Tahmini Ders İçeriği
(Tentative Couse Schedule – Syllabus)

1. Hafta: Sayı sistemleri, onluk/ikilik taban sayı gösterimleri, mantıksal kapılar, computer system overview, başarım (performance)

2. Hafta: 2'lik tabanda işaretli sayılar, mikroişlemci tarihi, benchmarking, başarım,

3. Hafta: Başarım, Amdahl yasası, RISC-V development Environment, Verilog HDL ile Birleşik (Combinational) devreler

4. Hafta:, Verilog HDL ile sıralı (sequential) mantıksal devre ve sonlu durum makinası tasarımı, timing analysis

5. Hafta: Aritmetik devre tasarımları: Toplama, çıkarma, çarpma, bölme, trigonometri, square-root, hyperbolic, exponential, logarithm

6. Hafta: Fixed ve Floating-Point sayı gösterimleri

7. Hafta: RISC-V buyruk kümesi mimarisi (ISA) ve buyrukların tanıtımı

8. Hafta: RISC-V buyruk kümesi mimarisi (ISA) ve buyrukların tanıtımı

9. Hafta: Tek-çevrim işlemci tasarımı (single-cycle CPU)

10. Hafta: Çok-çevrim işlemci tasarımı (multi-cycle CPU)

11. Hafta: Boruhatlı işlemci tasarımı (pipelined CPU)

12. Hafta: Bellek sistemi ve hiyerarşisi

13. Hafta: İleri mimari konuları: Branch prediction, superscalar cpu, out-of-order execution, multi-core systems

14. Hafta: Gömülü sistemler, mikrodenetleyiciler, SoCs

TEKNOFEST 2023'ün ARDINDAN

TEKNOFEST 2023’ün ARDINDAN

Sayısal İşlemci:

Sayısal Görüntü İşleme:

Analog:


Fatih Uğurdağ
Özyeğin Üniversitesi
Müh. Fakültesi Dekanı
Ali Baran
Fatih Say
BARAN
kasırga


TEKNOFEST
2023
ÇIP TASARIM
YARIŞMASI
ÇİP TASARIM
YARIŞMASI
BİLGEN
YONGATEK

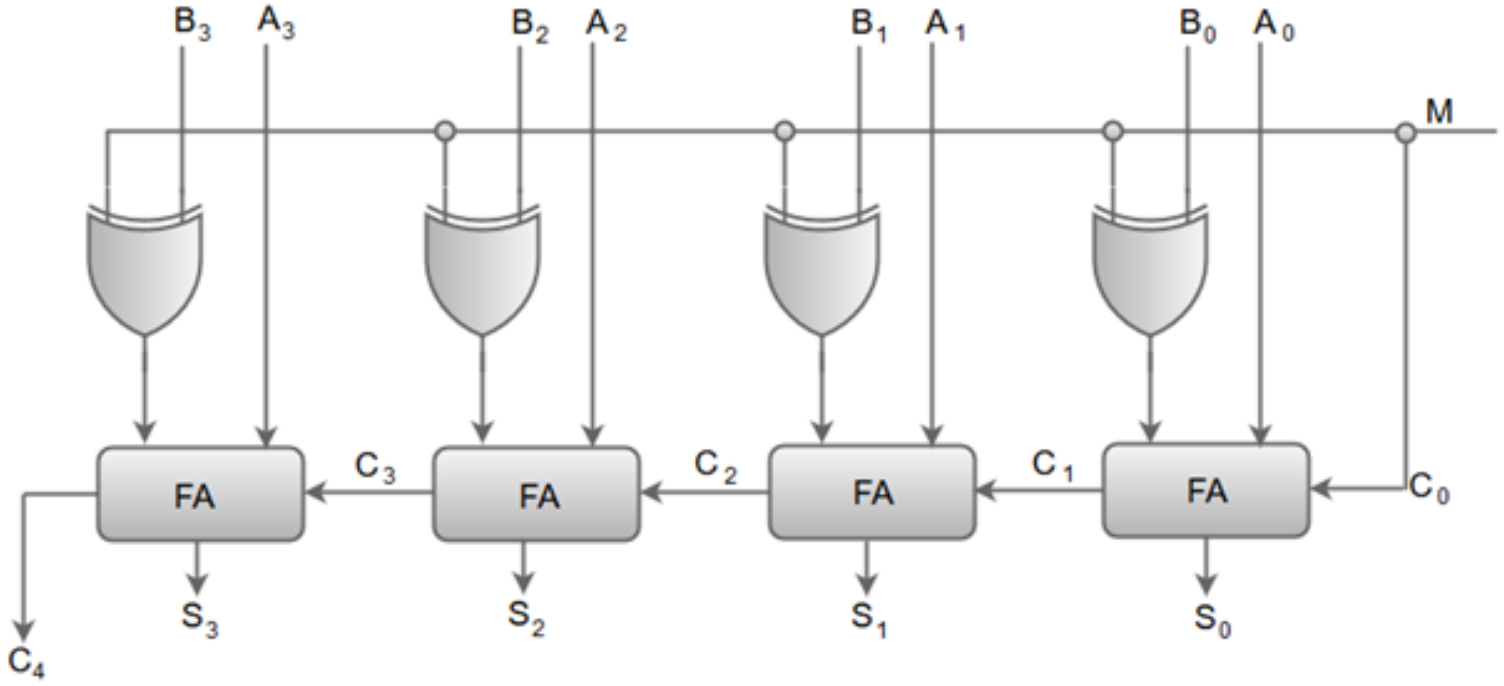

ÇİP TASARIM YARIŞMASI
CHIP DESIGN COMPETITION
#MİLLİ TEKNOLOJİ HAMLESİ
İSTANBUL
6
1
SAKARYA
KARABÜK
5
FİNALİST TAKIM SAYISI


TEKNOFEST
2023
ÇIP TASARIM

Instruction Set Architecture (ISA)
(Buyruk Kümesi Mimarisi)

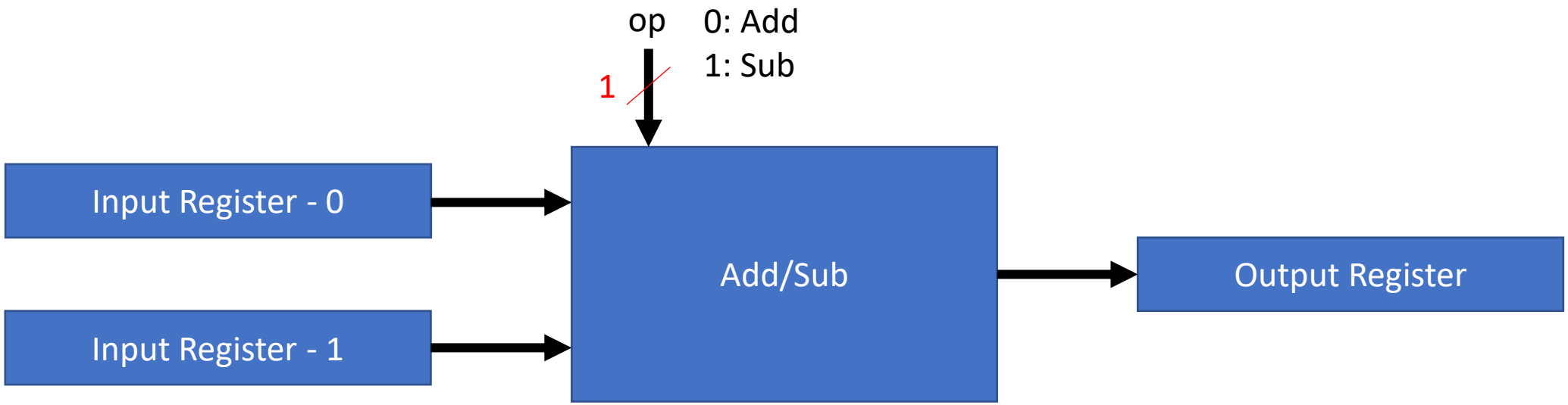
Sabit bir donanım üzerinde kontrol sinyalleri yardımıyla farklı işlemler yapılabilir. Örneğin adder/subtractor devresi:

4 bit adder-subtractor:

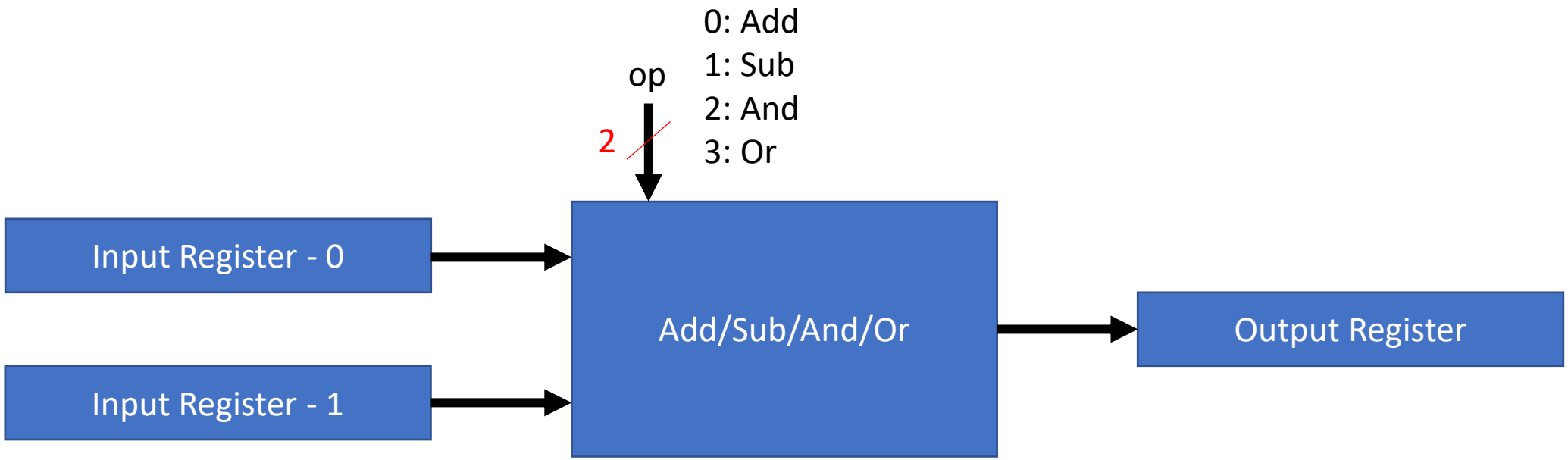



```
module add_sub
#(parameter N = 32)
(
input [N-1:0] a_i,
input [N-1:0] b_i,
input op_i,              // add if 0, sub if 1
output [N-1:0] s_o,
output cout_o
);

wire [N:0] carry;
wire [N-1:0] b_reg;

genvar i;
generate
for (i=0; i<N; i=i+1) begin
    assign b_reg[i] = b_i[i] ^ op_i;
    full_adder full_adder_inst
    (
        .a_i(a_i[i]),
        .b_i(b_reg[i]),
        .cin_i(carry[i]),
        .s_o(s_o[i]),
        .cout_o(carry[i+1])
    );
end
endgenerate

assign carry[0] = op_i;
assign cout_o   = carry[N];

endmodule
```

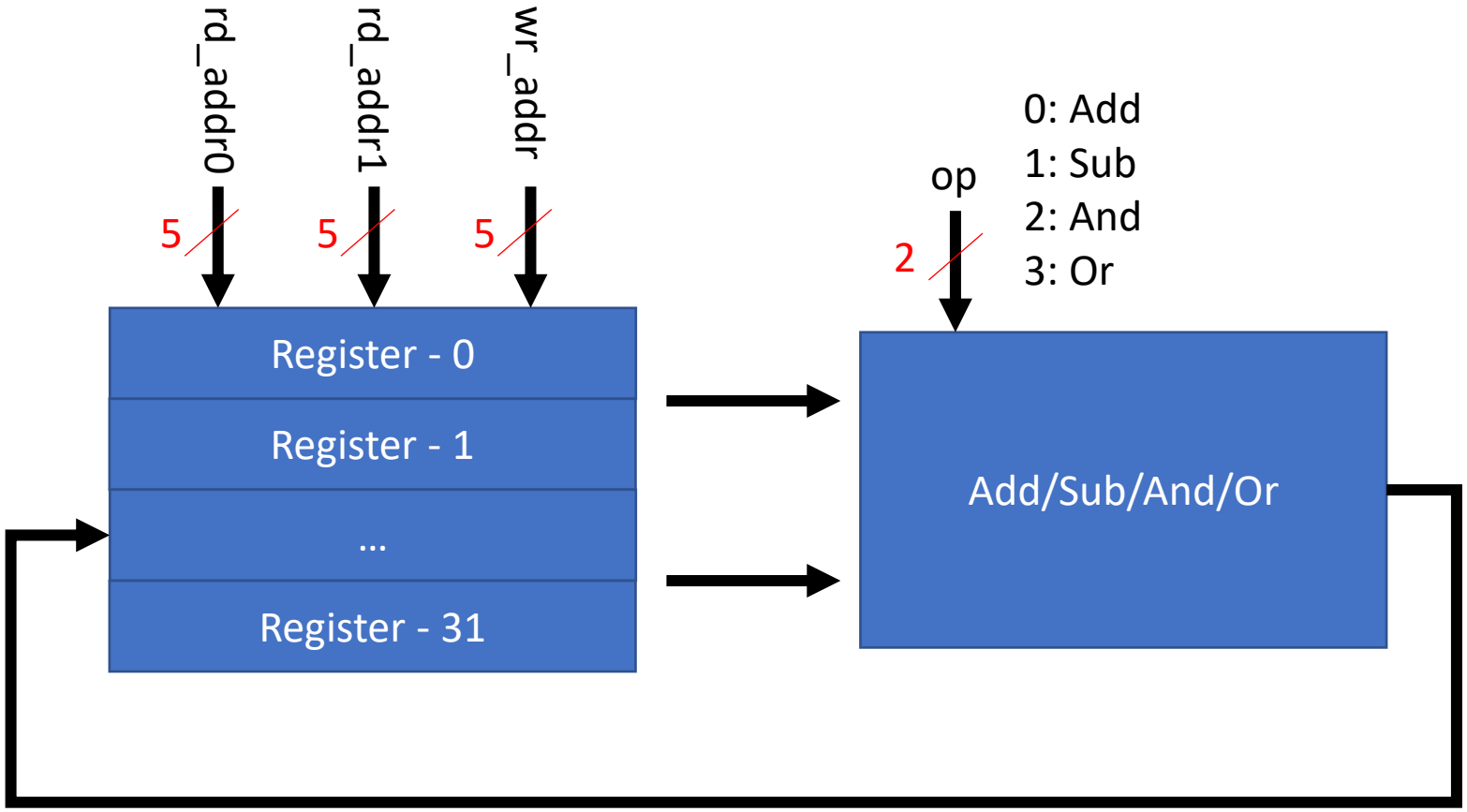
Instruction Set Architecture (ISA)



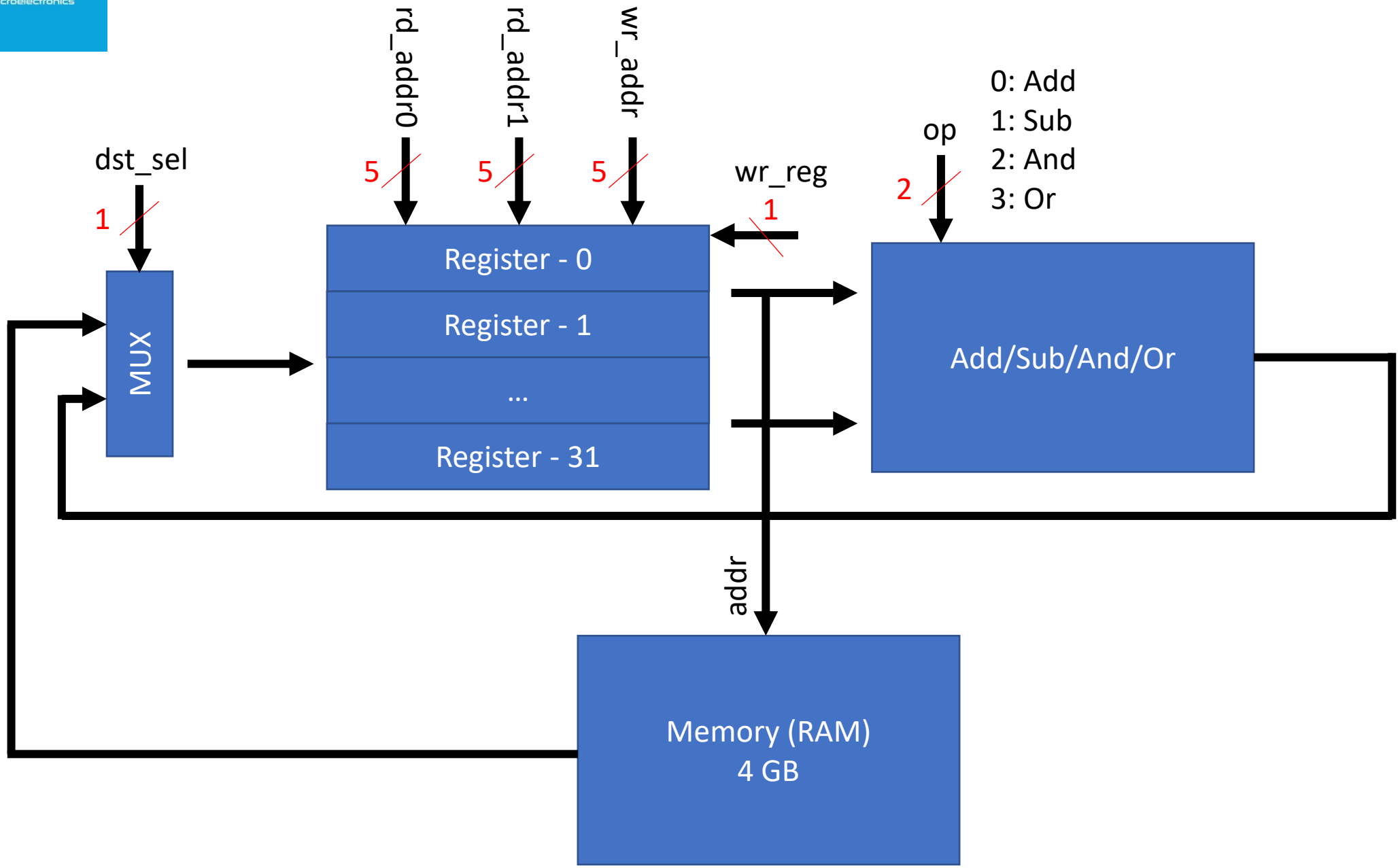
Instruction Set Architecture (ISA)



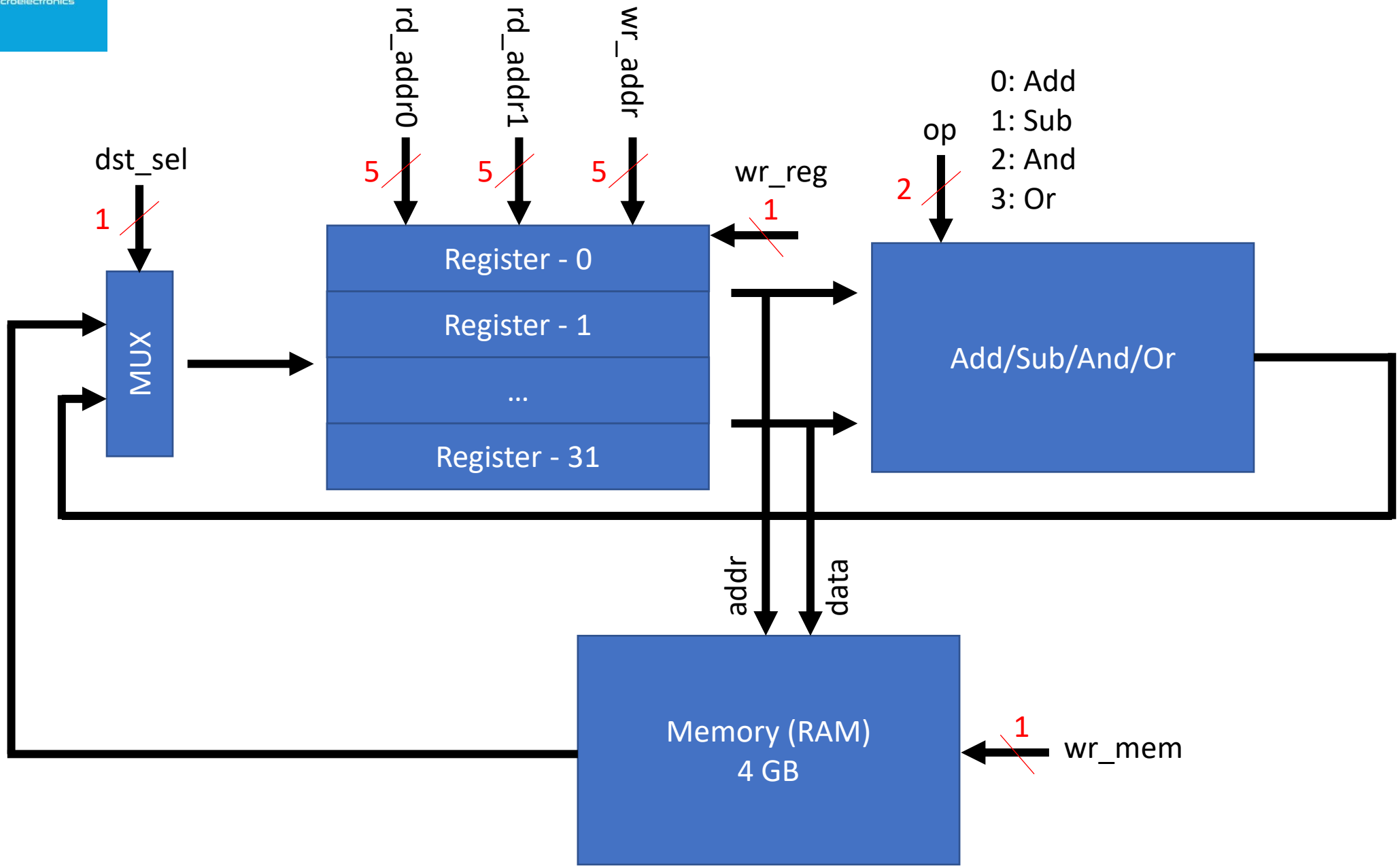
Instruction Set Architecture (ISA)



Instruction Set Architecture (ISA)



Instruction Set Architecture (ISA)



Instruction Set Architecture (ISA)

Donanım devrelerinin kontrol edilmesi gerekiyor.

Örneğin belleğe bir veri yazılacaksa hangi register içeriği yazılacak (rd_addr0, rd_addr1, wr_ram, wr_reg)

Bellekten veri okunacaksa (rd_addr0, dst_sel, wr_ram, wr_reg)

İki register okunup üzerlerinde add işlemi yapılıp başka registera yazılacaksa (rd_addr0, rd_addr1, wr_addr, wr_ram, wr_reg, dst_sel)

Peki yazılım geliştirici bu kontrol sinyallerini nasıl ayarlayacak?

Donanımın bütün detaylarına hakim olması gerekmez mi?

Assembly Language

Kontrol sinyalleri sıfır ve birlerden oluşur. Donanımı (makineyi) kontrol edebilmek için sıfır ve birlerden oluşan bir veri dizisi oluşturulup kontrol sinyallerine yazılmalıdır.

Sıfır ve birlerden oluşan bir veri dizisi ile programlama yapmak yerine, insanın anlayabileceği bir dilde programlama yapmak daha kolaydır. Bu yüzden assembly dili geliştirilmiştir.

Örneğin RISC-V ISA destekleyen bir işlemcide 2 adet register toplayıp cevabı yazmak için:

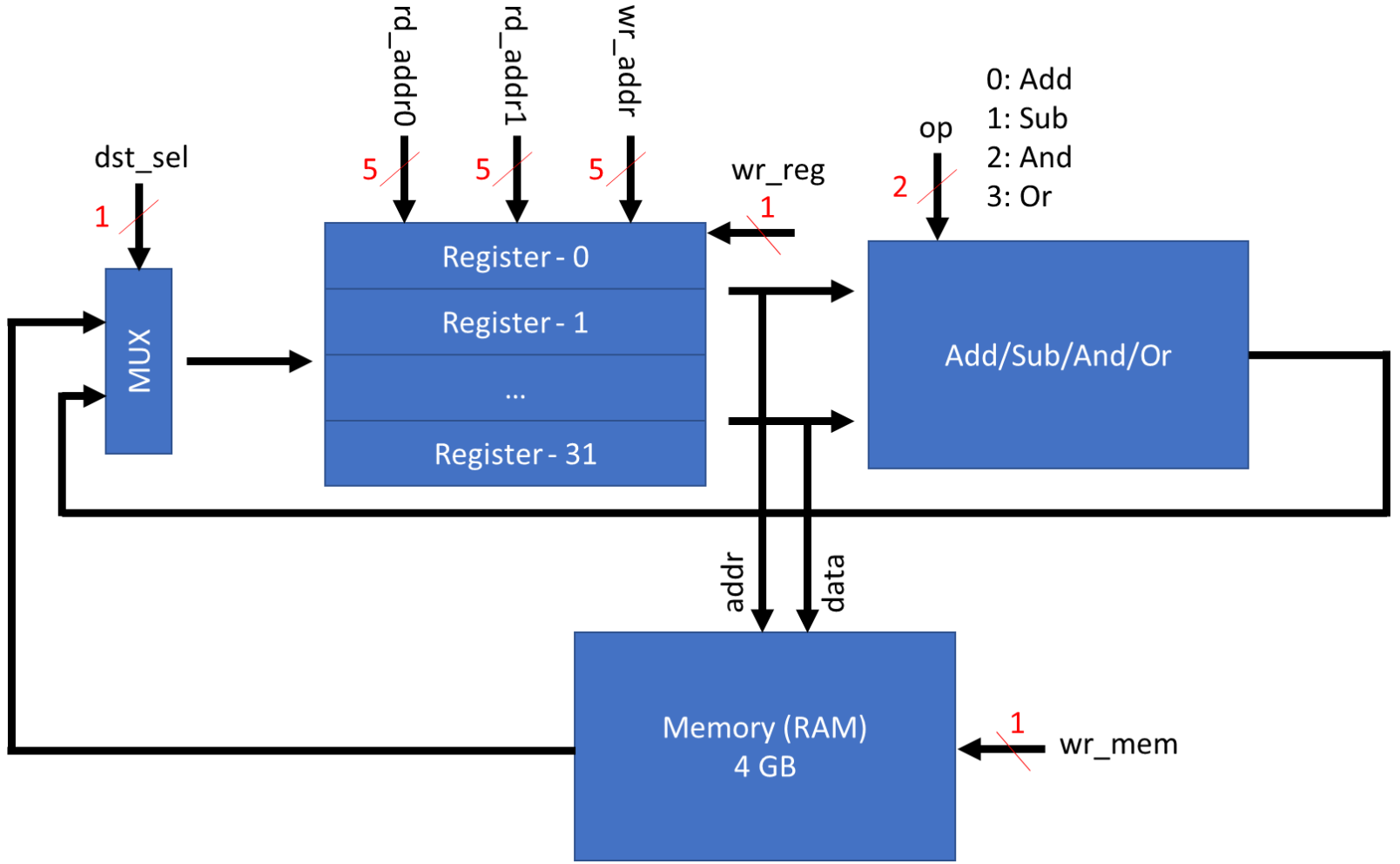
add x9, x20, x21

Burada hedef yazmacı (destination register) 9 numaralı (adresi 9 olan) yazmaçtır
Kaynak yazmaçları 20 ve 21 numaralı (adresleri 20 ve 21 olan) yazmaçlardır
Yani yukarıdaki assembly instruction, 20 ve 21 adreslerindeki yazmaç değerlerini toplayıp 9 adresindeki yazmaca yazar

Assembly Language

add x9, x20, x21

dst_sel	rd_addr0	rd_addr1	wr_addr	wr_reg	Op	…
1	20	21	9	1	0	…



Instruction Set Architecture (ISA)

Donanım yazılımdan, yazılım donanımdan bağımsız düşünülemez ve tasarlanamaz.

Donanım ve yazılım arasında haberleşmeyi sağlayacak, iki tarafın da anlayabileceği bir yapıya ihtiyaç vardır

Bu yapının adı instruction set architecture (ISA) olarak tanımlanmıştır.

ISA ile donanımın anlamlandıracağı instructionlar ve registerlar tanımlanır

Yüksek seviyeli dillerde yazılan programlar derleyici (compiler) ile derlenebilmesi için derleyicinin ISA bilgisine ihtiyacı vardır. Örneğin kaç adet register var ve register genişliği kaç bit? İki sayının toplanması için hangi instructionlar mevcut?

Tasarlanacak olan işlemci donanımı da ISA baz alınarak tasarlanır. Bu sayede işlemcinin hangi programları çalıştırabileceği bilinir.

Instruction Set Architecture (ISA)

Farklı ISA örnekleri:

Intel x86: Register-Memory architecture → İşlemler hem register hem de bellekten operand alabiliyor

MIPS, RISC-V, SPARC (Sun), ARM: Register-Register architecture → İşlemler sadece registerlar üzerinden yapılır, bellekten erişim için load-store gerekir.

İlk mikroişlemci mimarilerinde az instruction ile çok iş yapmaya yönelik mimariler ön plana çıkmıştı (Örn. Intel x86, VAX, Motorola 6800, Zilog Z80). O yıllarda bellek kapasitesinin azlığından ve önbellek gibi birimlerin eksikliğinden dolayı instruction sayısı ve bellek erişimini az tutarak performansı arttırma göz önünde tutuldu: CISC (Complex Instruction Set Computer)

1980'li yıllardan itibaren bellek kapasitesinin artmasıyla ve önbelleklerin gelişmesiyle instructionlarda sadeliğe gidildi, operand olarak sadece registerlar kullanıldı. Bu sayede instruction decoding işlemi basitleşirken pipeline imkanı arttı: RISC (Reduced Instruction Set Computer).

Instruction Set Architecture (ISA)

1980’lerden itibaren RISC popülerleşti ve günümüzde en meşhur RISC ISA ARM firmasının yine ARM adındaki ISA’ıdır (ARMv7, ARMv8).
Yine RISC-V günümüzde popülerleşmektedir.

RISC filozofisinde bazı belli başlı mimari tercihleri mevcuttur:

- Instruction’ların hepsi eşit uzunlukta olsun
- Az sayıda register yeterli, örneğin 32 adet
- Az sayıda instruction type olsun ve field’lar aynı bit alanlarında olsun
- ALU işlemleri registerlar arasında olsun, direk bellekten olmasın

Bu sayede instruction decoding kolaylaşmıştır, pipelining imkanı artmıştır, register file küçük olacağından registerlara erişim hızlıdır.

x86 ISA – Backward Compatibility

Intel was betting its future on a high-end microprocessor, but that was still years away. To counter Zilog, Intel developed a stop-gap processor and called it the 8086. It was intended to be short-lived and not have any successors, but that's not how things turned out. The high-end processor ended up being late to market, and when it did come out, it was too slow. So the 8086 architecture lived on—it evolved into a 32-bit processor and eventually into a 64-bit one. The names kept changing (80186, 80286, i386, i486, Pentium), but the underlying instruction set remained intact.

—Stephen P. Morse, architect of the 8086 [Morse 2017]

```
The AL register is the default source and destination.
If the low 4-bits of AL register are > 9,
   or the auxiliary carry flag AF = 1,
 Then
    Add 6 to low 4-bits of AL and discard overflow
    Increment the high byte of AL
    Carry flag CF = 1
    Auxiliary carry flag AF = 1
 Else
    CF = AF = 0
Upper 4-bits of AL = 0
```

Description of the x86-32 ASCII Adjust after Addition (aaa) instruction. It performs computer arithmetic in Binary Coded Decimal (BCD), which has fallen into the dustbin of information technology history

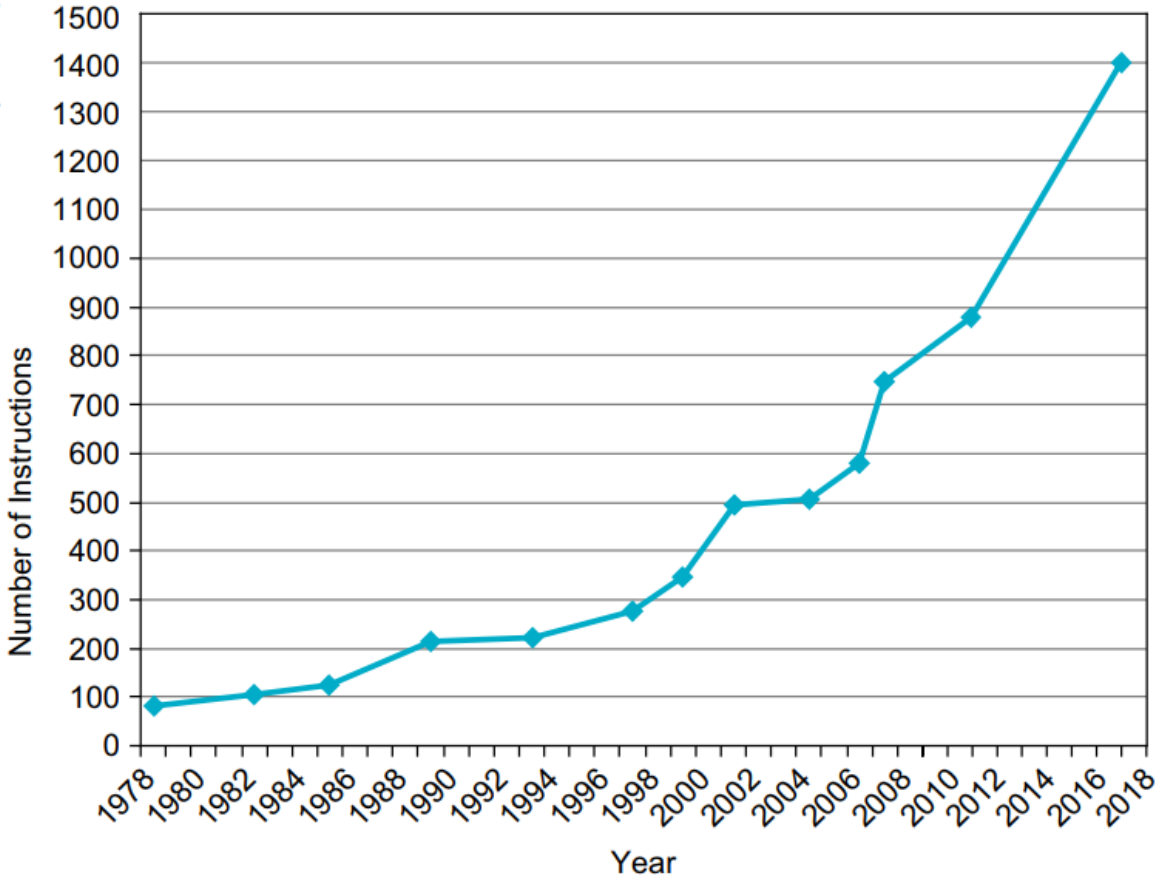
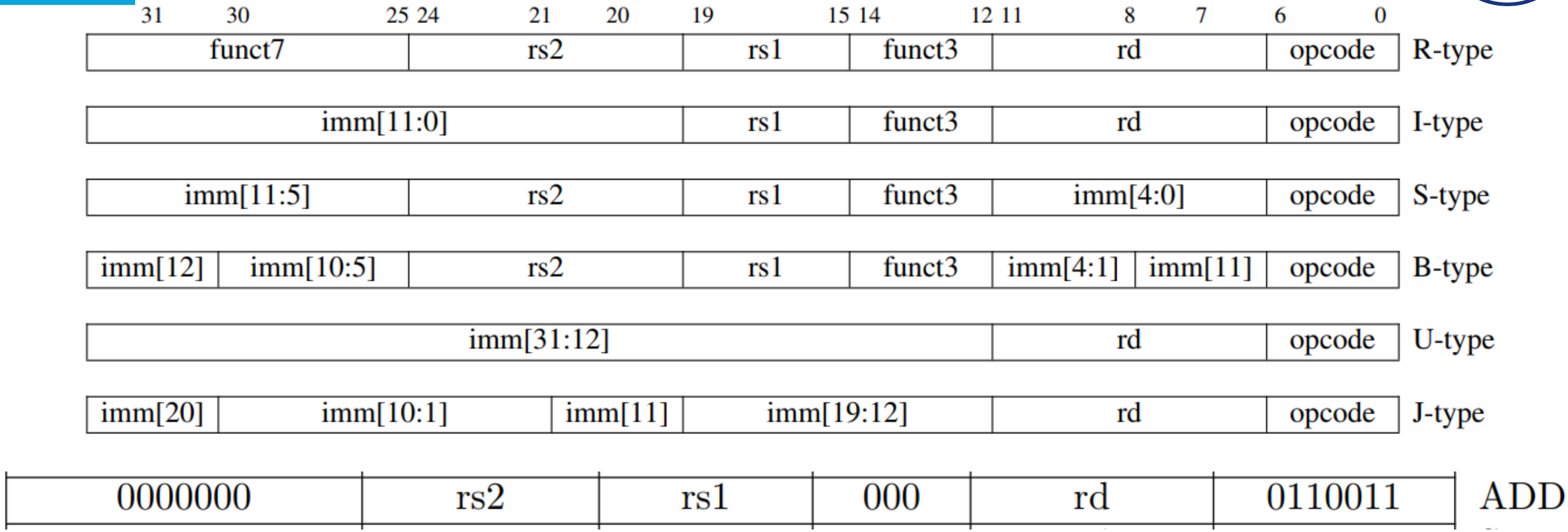


FIGURE 2.46 Growth of x86 instruction set over time. While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors.

RISC-V Instruction Types



add x9, x20, x21

func7 (7-bit)	rs2 (5-bit)	rs1 (5-bit)	funct3 (3-bit)	rd (5-bit)	opcode (7-bit)
0000000	10101	10100	000	01001	0110011

0x015A04B3

RISC-V R-Type Instructions

0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and

RISC-V Fields

RISC-V Fields

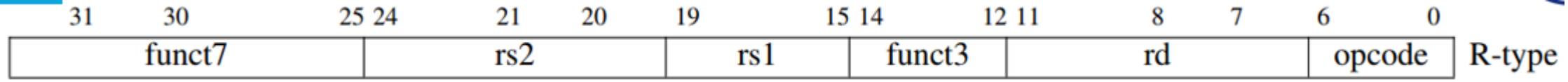
RISC-V fields are given names to make them easier to discuss:

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Here is the meaning of each name of the fields in RISC-V instructions:

- ***opcode***: Basic operation of the instruction, and this abbreviation is its traditional name.
- *rd:* The register destination operand. It gets the result of the operation.
- *funct3:* An additional opcode field.
- *rs1:* The first register source operand.
- *rs2:* The second register source operand.
- *funct7:* An additional opcode field.

R-Type Instruction Problem



Yazılım geliştirmede array yapıları sıklıkla kullanılır. Array elemanları bellekte bulunur ve üzerinde işlem yapılabilmesi için register file içerisinde bir registera getirilmesi (load) gerekir.

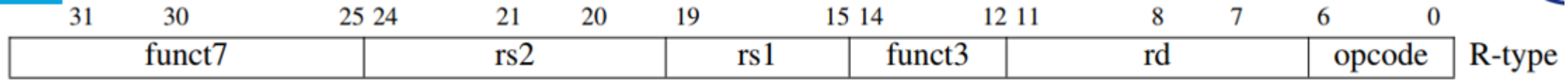
```
addi a0,zero,5
addi a1,zero,7
lui s2,0x1000
add a2,a0,a1
sw a2,4(s2)
lw a3,4(s2)
nop
```

Load işleminin gerçekleştirilebilmesi için bellek adresi verilmelidir. RISC-V ISA'da bellek adresi bir register ve offset değeri ile verilmektedir. Örnek olarak:

lw a3,4(s2)

Yani s2 registerında bulunan değere 4 ekle ve o bellekteki o adresteki değeri a3 registera yaz

R-Type Instruction Problem



lw a3,4(s2)

Yani s2 registerında bulunan değere 4 ekle ve o bellekteki o adresteki değeri a3 registera yaz

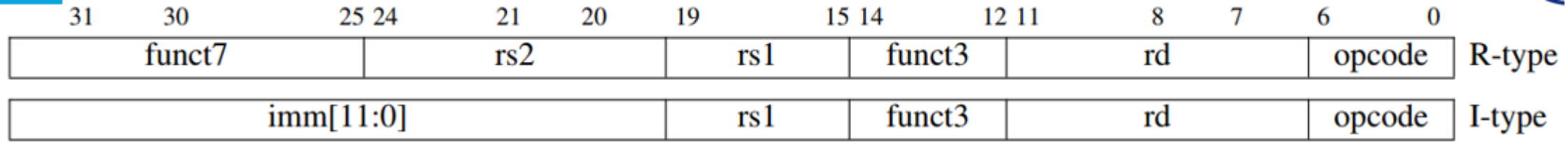
Offset verilmesinin sebebi, array veri tiplerine erişimlerde compiler'ın bir sonraki elemanı registera getirmek için direk offset değerini arttırarak instruction oluşturabilmesidir. Bir sonraki array elemanını register file'a getirmek için

lw a4,8(s2)

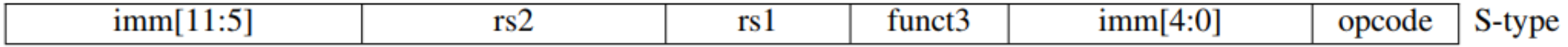
instruction'ı yeterli olacaktır.

Eğer R-Type instruction kullanılırsa offset olarak rs2'ye yazılan değer kullanılırde ve 5-bit ile sadece 32 değer oluşturulurdu. Bu da compiler'ın verimsiz kod üretmesine neden olurdu.

I-Type & S-Type Instructions



I-Type instruction sayesinde 12-bit immediate (imm) alanından adrese offset verilebilmektedir



S-Type instruction ise store işleminde 2 adet kaynak register ve 12-bit immediate (imm) alanından oluşmaktadır. Store işlemi için yazılacak adres rs1 alanından ve yazılacak değer de rs2 alanından alınır. Offset değeri olan 12-bit immediate ise 5 ve 7 bit olarak farklı alanlardadır.

R-Type, I-Type & S-Type Instructions Summary

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011

Instruction	Format	immediate	rs1	funct3	rd	opcode
addi (add immediate)	I	constant	reg	000	reg	0010011
lw (load word)	I	address	reg	010	reg	0000011

Instruction	Format	immed-iate	rs2	rs1	funct3	immed-iate	opcode
sw (store word)	S	address	reg	reg	010	address	0100011