packed and unpacked symbols.
  - Longest Huffman code for 8-bit inputs is 256 bits.
- symbol-table indexes encoded verison of each symbol.
  - load() puts data in a new symbol table.

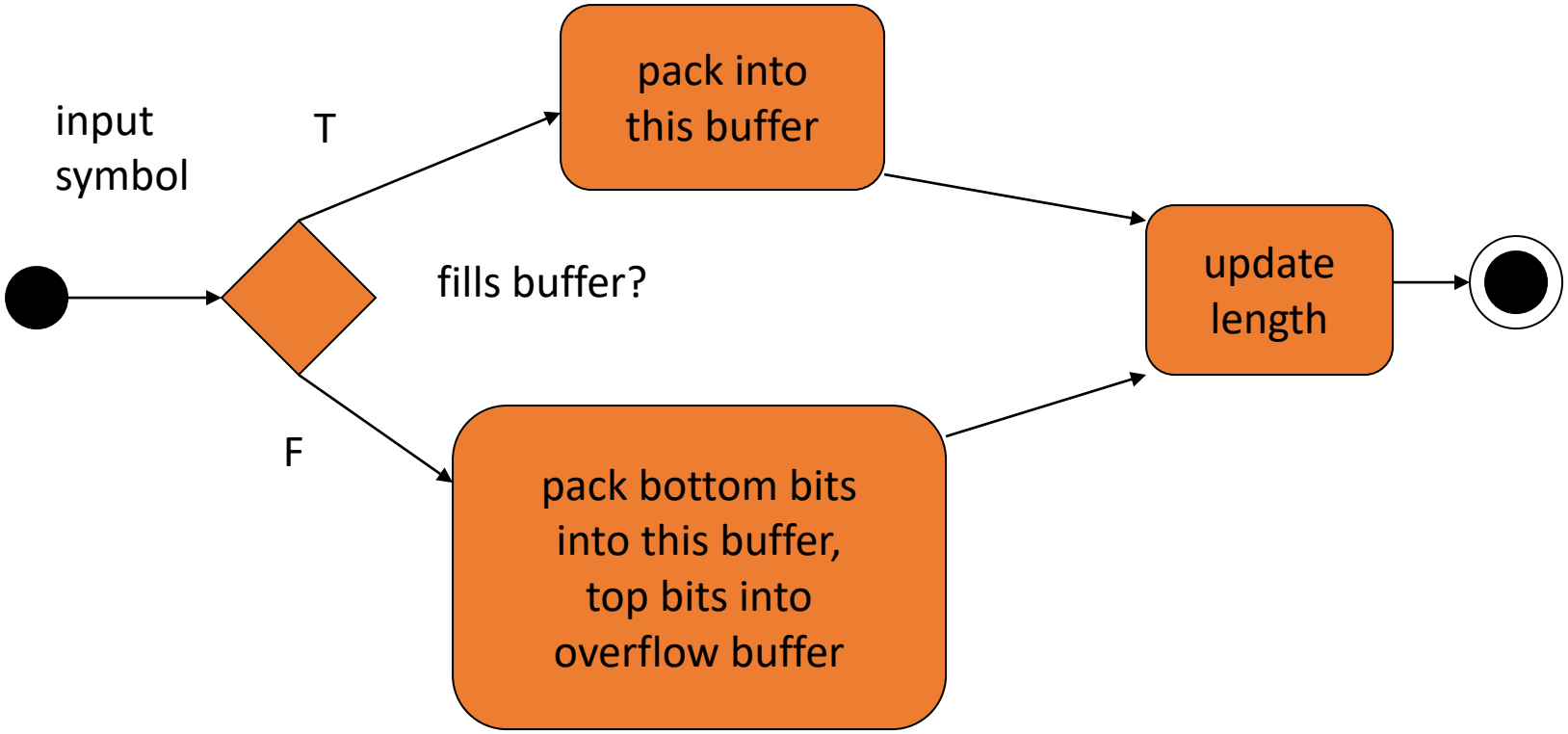
Class relationships



Encode behavior



Insert behavior



Program design

- In an object-oriented language, we can reflect the UML specification in the code more directly.
- In a non-object-oriented language, we must either:
  - add code to provide object-oriented features;
  - diverge from the specification structure.

C++ classes

```
class data_buffer {
   char databuf[databuflen];
   int len;
   int length_in_chars() { return len/bitsperbyte; }
public:
   void insert(data_buffer,data_buffer&);
   int length() { return len; }
   int length_in_bytes() { return (int)ceil(len/8.0); }
   int initialize();
   …
```

C++ classes, cont’d.

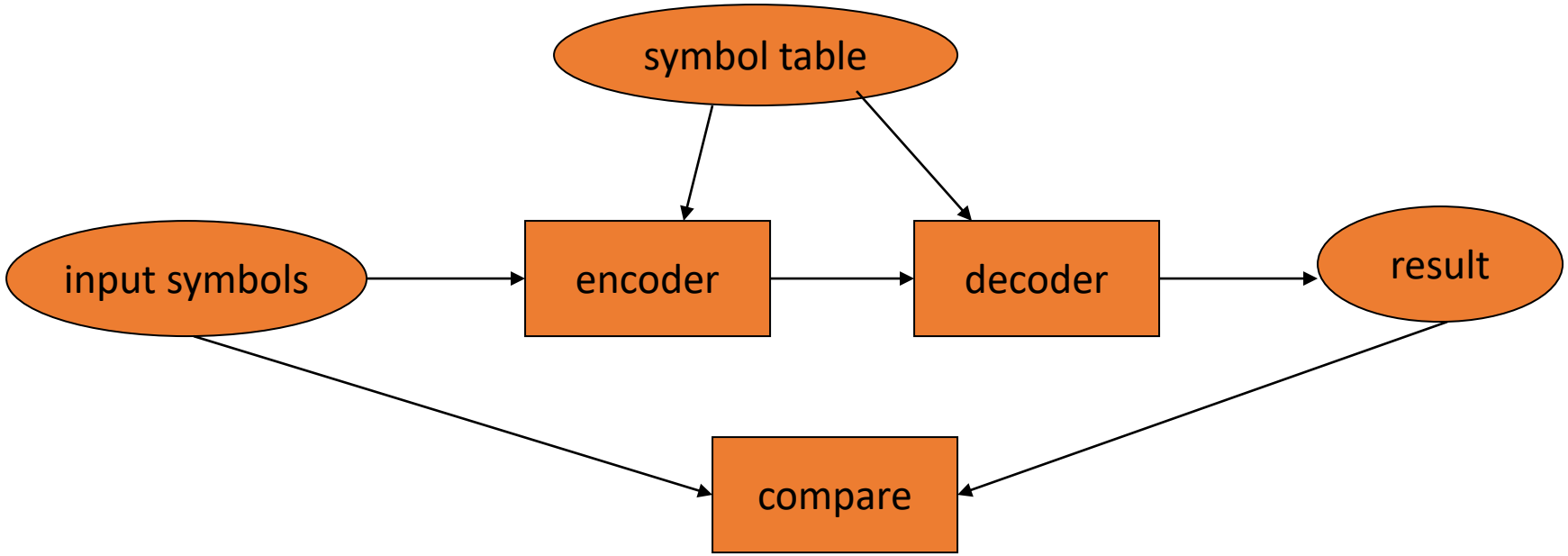
```
class data_compressor {
   data_buffer buffer;
   int current_bit;
   symbol_table table;
public:
   boolean encode(char,data_buffer&);
   void new_symbol_table(symbol_table);
   int flush(data_buffer&);
   data_compressor();
   ~data_compressor();
   }
```

C code

```
struct data_compressor_struct {
   data_buffer buffer;
   int current_bit;
   sym_table table;
}
typedef struct data_compressor_struct data_compressor,
   *data_compressor_ptr;
boolean data_compressor_encode(data_compressor_ptr mycmptrs, char isymbol,
   data_buffer *fullbuf) ...
```

Testing

- Test by encoding, then decoding:



Code inspection tests

- Look at the code for potential problems:
  - Can we run past end of symbol table?
  - What happens when the next symbol does not fill the buffer? Does fill it?
  - Do very long encoded symbols work properly? Very short symbols?
  - Does flush() work properly?