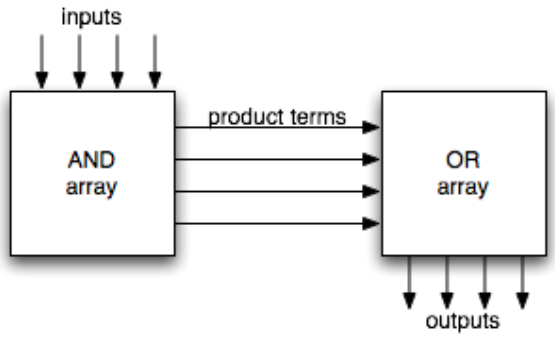
Quick Review

Convert the truth table in the exercise below into a reduced sum of products Boolean expression.



Programmable Logic Arrays

It is cumbersome to build a custom circuit out of individual gates; so programmable logic components can be configured to implement arbitrary logic, and yet can still be manufactured as standard components. PLAs directly implement sum of products expressions, making them very good for control implementations. Field Programmable Gate Arrays (FPGAs) are more advanced versions that can be reprogrammed and are in general cooler (internal registers, etc).

PLA Exercise

Draw in wires in the diagram that implement the following truth table:

ABC	F
000	0101
001	1000
010	0000
011	0010
100	1111
101	1001
110	1111
111	1111

$F_0 = \neg BC + A$
$F_1 = BC + AB$
$F_2 = A\neg B\neg C + \neg ABC + AB$
$F_3 = A + \neg B\neg C$

It wasn't possible to reduce F2 to two-input ANDs and ORs, as the diagram suggested.

Pipelining

Any process that goes through a series of distinct steps can be made more efficient through pipelining. The basic idea is to handle multiple tasks in parallel (at the same time) in order to make full use of your resources (washing machines, ALUs, etc).

Latency – the time it takes to process a *single* task completely (measured in seconds)
Throughput – the total amount of tasks completed in a period of time (measured in tasks per second)
Pipelining improves total *throughput*, not the *latency* of an individual instruction. This means that a single instruction will take the same amount of time or even longer, but over a longer period of time a pipelined processor can get more done.

MIPS Pipelining

MIPS, keeping with its simplicity, is typically implemented with a 5-stage pipeline (as long as the ISA is maintained it could implemented in any way)
The five MIPS pipeline stages:

* Instruction Fetch (IF) - Computes the next PC, and requests the next instruction from Memory
* Instruction Decode (ID) - Reads the registers, and starts to set the control signals based on the instruction
* Execute (EX) - Does the actual computation specified by the operation (includes computing the memory address for memory instructions)
* Memory (MEM) - Performs the needed operation from memory - reads for load instructions and writes for store instructions
* Write Back (WB) - Writes back the results of the operation to the Register File

Hazards

Structural Hazards – Hazards that occur due to competition for the same resource (register file read vs. write back, instruction fetch vs. data read). These are solved by caching and clever register timing.
Control Hazards – Hazards that occur due to non-sequential instructions (jumps and branches). These cannot be solved completely by forwarding, so we're forced to introduce a branch-delay slot.
Data Hazards – Hazards that occur due to data dependencies (instruction 2 requires the result of instruction 1). These are mostly solved by forwarding, but `lw` still requires a bubble.

Pipelining Exercises

Suppose you've designed a MIPS processor implementation in which the stages take the following lengths of time: IF=20ns, ID=10ns, EX=20ns, MEM=35ns, WB=10ns. What is the minimum clock period for which your processor functions properly? Where should the bulk of your R&D budget go for the next generation of processors?
Memory is the bottleneck, limiting to the period to 35ns, so it needs more R&D.
Your friend tells you that his processor design is 10x better than yours, since it has 50 pipeline stages to your 5. Is he right? (This is intentionally vague)
No for a variety of reasons, including more complexity and more costly flushes
Rewrite the following program to minimize hazards (assume the cleverest HW possible):

```
Find:   lbu $t0, 0($a0)
        addi $a0, $a0, 1
        bne $t0, $a1, Find
        nop
        add $v0, $0, $0
Loop:   addi $v0, $v0, 1
        addi $t1, $a0, 1
        lbu $t0, 0($t1)
        sb $t0, 0($a0)
        addi $a0, $a0, 1
        bne $t0, $0, Loop
```

```
Find:   lbu $t0, 0($a0)
        addi $a0, $a0, 1
        bne $t0, $a1, Find
        add $v0, $0, $0 #A
Loop:   addi $t1, $a0, 1
        lbu $t0, 0($t1)
        addi $v0, $v0, 1 #B
        sb $t0, 0($a0)
        bne $t0, $0, Loop
        addi $a0, $a0, 1 #C
        jr $ra
```

```
        nop
        jr $ra
```

A: branch delay slot – it doesn't hurt to execute the instruction in each loop iteration. One instruction saved
B: load delay slot – One inst saved per loop.
C: branch delay slot – One inst saved per loop.