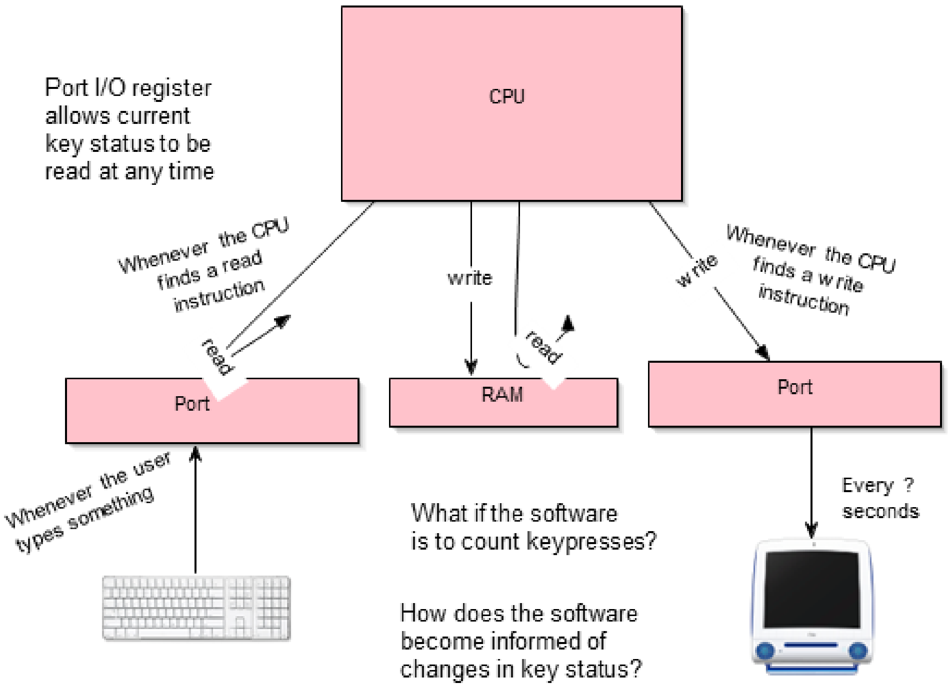
I/O Synchronization



- Embedded systems are said to be "event-driven"
  - i.e., the primary function is to respond to "events".
- But how does the program become aware of changes in the system's environment? How does it respond to events?
- Two approaches
  - Status driven using polling (busy waiting)
  - Interrupts driven

Polling (Busy Waiting)

- The program executes an infinite loop that tests each possible event to which the system must respond.
- Straightforward in design and programming.
  - Used in systems with relatively loose response time requirements.
  - When the response time is quicker than using interrupts.
  - When large amounts of data are expected to arrive at particular intervals,

- In the status driven model, the CPU polls the status registers until a change occurs.

```
int old = KEY_STATUS_REG;
int val = old;
while(old==val){
     val = KEY_STATUS_REG;
}
//status changed
```

- It can be used to define functions that make input look like reading variables (reading from memory!)

```
char getchar(){
     while(KEY_STATUS_REG & PRESSED);
     while(!(KEY_STATUS_REG & PRESSED));
     return KEY_VALUE_REG;
}
```

Polling (Busy Waiting)

```
int old = KEY_STATUS_REG;
int val = old;
while(old==val){
    val = KEY_STATUS_REG;
}
//status changed
```

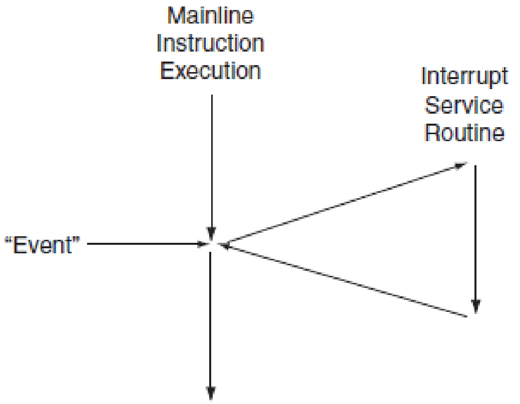
What if KEY_STATUS_REG were an ordinary variable?

- Problems
  - The CPU is busy but is doing nothing useful!
  - The CPU has no control over and uncertain when to exit the loop!
  - More problematic if there are too many I/O devices to check.
    - Time required to poll them can be considerable.

Interrupts

- Is an asynchronous electrical signal from a peripheral to the processor.
- Can be generated from peripherals external or internal to the processor, as well as by software.
- The idea of the interrupt is that:

1. the occurrence of an event "interrupts" the current flow of instruction execution, and
2. invokes another stream of instructions, called the interrupt service routine (ISR) or interrupt handler, that services the event.
3. When the ISR completes, the processor returns to the work that was interrupted.

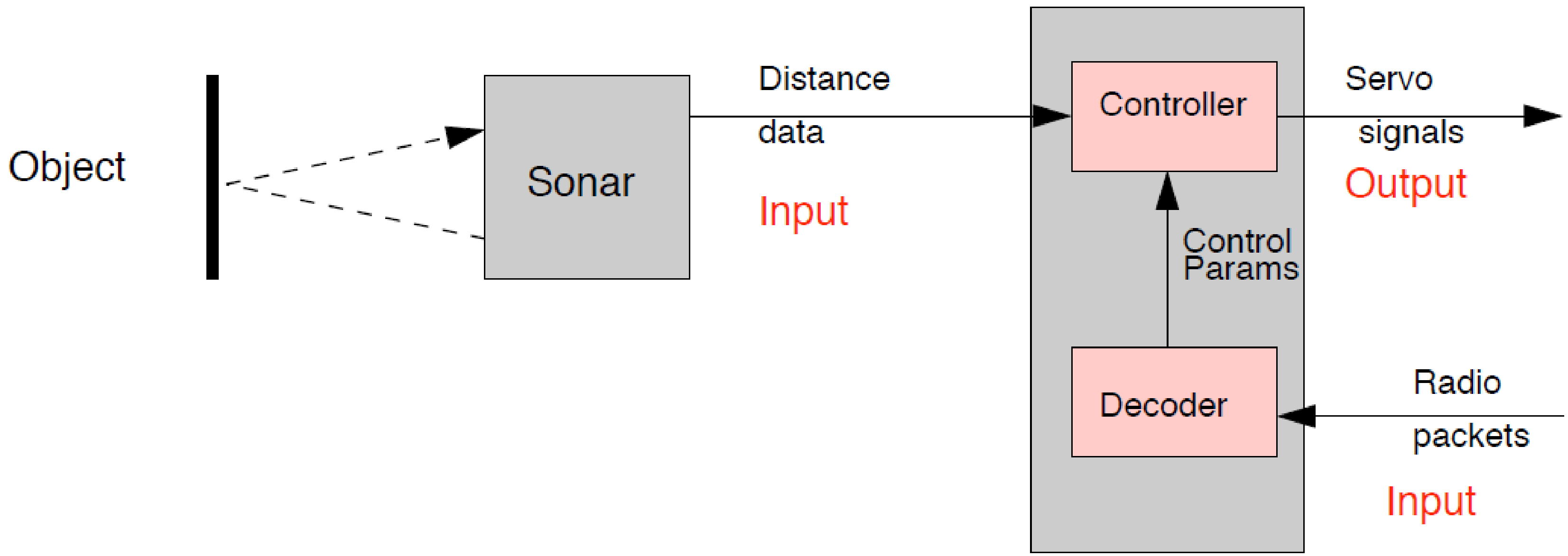


- The processor is able to use a larger percentage of its waiting time performing useful work.

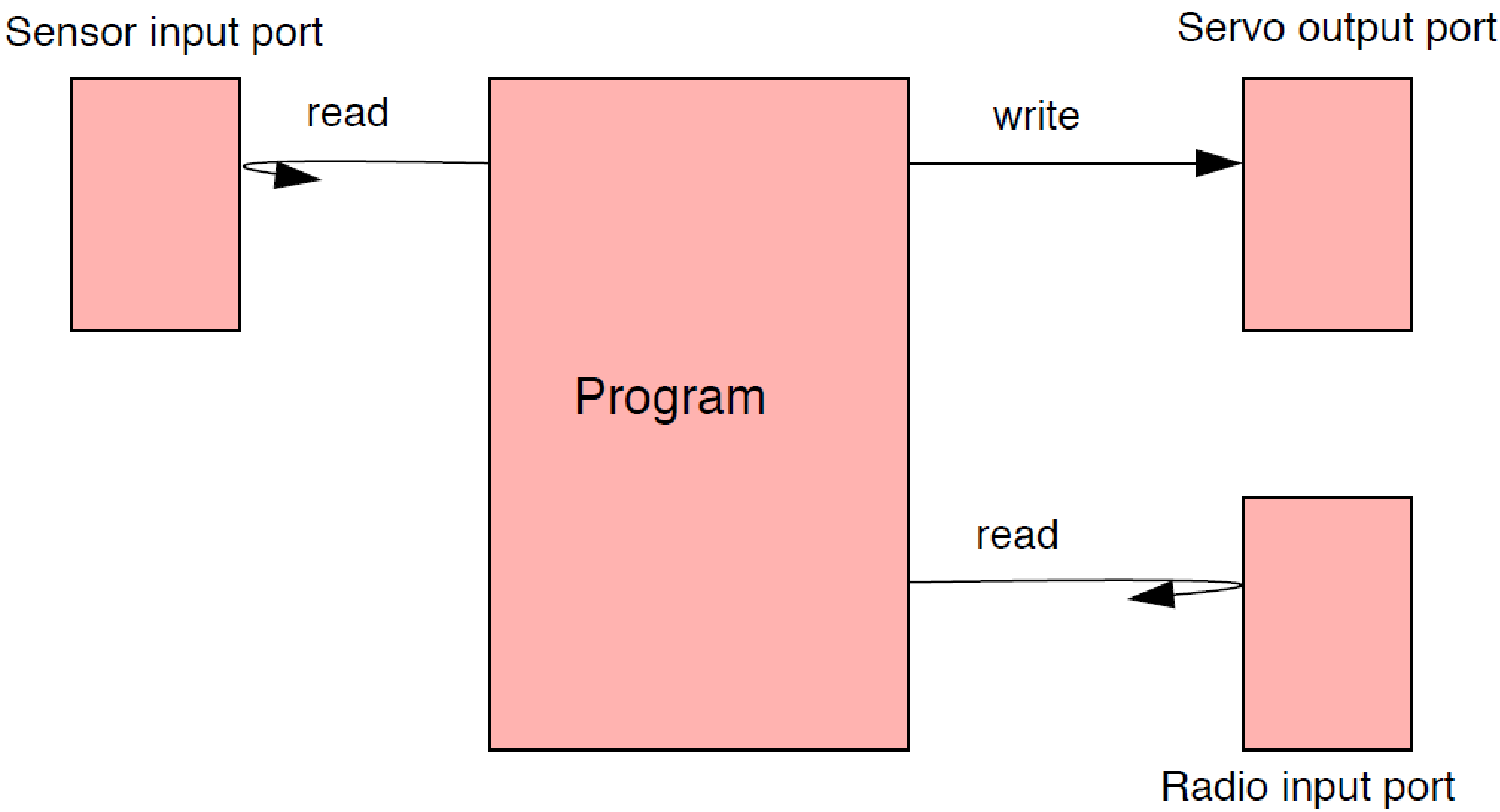
- However, there is some time overhead associated with each interrupt.
  - To put aside the processor's current work and transfer control to the interrupt service routine.
  - Many of the processor's registers must be saved in memory.
- Interrupts are used when efficiency is a requirement or when multiple devices must be monitored simultaneously.

A simple embedded system

1. Follow (track) an object using sonar echoes.
2. Control parameters are sent over wireless.
3. The servo controls wheels.



The view from the processor



How do we structure the software?

The program

- We will go through a series of attempts to organize the program leading to the need for threads.
- We will discuss new problems that arise because of programming with threads.
- Implementing threads.

The program: a first attempt

input

```
int sonar_read(){
      while(SONAR_STATUS & READY == 0);
      return SONAR_DATA;
}
```

polling

```
void radio_read(struct Packet *pkt){
      while(RADIO_STATUS & READY == 0);
      pkt->v1 = RADIO_DATA1;
      ...
      pkt->vn = RADIO_DATAn;
}
```

polling

```
// Assuming that status is automatically reset when data is read.
```

output

```
void servo_write(int sig){
      SERVO_DATA = sig;
}
```

```
main(){
      struct Params params;
      struct Packet packet;
      int dist, signal;
      while(1){
            dist = sonar_read();
            control(dist, &signal, &params);
            servo_write(signal);
            radio_read(&packet);
            decode(&packet,&params);
      }
}
```

Problem?

```
void control(int dist, int *sig, struct Params *p);
```

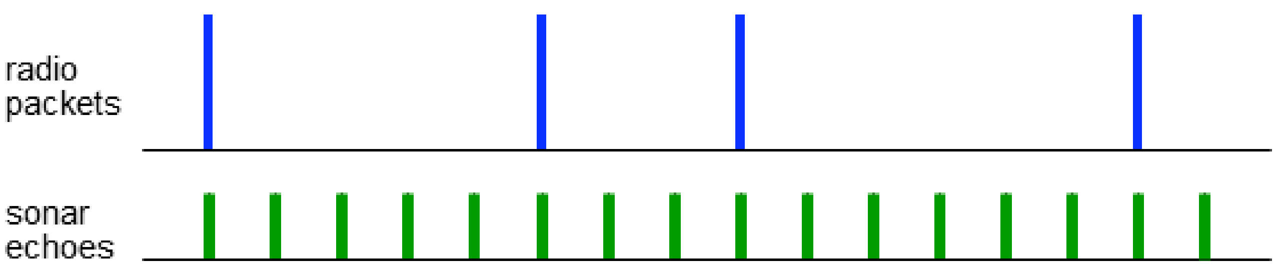
- Computes a servo signal on basis of its internal state, the given distance, and a set of control parameters

```
void decode(struct Packet *pkt, struct Params *p)
```

- Decodes a packet and calculates new control parameters

The program: a first attempt (cont.)

- Problem
  - Unknown and unrelated frequencies of events.
    - cannot know in advance which will come
- Consequence
  - busy-wait loops in alternating order is clearly ad hoc
    - Ignoring the other event while busy waiting!




```
int sonar_read(){
    while(SONAR_STATUS & READY == 0);
    return SONAR_DATA;
}
void radio_read(struct Packet *pkt){
    while(RADIO_STATUS & READY == 0);
    pkt->v1 = RADIO_DATA1;
    ...
    pkt->vn = RADIO_DATAn;
}
void servo_write(int sig){
    SERVO_DATA = sig;
}
main(){
    struct Params params;
    struct Packet packet;
    int dist, signal;
    while(1){
        dist = sonar_read();
        control(dist, &signal, &params);
        servo_write(signal);
        radio_read(&packet);
        decode(&packet,&params);
    }
}
```

The program: a second attempt

output

```
void servo_write(int sig){
    SERVO_DATA = sig;
}
main(){
    struct Params params;
    struct Packet pa
    int dist,
    while(1){
        if(SONAR_STATUS & READY) {
            dist = SONAR_DATA;
            control(dist,&signal,&params);
            servo_write(signal);
        }
        if(RADIO_STATUS & READY){
            packet->v1 = RADIO_DATA1;
            ...;
            packet->vn = RADIO_DATAn;
            decode(&packet,&params);
        }
    }
}
```

Centralized polling

- Remove the functions for reading and merge the busy-wait, i.e. have only one busy waiting loop!

Problem?

Centralized polling (busy waiting)

- Breaking modularity
  - Checking both events in one big busy-waiting loop.
  - Complicating the simple read operations.
    - Need to rewrite the loop if a 3rd input is added.
- Not efficient
  - 100% CPU usage, no matter how frequent input data arrives.
- How to make the main loop run less often?

The program: a third attempt

output

```
void servo_write(int sig){
     SERVO_DATA = sig;
}
main(){
     struct Params params;
     struct Packet packet;
     int dist, signal;
     while(1){
          sleep_until_next_timer_interrupt();
          if(SONAR_STATUS & READY) {
               dist = SONAR_DATA;
               control(dist,&signal,&params);
               servo_write(signal);
          }
          if(RADIO_STATUS & READY){
               packet->v1 = RADIO_DATA1;
               ...;
               packet->vn = RADIO_DATAn;
               decode(&packet,&params);
          }
     }
}
```

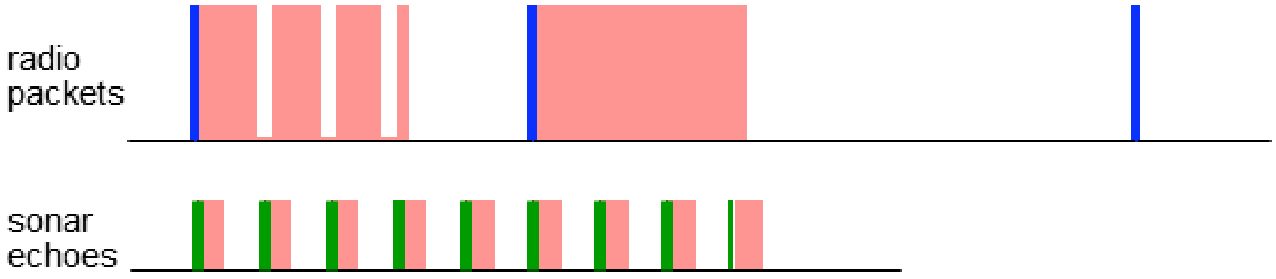
Cyclic executive

- Offline **scheduling**
- Execute tasks in sequence at a fixed rate in a big loop
  - Possibility to limit certain tasks to every N turns of the loop only
- Waits for a periodic interrupt for synchronization
  - The timer period must be set to trade power consumption against task response!
- Loops the execution of routines/procedures

Problem?

The program: a third attempt (cont.)

- Problems
  - If processing time for the infrequent radio packets is much longer than for the frequent sonar echoes …



  - You must construct the "scheduler" manually.
    - For each routines/procedures
      - Period and computation time
  - Many more…

Concurrency

- I.e. "running together."
  - Non-concurrent programs specify a sequence of instructions to execute, i.e. a sequential program has a single thread of control.
  - A system is said to be **concurrent** if different parts of the system (components) **conceptually** operate at the same time, i.e., a concurrent program has multiple threads of control executing simultaneously.
  - A program is said to be **parallel** if different parts of the program **physically** execute simultaneously on distinct hardware (such as on multicore machines, on servers in a server farm, or on distinct microprocessors).
- Why concurrent execution?
  - Improve responsiveness
  - Improve performance
  - Directly control the timing of external interactions

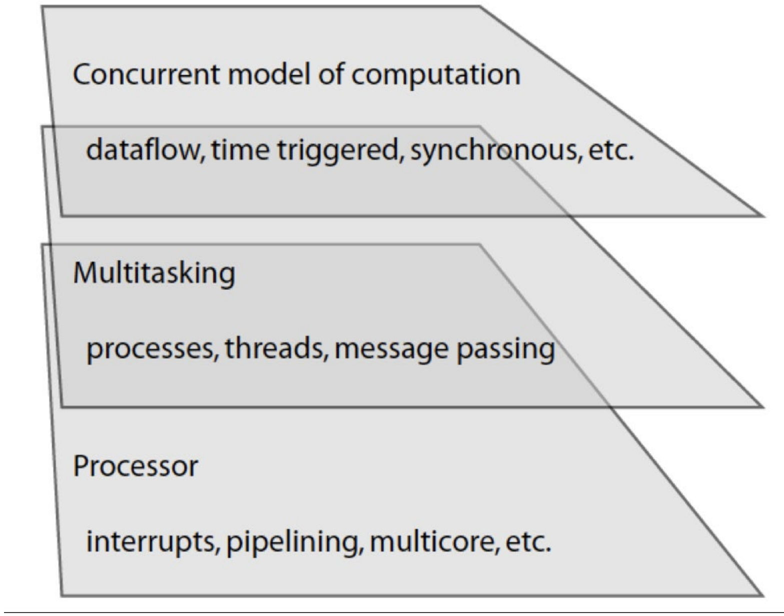
Concurrency (cont.)

- Improve **responsiveness**
  - Avoid situations where long-running programs can block a program that responds to external stimuli (e.g. sensor data or a user request).
  - Improved responsiveness reduces latency.
- Improve **performance**
  - by allowing a program to run simultaneously on multiple processors or cores.
- Directly **control the timing** of external interactions.
  - A program may need to perform some action, such as updating a display, at particular times, regardless of what other tasks might be executing at that time.

Concurrency (cont.)

- Layers of Abstraction

- Concurrent programs can be executed sequentially or in parallel.
- Sequential execution of a concurrent program is **multitasking**
  - mid-level techniques
  - implemented using the low-level mechanisms
  - supporting concurrent execution of multiple tasks by interleaving their execution in a single sequential stream of instructions.



The program: a concurrent attempt

- We could solve (in a rather ad-hoc way) how to wait concurrently.
- Now we need to express concurrent execution ..

Imagine . . .

- . . . that we could interrupt (the right term is **preempt**) execution of packet decoding when a sonar echo arrives so that the control algorithm can be run. Then decoding could resume!
- The two tasks fragments are **interleaved**.

The program: interleaving by hand

```
int sonar_read(){
    while(SONAR_STATUS & READY == 0);
    return SONAR_DATA;
}
void radio_read(struct Packet *pkt){
    while(RADIO_STATUS & READY == 0);
    pkt->v1 = RADIO_DATA1;
    ...
    pkt->vn = RADIO_DATAn;
}
void servo_write(int sig){
    SERVO_DATA = sig;
}
main(){
    struct Params params;
    struct Packet packet;
    int dist, signal;
    while(1){
        dist = sonar_read();
        control(dist, &signal, &params);
        servo_write(signal);
        radio_read(&packet);
        decode(&packet,&params);
    }
}
```

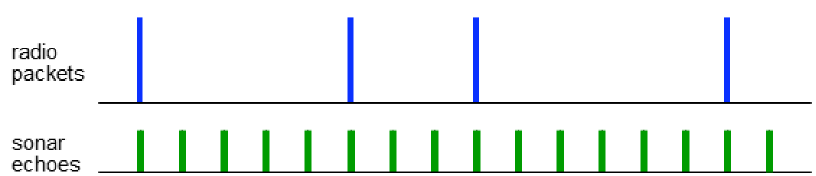
```
void decode(struct Packet *pkt, struct Params p){
    phase1(pkt,p);
    try_sonar_task();
    phase2(pkt,p);
    try_sonar_task();
    phase3(pkt,p);
}
void try_sonar_task(){
    if(SONAR_STATUS & READY){
        dist = SONAR_DATA;
        control(dist,&signal,&params);
        servo_write(signal);
    }
}
```

Seizing control and allowing for other tasks to take over:

- interleaving task fragments.

Run try_sonar_task sufficiently often.

- Again, we break the logical organization of the program in an ad-hoc way!
- How many phases of decode will we need to run the sonar often enough?



The program: interleaving by hand

- More fine breaking up might be needed ... to run `try_sonar_task` more often

```
void phase2(struct Packet *pkt, struct Params *p){
	while(expr){
		try_sonar_task();
		phase21(pkt,p);
	}
}
```

- Even more fine breaking up might be needed ... to run `try_sonar_task at every` 800:th iteration

```
void phase2(struct Packet *pkt, struct Params *p){
	int i = 0;
	while(expr){
		if(i%800==0) try_sonar_task();
		i++;
		phase21(pkt,p);
	}
}
```

- Code can become very unstructured and complicated very soon.
- And then someone might come up with a new, better decoding algorithm ...
- Moreover, what if the control algorithm must be broken up too?

HALMSTAD UNIVERSITY

Automatic interleaving?
Low-level concurrency

- There are 2 tasks, driven by independent input sources.

1. Handle sonar echoes running the control algorithm and updating the servo.
2. Handle radio packets by running the decoder.

- Had we had access to 2 CPUs we could place one task in each.
- We can imagine some construct that allows us to express this in our program.