# CMPE-443 Real-Time System Design Problem Session 09.11.2023

## Introduction to RTS

* 1. Concepts of system, input, output
  2. Definitions of soft, firm, and hard real-time systems (RTS)
  3. Flow of control, natural flow of control, change of the natural flow of control, events; synchronous and asynchronous events; periodic, aperiodic, and sporadic events
  4. Deterministic and non-deterministic systems
  5. Utilization of RTS, ranges of safe and not-safe load, utilization formula for a system of periodic tasks, execution time, computational complexity, CPU performance, choice of CPU for safe RTS
  6. Plant, controller, control law, sensors, actuators, A/D and D/A converters
  7. Reference and control signals, error, proportional, integral, and derivative controller, right rectangle integration, finite difference derivative, recursive calculation of the control PID controller signal, controller algorithm
  8. Time sampling, Nyquist-Kolmogorov theorem on the sampling frequency

## Hardware issues

* 1. CPU von Neumann architecture
  2. Latching, edge and level signal, tri-state logic, wait states
  3. UART. Parallel and serial data representation, Shift registers, Serial bus, collisions, inductive coupling, SCSI, daisy chain connection, Firewire, asynchronous and isochronous modes
  4. Fetch, decode, execute CPU cycle, instruction formats, 0-, 1-, 2-, and 3-operand instructions formats, program counter, instruction register, microcontrollers, BTS (bit test and set) use for synchronization, addressing modes (immediate data, direct memory location, indirect memory location, register indirect, double indirect)
  5. CISC (complex instruction set computers), RISC (reduced instruction set computers)
  6. Memory access synchronization by DST (data strobe) signals, memory hierarchy (internal CPU memory, CPU registers, cache, main, external), DRAM, SRAM, UVROM, EEPROM, fusible link PROM, Flash, Ferroelectric RAM, Ferrite core, Memory organization
  7. Programmed I/O, DMA, DMA controller, Memory mapped I/O
  8. Interrupts, internal and external interrupts, interruption vector, interruption sequence, enabling/disabling interrupts
  9. Programmable interrupt controller, interrupt priority, interrupt number
  10. Interfacing devices via interrupts
  11. Interruptible instructions
  12. Watchdog timers
  13. Enhancing performance, locality of references, locality in code segment, locality in data segment, cache memory, hit ratio, average access time with cache memory, pipelining, pipe depth, speedup dependence on depth and the number of instruction, superpipelined and superscalar architecture. Conditions of pipelined architecture efficiency, coprocessors (floating point, graphical, i/o, etc.).
  14. ASICs, PAL, PLA, FPGA, transducers, thermometers, accelerometers, position resolvers, gyroscopes, stable rotation masses, ADC, DAC

**Tasks**

1. Should a task be allowed to interrupt itself? (Laplante, 2nd Ed., p. 167, Ex. 14 ; Laplante, 3rd Ed. p. 156, Ex. 3.2)

Why not? If we assume, that tasks of the previous exercise are interruption handlers, we see that while handling one interruption signal there may come again interruption signal of other, as in exercise 1, level of priority and from the other source, or of the same level of priority, and from the same source. This situation corresponds to possible bursts of requests for serving. They may be queued on the level of controller, or by allowing of interrupting itself. We shall have in memory stack the sequence of interrupted calls. Of course, if such bursts will be for a long period, stack may be overflowed and this may cause failure of the system. So, self interrupting may be allowed, but run-time stack size is to be designed taking into account maximal possible depth of interruptions, and size of stack required for each context switch.

Better, is not to allow self-interruptions. In such an approach, size of the stack is better predictable.

1. What criteria are needed to determine the size of the run-time stack in a multiple-interrupt system? What safety precautions are necessary? (Laplante, 2nd Ed., p. 167, Ex. 15; Laplante, 3rd Ed. p. 156, Ex. 3.3)

This is a question, connected with the previous exercise. For designing run-time stack we are to multiply maximal possible depth of interruptions by size of context for switching saved in stack each time of interruption occurs. For safety precautions, this figure is to be increased by size of one or two more interruptions.

1. What are the desirable features that a system should have to provide for predictability in time-critical applications (Laplante, 3rd Ed. p. 156, Ex. 3.5)?

- Interruptions should be supported

- Interruptions should have priorities (higher priority interruption should be served first)

- Instructions should not take much time

- Context switching should be fast

- preemptive multitasking should be supported

1. Calculate processor utilization for the following task set:

|  |  |  |
| --- | --- | --- |
| Task# | E | P |
| 1 | 3 | 7 |
| 2 | 5 | 16 |
| 3 | 3 | 15 |



1. Calculate processor utilization for the following task set:

|  |  |  |  |
| --- | --- | --- | --- |
| Task# | E | P | D |
| 1 | 1 | 5 | 4 |
| 2 | 2 | 8 | 6 |
| 3 | 1 | 4 | 3 |

<=1

1. Calculate processor utilization for the following task set:

|  |  |  |
| --- | --- | --- |
| Task# | E | P |
| 1 | 0.8 | 2 |
| 2 | 1.4 | 4 |
| 3 | 2 | 8 |

**Exercise 2.2.** Implement arithmetic and jump operations in 0-, 1-, 2-, and 3-address machine

The table below is from <http://www.csse.monash.edu.au/~lloyd/tildeProgLang/PL-Architecture/>

|  |  |
| --- | --- |
| **0-address machine (reverse Polish)** | |
| where the processor has a stack and some supporting hardware, at least a top of stack (TOS) pointer. | |
| **operation** | **e.g. or comment** |
| load\_literal <int> | effect: TOS:=TOS+1; stack[TOS]:=<int> load a *constant* onto the top of stack; this can be used in arithmetic or to get an address onto the stack for use by a load or a store instruction later (it is splitting hairs to argue whether the literal is an address or a constant which might happen to be used as an address elsewhere) |
| load | effect: stack[TOS]:=memory[stack[TOS]] take the top-of-stack as an address, replace the top-of-stack with the contents of that address. |
| sto | effect: memory[stack[TOS-1]]:=stack[TOS]; TOS:=TOS-2 store contents of top of stack at the address in stack[TOS-1] then pop the value and the address |
| <opcd> | where <opcd> is add | sub |... effect: stack[TOS-1] := stack[TOS-1] <op> stack[TOS]; TOS:=TOS-1 |
| possible code generated for   x := y + z:    load\_literal @x  load\_literal @y  load  load\_literal @z  load  add  sto    where @x is the address of x; the compiler retrieves the address of a variable from the compiler's lookup-table, e.g., @x might be 36, say.    Some possible extensions to the instruction set include:  load <addr>  (equiv. to load\_literal <addr>; load);  sto <addr>   (similar);  instructions (such as permute, or duplicate) to operate on elements near the top of the stack (which can make up for the lack of reverse-subtract or reverse-divide) | |

|  |  |  |
| --- | --- | --- |
| **1-address machine** | | |
| where the processor has a special *accumulator* which is implicitly used in each arithmetic operation. The accumulator could be the *only* register, or there could also be one or more index-registers in which case there would very probably be accumulator-to-indexReg transfer operations. In the former case an address is limited to being an integer constant; in the latter case indexed addresses would be allowed. | | |
| **operation** | **alternative notation** | **e.g. or comment** |
| load <addr> | acc:=<addr> | load the word at the given address into the acc |
| load\_indirect1 <addr> | acc := [<addr>] | use the word at the address as a pointer to the word to be loaded into acc; something like this is necessary if there are no index registers, or... |
| load\_indirect2 | acc := [acc] | ... use the contents of the acc as a pointer to the word to be loaded into the acc; something like this is necessary if there are no index registers |
| sto <addr> | <addr>:=acc | store contents of accumulator at <addr> |
| store indirect, similar considerations to load indirect | | |
| <opcd> <addr> | acc <op>= <addr> | effect: acc := acc <op> memory[<addr>] |
| possible code generated for   x := y + z:    load @y *--acc := y*  add @z *--acc += z*  sto @x *--x := acc*    the compiler retrieves the address of a variable from the compiler's lookup-table. | | |

|  |  |  |
| --- | --- | --- |
| **2-address machine (?1.5-address?)** | | |
| **where**  <reg> ::= R0 ... R15, say.  general-purpose index- | integer- registers.  <addr> ::= <offset> | <offset>[<reg>]  if present the register is an index and it and the offset are added to give the address.    In some machines only the early registers, e.g., R0-R3, can be used for indexing.    In some 2-address machines indexing with R0 is used to denote do not index, in which case <offset>[R0] is equivalent to <offset>. | | |
| **operation** | **alternative notation** | **e.g. or comment** |
| load <reg>, <reg> | <reg>:=<reg> | load R7, R3, or R7:=R3 transfer from one register to another |
| <opcd> <reg>,<reg> | <reg> <op>= <reg> | add R7, R3, or R7 += R3, arithmetic operation on two registers |
| load <reg>, <addr> | <reg>:=<addr> | load R7,36[R2], or R7 := 36[R2]. load the register with the contents of the word at the given address |
| sto <reg>, <addr> | <addr>:=<reg> | sto R7,36[R2], or 36[R2] := R2, store the contents of the register in the word at the given address |
| <opcd> <reg>, <addr> | <reg><op>=<addr> | add R3, 36[R2], or R3+=36[R2] |
| jmp <reg> | jmp <reg> | unconditional jump to the address held in the register |
| jmp <addr> | jmp <addr> | unconditional jump to the address |
| jmp <cond> <addr> | jmp <cond> <addr> | jump to the address *if* the condition has been set, e.g., by a compare instruction |
| BAL <reg>, <addr> | BAL <reg>, <addr> | Branch and link: save the program counter in the register (e.g., for procedure return) and then jump to the address |
| possible code generated for   x := y + z:    load R5, @y *--R5 := y*  add R5, @z *--R5 += z*  sto R5, @x *--x := R5*    the compiler retrieves the address of a variable from the compiler's lookup-table. Note that local, non-local and global variables in a [block-structured](http://www.csse.monash.edu.au/~lloyd/tildeProgLang/PL-Block/) language require different, perhaps more complex, code sequences to access them.  Each instruction above applies to a register and a "full" memory address. Arguably a register is a restricted kind of address, so you might call them 1.5-address instructions. There may also be some 2-full-address instructions. These are in fact necessary for any operation on arbitrarily long operands such as strings or decimal numbers; there is also the matter of specifying an operand's length -- either in the instruction, or in the operand, or in a register. | | |
| <opcd> <addr>,<addr> | <addr> <op>= <addr> | add 10[R3], 20[R6], or 10[R3] += 20[R6] |

|  |  |  |
| --- | --- | --- |
| **3-address machine** | | |
| where three-address instructions act on values in memory, not in registers. | | |
| **operation** | **alternative notation** | **e.g. or comment** |
| <opcd> <addr> <addr> <addr> | <addr> := <addr> <op> <addr> | add 10[R1],20[R2],30[R3], or 10[R1]:=20[R2]+30[R3] Combine the second and third operands using the operator (e.g., +) and store the result in the location indicated by the first address. |
| possible code generated for   x := y + z:    add @x, @y, @z | | |

|  |  |  |
| --- | --- | --- |
| **Register-file machine (3 register addresses)** | | |
| has 2-address (1.5-address) instructions for loading from, and storing in, memory but the other instructions each specify three *registers*. | | |
| **operation** | **alternative notation** | **e.g. or comment** |
| load <reg> <addr> | <reg> := <addr> |  |
| sto <reg> <addr> | <addr> := reg> |  |
| <opcd> <reg><reg><reg> | <reg> := <reg><op><reg> | add R2 R3 R4, or R2:=R3+R4 etc. |
| possible code generated for   x := y + z:   |  |  | | --- | --- | | load R1, @y | *--R1 := y* | | load R2, @z | *--R2 := z* | | add R6, R1, R2 | *--R6 := R1 + R2* | | sto R6, @x | *--x := R3* | | | |

The zero-address machine needs more instructions for *x := y + z* than the 3-address machine, say, but they are *shorter* instructions so the 0-address machine does not necessarily have a performance disadvantage, and the 3-address machine does not necessarily have a performance advantage, because of this.

Jump instructions can be implemented as follows <https://en.wikipedia.org/wiki/Status_register>

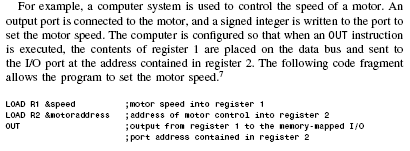
|  |  |
| --- | --- |
| TEST | if stack(top −1) = stack(top) then CCR ← 0 (Condition Code Register)  if stack(top −1) < stack(top) then CCR ← −1  if stack(top −1) > stack(top) then CCR ← 1 |
| JUA m | PC ← m |
| JNE m | if CCR ≠ 0 then PC ← m |
| JEQ m | if CCR = 0 then PC ← m |
| JLT m | if CCR = −1 then PC ← m |
| JLE m | if CCR < 1 then PC ← m |
| JGT m | if CCR = 1 then PC ← m |
| JGE m | if CCR > −1 then PC ← m |
| BRANCH m | Top++; stack(top) ← PC  PC ← m |
| RETURN | PC ← stack(top); top-- |
| 〈interrupt〉 | Top++; stack(top) ← PC  PC ← (interrupt vector) |

**2.3.** Why is DMA controller access to memory in most systems given higher priority than CPU access to main memory?

While an external device gets access to RAM, CPU can run the process not depending on the being executed I/O operation. Also, having data supplied, CPU has greater flexibility in choosing jobs.

**2.4.** Discuss the relative advantages/disadvantages of DMA, programmed I/O, and memory mapped data transfer as they pertain to real-time systems.

In programmed I/O both, CPU and external device, are involved, and, thus cost time that could impact real-time performance.



In DMA, CPU participation is not required, data transfer is fast. The CPU is prevented from performing a data transfer during DMA through the use of a signal called a bus grant.

In memory mapped I/O, certain memory locations appear as virtual I/O ports. For example, to control the speed of a stepping motor, the required assembly language code might look like the following:

LOAD R1, &speed ;motor speed into R1

STORE R1, &motoraddress ;store to address of motor control

**2.5.** Describe relationship between the main processor and coprocessor in a system with which you are familiar or one that you discover through Web research.

## Intel coprocessors (<http://en.wikipedia.org/wiki/Coprocessor> )

The original [IBM PC](http://en.wikipedia.org/wiki/IBM_PC) included a socket for the [Intel 8087](http://en.wikipedia.org/wiki/Intel_8087) floating point coprocessor (aka [FPU](http://en.wikipedia.org/wiki/Floating_point_unit)) which was a popular option for people using the PC for CAD or mathematics-intensive calculations. In that architecture, the coprocessor sped up floating-point arithmetic on the order of fiftyfold. Users that only used the PC for word processing, for example, saved the high cost of the coprocessor, which would not have accelerated performance of text manipulation operations.

The 8087 was tightly integrated with the 8088 and responded to floating-point [machine code](http://en.wikipedia.org/wiki/Machine_code) operation codes inserted in the 8088 instruction stream. An 8088 processor without an 8087 would interpret these instructions as an internal interrupt, which could be directed to trap an error or to trigger [emulation](http://en.wikipedia.org/wiki/Emulation) of the 8087 instructions in software.

[](http://en.wikipedia.org/wiki/Image:80386with387.JPG)

Intel 80386 CPU w/ 80387 Math Coprocessor

Another coprocessor for the 8086/8088 central processor was the 8089 input/output coprocessor. It used the same programming technique as 8087 for input/output operations, such as transfer of data from memory to a peripheral device, and so reducing the load on the CPU. But IBM didn't use it in IBM PC design and Intel stopped development of this type of coprocessor.

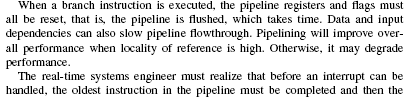
During the era of 8- and 16-bit desktop computers another common source of floating-point coprocessors was [Weitek](http://en.wikipedia.org/wiki/Weitek). The [Intel 80386](http://en.wikipedia.org/wiki/Intel_80386) [microprocessor](http://en.wikipedia.org/wiki/Microprocessor) used an optional "math" coprocessor (the 80387) to perform floating point operations directly in [hardware](http://en.wikipedia.org/wiki/Computer_hardware).

The Intel 80486DX processor included floating-point hardware on the chip. Intel released a cost-reduced processor, the 80486SX, that had no FP hardware, and also sold an 80487SX co-processor that essentially disabled the main processor when installed, since the 80487SX was a complete 80486DX with a different set of pin connections. While consumers may have resented paying for a processor that was essentially wasted when the upgrade was installed, this marketing strategy did allow increased volume of sales of 80486 family processors, thereby accelerating the eventual price reductions.

Intel processors later than the 80486 integrated floating-point hardware on the main processor chip; the advances in integration eliminated the cost advantage of selling the floating point processor as an optional element. It would be very difficult to adapt circuit-board techniques adequate at 75 MHz processor speed to meet the time-delay, power consumption, and radio-frequency interference standards required at gigahertz-range clock speeds. These on-chip floating point processors are still referred to as coprocessors because they operate in parallel with the main CPU.

**2.6.** What special problems do pipelined architectures pose for real-time system designers? Are they any different for non-real-time systems?

Pipelining is a form of speculative execution in that the instructions that are pre-fetched are taken to be the next sequential instructions. If any of the instructions in the pipeline are a brunch instruction, the pre-fetched instructions further in the pipeline are no longer valid.





The problems are the same but for RTS are more sensitive.

Having conveyor stages and tasks, speedup can be estimated as follows:

,

where is the time of execution of a system of tasks serially (with only one device in the pipeline and each task needs stages of processing), and is the time of execution of the set of tasks but now using a pipeline with stages. Assuming each task completes needs one time unit to complete on each stage on the conveyor, we get



**2.7.** Compare and contrast the different memory technologies discussed in this chapter as they pertain to real-time systems

Ferrite-core technique may be used in space-born and military RTS as they cannot be discharged by electrostatic discharge or by a charged particle in space. It is slow (10 microsecond access time), bulky and consumes lots of power.

Fusible link ROMs are a type of nonvolatile memory, access time about 50 ns. They are used to keep static programs and data.

PLA can be used as fusible link ROMs.

EEPROM can be used to keep data as for black-boxes.

# Programmable Logic Array (PLA) (from <http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/Comb/pla.html> )

One way to design a combinational logic circuit is to get gates and connect them with wires. One disadvantage with this way of designing circuits is its lack of portability.

You can now get chips called PLA (programmable logic arrays) and "program" them to implement Boolean functions.

Fortunately, a PLA is quite simple to learn, and produces nice neat circuits too.

Let's try to implement a truth table with a PLA.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x2** | **x1** | **x0** | **z1** | **z0** |
| **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **1** | **1** | **0** |
| **0** | **1** | **0** | **0** | **0** |
| **0** | **1** | **1** | **1** | **0** |
| **1** | **0** | **0** | **1** | **1** |
| **1** | **0** | **1** | **0** | **0** |
| **1** | **1** | **0** | **0** | **0** |
| **1** | **1** | **1** | **0** | **1** |

Each of the vertical lines with an AND gate corresponds to a minterm. For example, the first AND gate (on the left) is the minterm: **\x2\x1x0**.

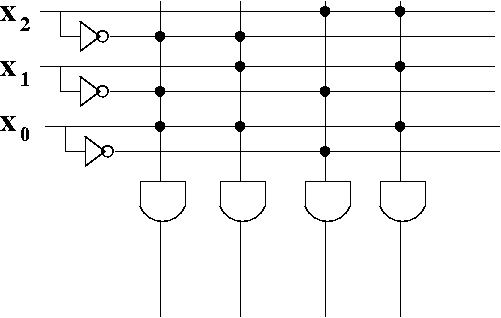
The second AND gate (from the left) is the minterm: **\x2x1x0**.

The third AND gate (from the left) is the minterm: **x2\x1\x0**.

The fourth AND gate is the minterm: **x2x1x0**.

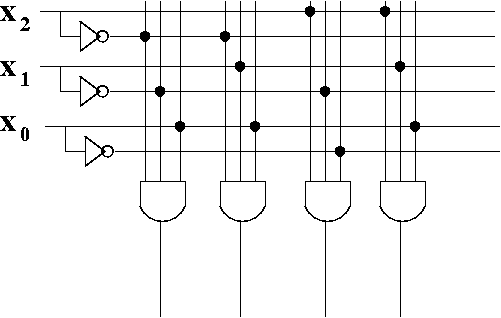
The first three minterms are used to implement **z1**. The third and fourth minterm are used to implement **z0**.

This is how the PLA looks after we have all four minterms.



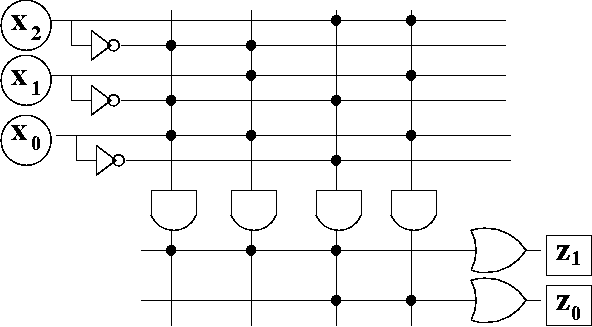
Now you might complain. How is it possible to have a one input AND gate? How can three inputs be hooked to the same wire to an AND gate? Isn't that invalid for combinational logic circuits?

That's true, it is invalid. However, the diagram is merely a simplification. I've drawn the each of AND gate with three input wires, which is what it is in reality (there is as many input wires as variables). For each connection (shown with a black dot), there's really a separate wire. We draw one wire just to make it look neat.



The vertical wires are called the AND plane. We often leave out the AND gates to make it even easier to draw.

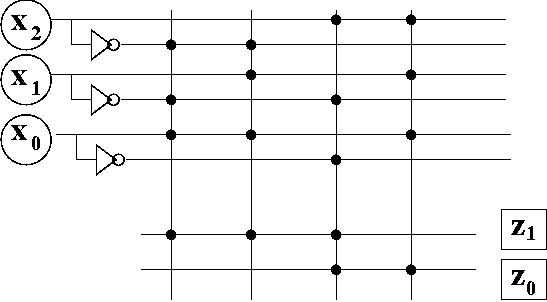
We then add OR gates using horizontal wires, to connect the minterms together.



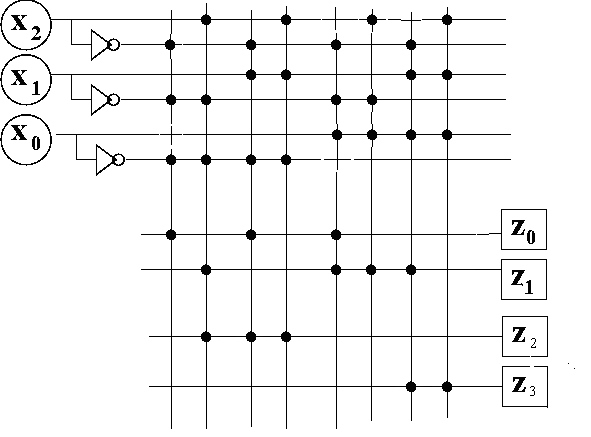
Again, a single wire into the OR gate is really 4 wires. We use the same simplification to make it easier to read.

The horizontal wires make up the OR plane.

This is how the PLA looks when we leave out the AND gates and the OR gates. It's not that the AND gates and OR gates aren't there---they are, but they've been left out to make the PLA even easier to draw.



PLA 8 memory cells with 4 bits each are represented below



For example, to read memory cell 011, we feed x0=0, x1=1, x2=1, getting as output 0010 (z0=0, z1=0, z2=1, z3=0).

**2.8.** Should the instruction following the TEST instruction be interruptible? If so, what must the implicit BRANCH instruction (interrupt) do?

Flag register must be saved together with PC register.

**2.9.** It is common for programmers to create continuous test and loop code in order to poll I/O devices or wait for interrupts to occur. Some processors provide an instruction (WAIT or HALT) that allows the processor to hibernate until an interrupt occurs. Why is the second form more efficient and desirable?

For example, battery capacity can be saved in the second case.

**2.12.** What is the difference between coprocessing and multiprocessing? What are the advantages and disadvantages of each?

In coprocessing, processors work serially. In multiprocessing, they work concurrently. The second way is more complicated and provides greater performance.

1. Interruptions: interruption signal, interruption number, interruption vector, table of interruption vectors, interruption sequence, use of stack pointer SS:SP to keep return information (PSW, CS:IP), context, saving/restoring context, return from interruption. Chains of interruption handlers.