IV. ORGANIZATION OF OTHER COMPUTER SYSTEMS

Basic Types of CPU

Basic Types of CPU

- Accumulator-based
- General-purpose Register Type (GPR)
- Stack Machine

Accumulator-based Instruction Set

- Data Transfer

  load addr ; A <- [addr]

  store addr ; [addr] <- A

- Arithmetic Operations

  add addr ; A <- A + [addr]

  sub addr ; A <- A - [addr]

  mul addr ; A <- A * [addr]

  div addr ; A <- A / [addr]

Accumulator-based Instruction Set

- Logical Operations

```
and addr        ; A <- A and [addr]
or addr         ; A <- A or [addr]
not             ; A <- not A
not addr        ; A <- not [addr]
```

GPR Type Instruction Set

- Data Transfer

```
mov dst, src          ; dst <- src
mov dst, [src]        ; dst <- [src]
mov [dst], src        ; [dst] <- src
```

GPR Type Instruction Set

- Arithmetic Operations

```
add dst, src        ; dst <- dst + src
sub dst, src        ; dst <- dst – src
mul dst, src        ; dst <- dst * src
div dst, src        ; dst <- dst / src
```

GPR Type Instruction Set

- Logical Operations

```
and dst, src        ; dst <- dst and src
or dst, src         ; dst <- dst or src
not dst             ; dst <- not dst
```

Stack Machine Instruction Set

- Data Transfer

```
push data       ; push immediate
push addr       ; push memory variable
pop addr        ; pop memory variable
pop X           ; pop an address
```

Stack Machine Instruction Set

- Arithmetic Operations

```
add     ; push(pop() + pop())
sub     ; push(pop() – pop())
mul     ; push(pop() * pop())
div     ; push(pop() / pop())
```

Stack Machine Instruction Set

- Logical Operations

```
and     ; push(pop() and pop())
or      ; push(pop() or pop())
not     ; push(not pop())
```

More Examples

- Write assembly codes appropriate for each of the three different types of CPU for the following expression:

p := ((b + a) * c) / a

More Examples

- Accumulator-based

p := ((b + a) * c) / a

More Examples

- Accumulator-based

p := ((b + a) * c) / a

load b

More Examples

- Accumulator-based

p := ((b + a) * c) / a

load b

add a

More Examples

- Accumulator-based

p := ((b + a) * c) / a

load b

add a

mul c

More Examples

- Accumulator-based

p := ((b + a) * c) / a

load b

add a

mul c

div a

More Examples

- Accumulator-based

p := ((b + a) * c) / a

load b

add a

mul c

div a

store p

More Examples

- GPR-type

p := ((b + a) * c) / a

More Examples

- GPR-type

  p := ((b + a) * c) / a

  mov R0, b

More Examples

- GPR-type

p := ((b + a) * c) / a

mov R0, b

mov R1, a

More Examples

- GPR-type

p := ((b + a) * c) / a

mov R0, b

mov R1, a

add R0, R1

More Examples

- GPR-type

p := ((b + a) * c) / a

mov R0, b

mov R1, a

add R0, R1

mov R2, c

More Examples

- GPR-type

p := ((b + a) * c) / a

```
mov R0, b
mov R1, a
add R0, R1
mov R2, c
mul R0, R2
```

More Examples

- GPR-type

p := ((b + a) * c) / a

```
mov R0, b
mov R1, a
add R0, R1
mov R2, c
mul R0, R2
div R0, R1
```

More Examples

- GPR-type

p := ((b + a) * c) / a

```
mov R0, b
mov R1, a
add R0, R1
mov R2, c
mul R0, R2
div R0, R1
mov p, R0
```

More Examples

- Stack Machine

p := ((b + a) * c) / a

More Examples

- Stack Machine

p := ((b + a) * c) / a

push b

More Examples

- Stack Machine

p := ((b + a) * c) / a

push b

push a

More Examples

- Stack Machine

p := ((b + a) * c) / a

push b

push a

add

More Examples

- Stack Machine

p := ((b + a) * c) / a

push b

push a

add

push c

More Examples

- Stack Machine

p := ((b + a) * c) / a

push b
push a
add
push c
mul

More Examples

- Stack Machine

p := ((b + a) * c) / a

```
push b          mul
push a          push a
add
push c
```

More Examples

- Stack Machine

p := ((b + a) * c) / a

```
push b
push a
add
push c
mul
push a
div
```

More Examples

- Stack Machine

p := ((b + a) * c) / a

```
push b          mul
push a          push a
add             div
push c          pop p
```

More Exercises

- For the following expression, write assembly codes appropriate for each of the three different types of CPU:

```
p := (a/b) + ((b+a) * c)/a
```

IV. ORGANIZATION OF OTHER COMPUTER SYSTEMS

Instruction Set Format

Instruction Set Format

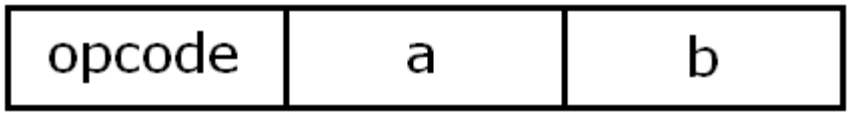
- is the number of operands given to an instruction or opcode
- N-address instruction means there are N operands that need to be given to that instruction
- Most modern computers do not use a single n-addressing for their instruction sets

Instruction Set Format

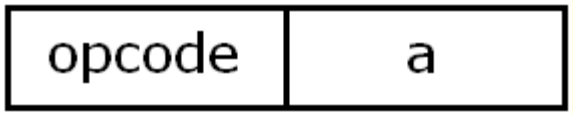
- Here are some possible instruction set formats:

3-Address Instruction

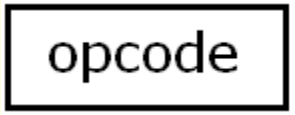
2-Address Instruction



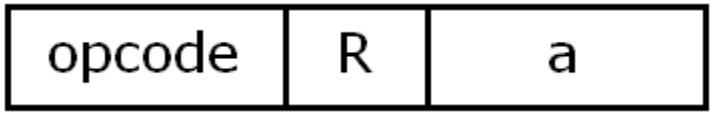
1-Address Instruction



0-Address Instruction



1½-Address Instruction

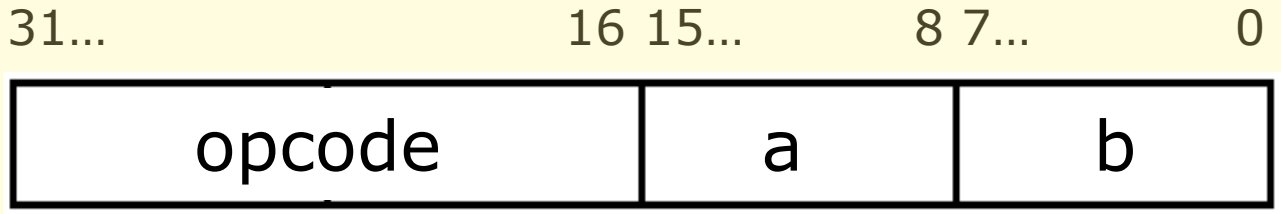


Recall

- add byte[x], al

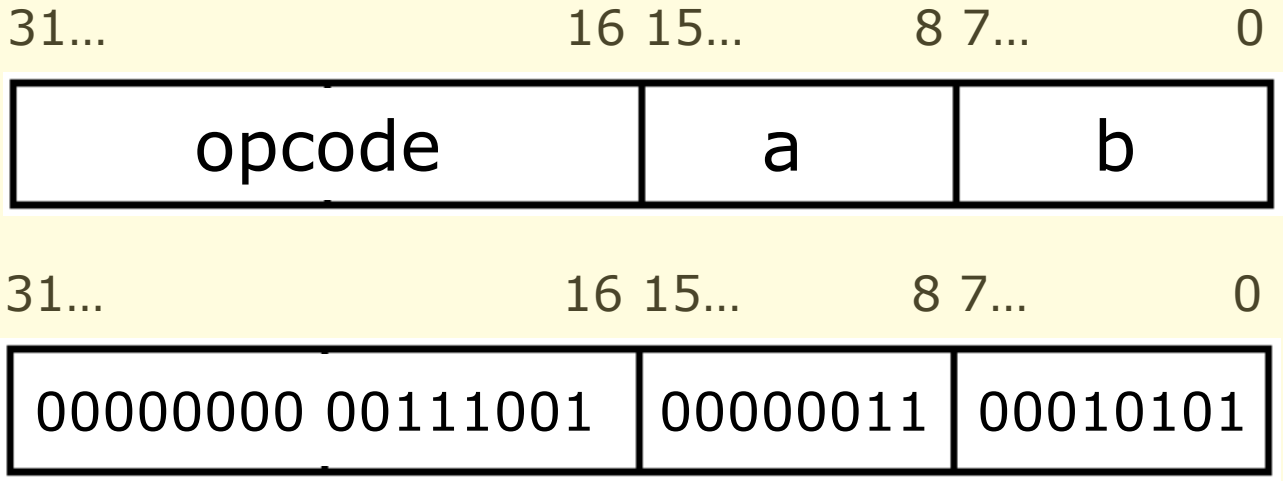
Recall

- add byte[x], al
- A translation of this instruction into a 32-bit machine (for example) can be:



Recall

- add byte[x], al
- A translation of this instruction into a 32-bit machine (for example) can be:



- if x=3 and al=21

Basis for Designing an Instruction Set

- Shorter instructions are better.
- All instructions should fit into the size of the machine word, that is the number of bits of the opcode and address that the CPU can handle.

Expanding Opcode

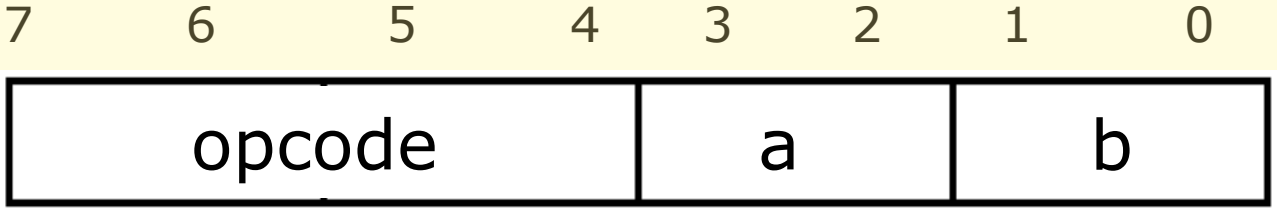
- Expanding opcode is a way of determining if a certain instruction set can fit into a machine of a specified machine word length.

Expanding Opcode

- Example: Can an instruction set with the following instructions fit into an 8-bit CPU? (Assume 2 bits are needed per operand)

13	2-address instructions
10	1-address instructions
9	0-address instructions
---	---
32	different opcodes

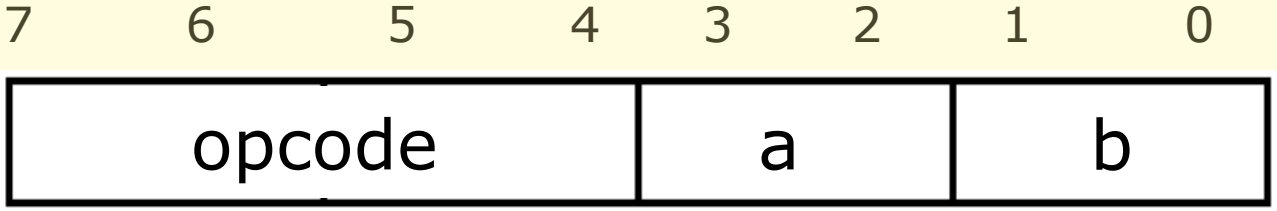
Expanding Opcode



- If each operand uses 2 bits, there are 4 bits left to represent the instruction. Therefore, there can be at most $2^4 = 16$ different opcodes with 2 operands.
- The 8-bit CPU can handle the first requirement:

  13 2-address instructions

Expanding Opcode

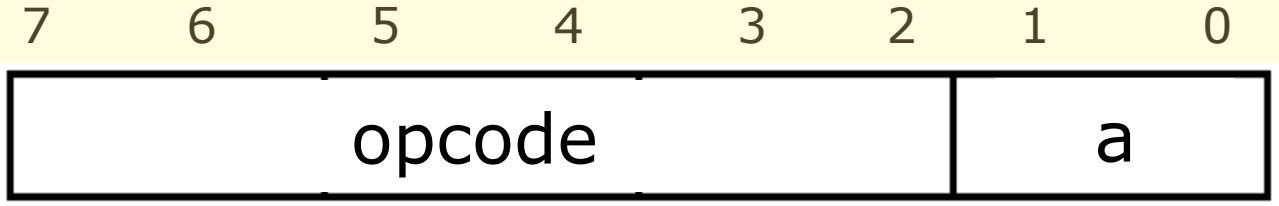


- The following combinations can be used to represent the 13 instructions:

0000	0100	1000	1100	
0001	0101	1001	1101	X
0010	0110	1010	1110	X
0011	0111	1011	1111	X

Expanding Opcode

- If there are unused combinations 1101, 1110 and 1111, they can be used for the next requirement: ten 1-address instructions



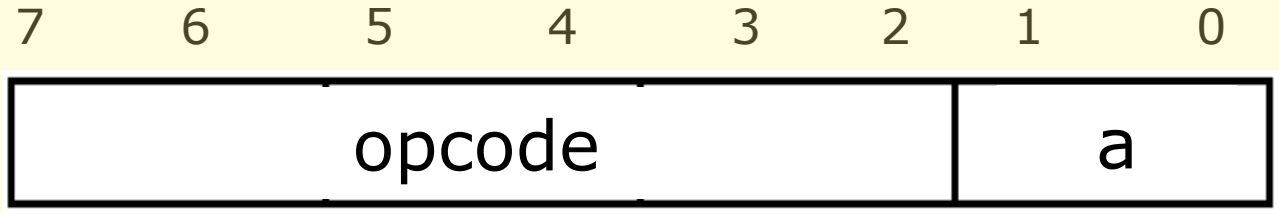
1101 0 0

1101 0 1

1101 1 0

1101 1 1

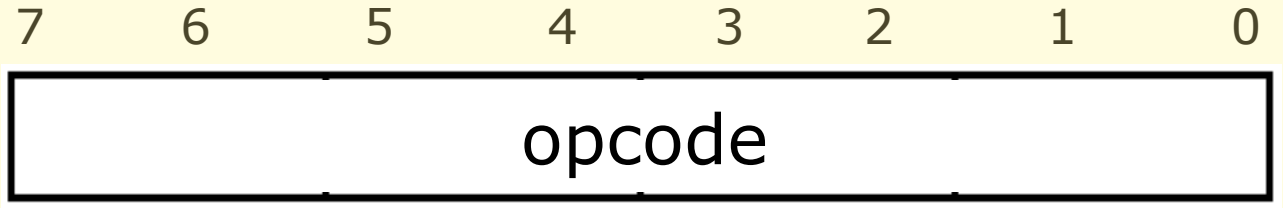
Expanding Opcode



1110	0 0	
1110	0 1	
1110	1 0	
1110	1 1	
1111	0 0	
1111	0 1	
1111	1 0	X
1111	1 1	X

Expanding Opcode

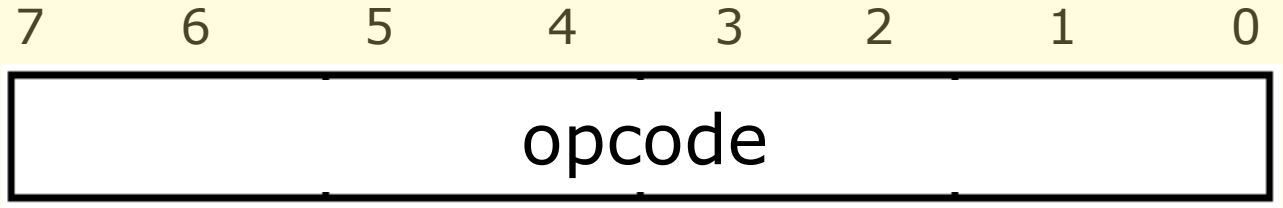
- Again there are unused combinations, but are they enough for the last set: 9 0-address instructions



1111	1	0	0 0
1111	1	0	0 1
1111	1	0	1 0
1111	1	0	1 1

Expanding Opcode

- Again there are unused combinations, but are they enough for the last set: 9 0-address instructions



1111	1	1	0 0
1111	1	1	0 1
1111	1	1	1 0
1111	1	1	1 1

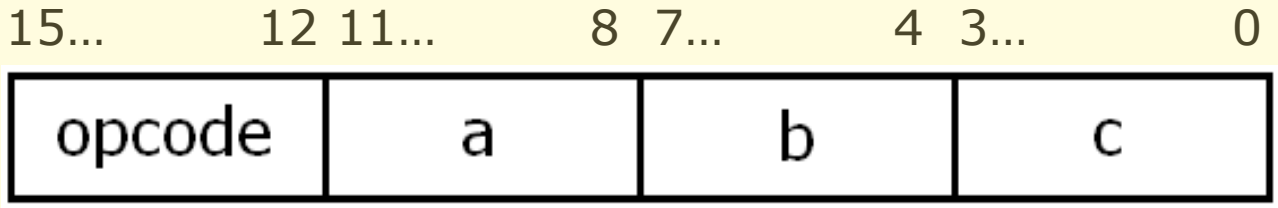
Expanding Opcode

- Another Example: Can an instruction set with the following instructions fit into a 16-bit CPU?

15	3-address instructions
14	2-address instructions
31	1-address instructions
16	0-address instructions
---	---
76	different opcodes

Expanding Opcode

- First let us consider the fact that if we have 16 bits, we can divide these 16-bits into four so as to contain the opcode and three operands.



- The operand can be given four bits each. The opcode is four bits as well. In this case, we see that we can have only 2^4 different opcodes.

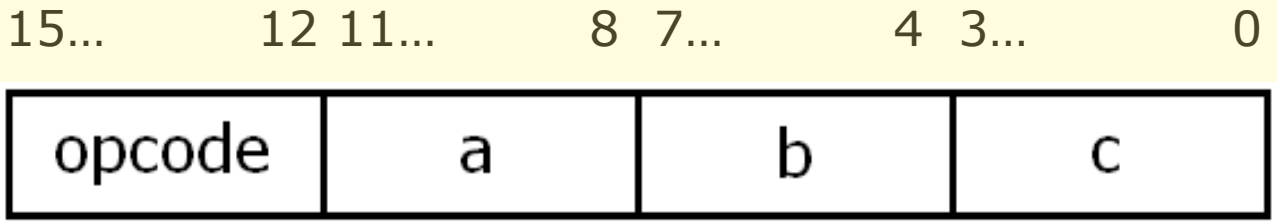
Expanding Opcode

- So, can an instruction set with the following instructions fit into the 16-bit CPU?

15	3-address instructions
14	2-address instructions
31	1-address instructions
16	0-address instructions
---	---
76	different opcodes

Expanding Opcode

- If each operand uses 4-bits, there can be 16 3-address opcodes for the machine.



- So it can handle the first requirement:
  15 3-address instructions

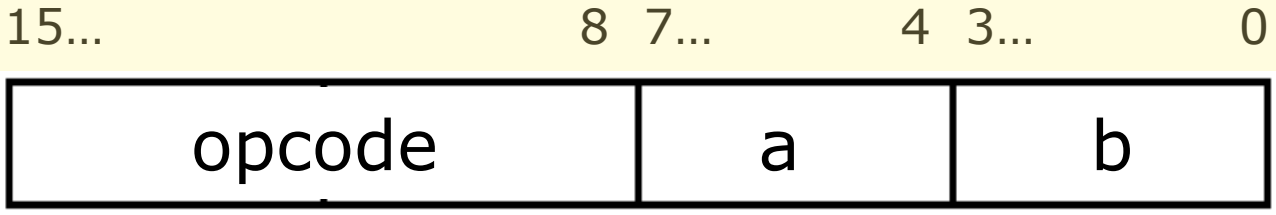
Expanding Opcode

- Second requirement:

15	3-address instructions
14	2-address instructions
31	1-address instructions
16	0-address instructions
---	---
76	different opcodes

Expanding Opcode

- Combinations 0000-1110 are used for the 15 instructions. Notice one unused combination 1111. We will use this remaining combination to expand the opcodes for the other instruction formats.
- For the 14 2-address instructions, the opcode expands occupying the 4-bits earlier given to one of the three operands.

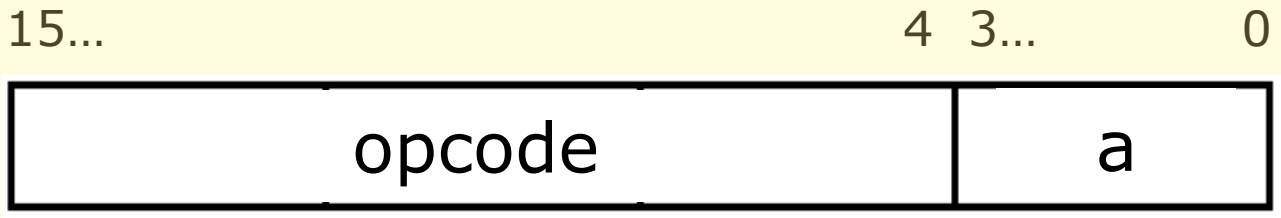


Expanding Opcode

- Combinations 1111 0000 – 1111 1101 are now used for the 14 instructions.
- This time, there are two unused combinations 1111 1110 and 1111 1111.
- Again, we will use this remaining combinations to expand the opcodes for the other instruction formats.

Expanding Opcode

- For the 31 1-address instructions, we will have:

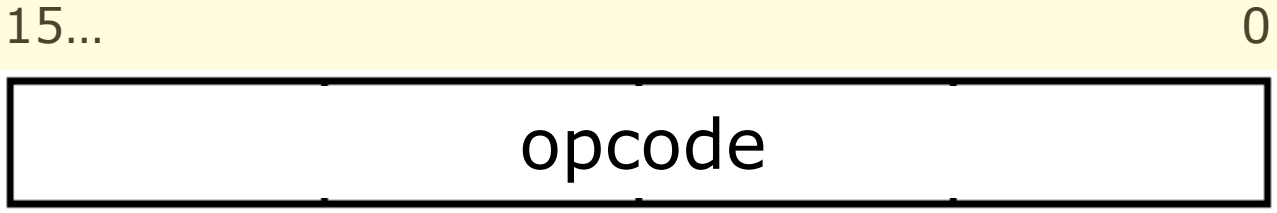


1111 1110 0000
. . .
1111 1110 1111
} 16 1-address instructions

1111 1111 0000
. . .
1111 1111 1110
} 15 1-address instructions

Expanding Opcode

- Now for the 16 0-address instruction:



Unused from previous format: 1111 1111 1111

1111 1111 1111 0000

. . .

1111 1110 1111 1111

} 16 0-address instructions

Expanding Opcode

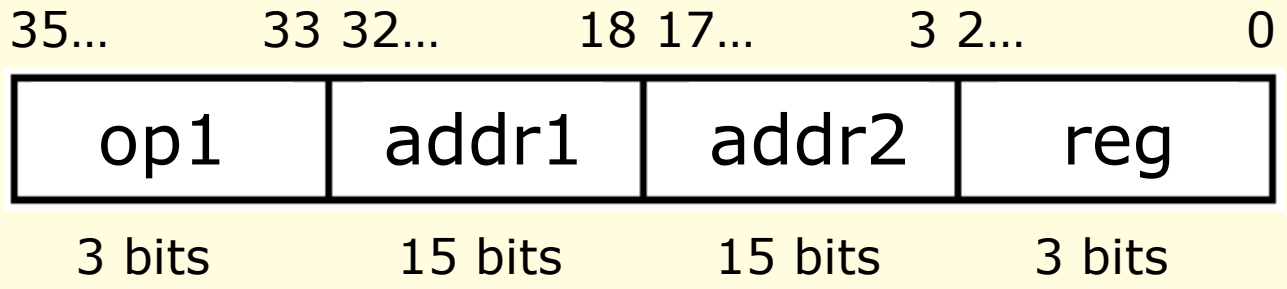
- Design an expanding opcode to allow all of the following to be encoded in a 36-bit instruction.

  7 instructions with two 15-bit addresses and one 3-bit register number

  500 instructions with one 15-bit address and one 3-bit register number

  50 instructions with no addresses or registers

Expanding Opcode



- If 3 bits are left for the opcode, we could have 8 possible combinations for it.
- Since we need 7 instructions of this kind, we have one combination left to use in the next format.

Expanding Opcode

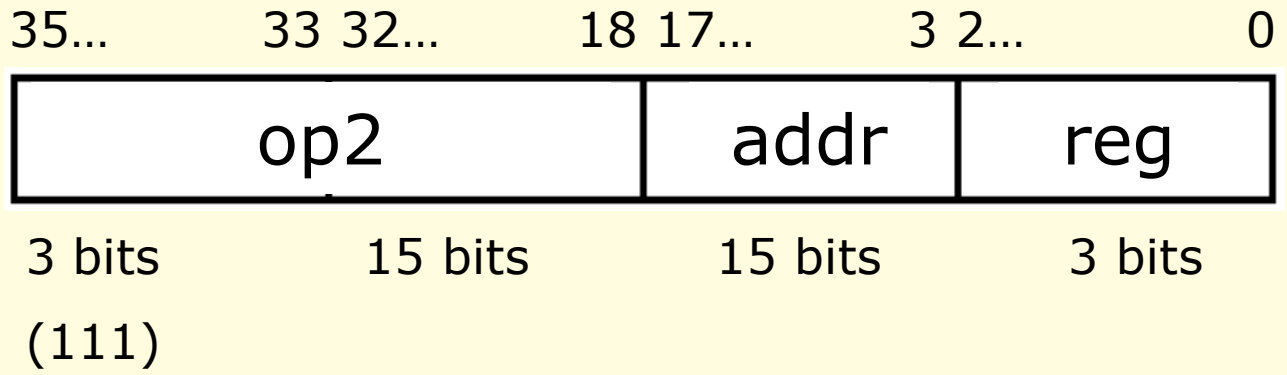
- Design an expanding opcode to allow all of the following to be encoded in a 36-bit instruction.

  7 instructions with two 15-bit addresses and one 3-bit register number

  500 instructions with one 15-bit address and one 3-bit register number

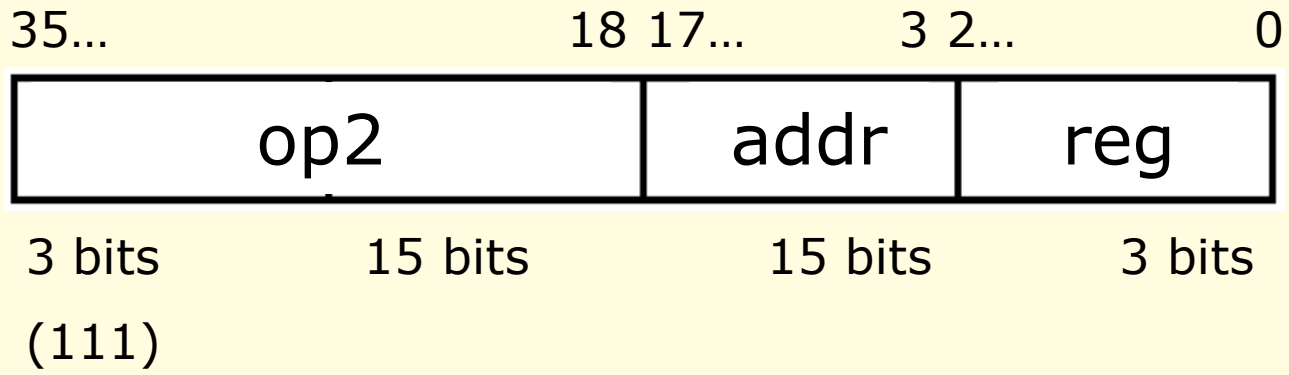
  50 instructions with no addresses or registers

Expanding Opcode



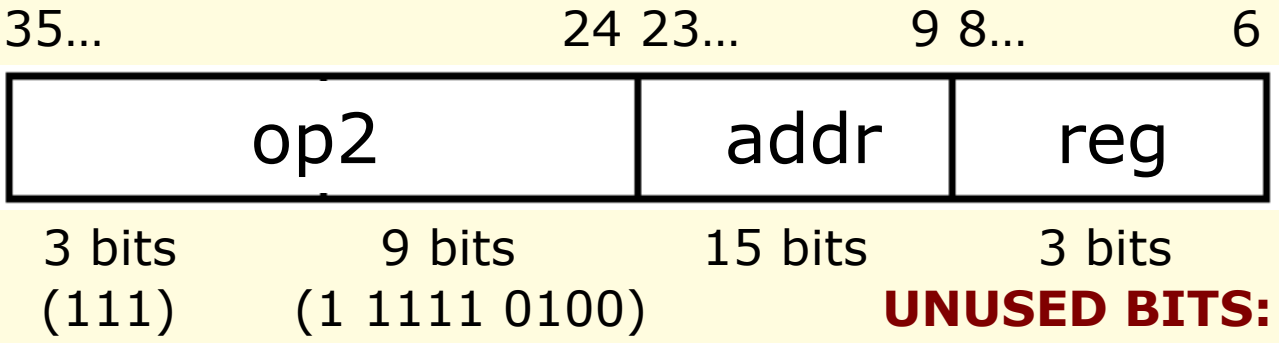
- The unused combination 111 will now be the starting address of the next format.
- Since there are now only 2 operands, there is an extra 15 bits which can be used for the 500 instructions.

Expanding Opcode



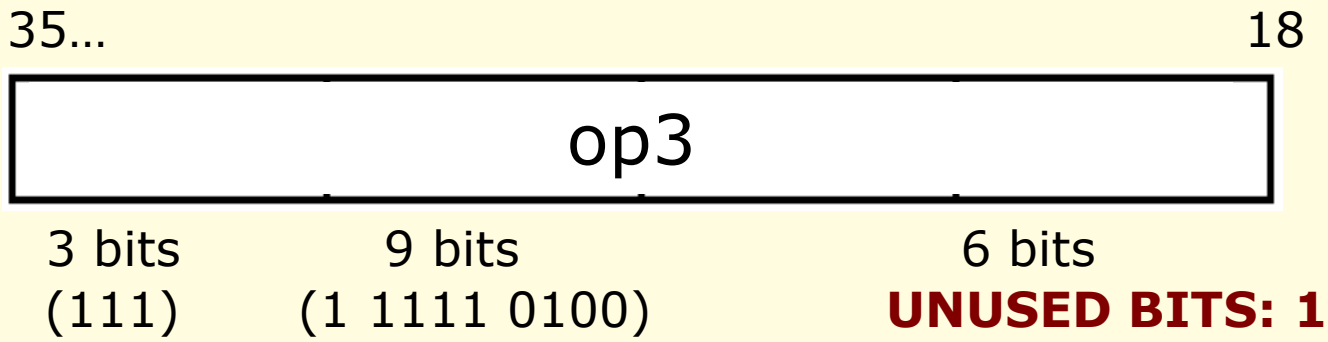
- However, 9 bits are enough to accommodate 500 instructions for this format.
- Therefore, there are still many unused combinations which can be used for the last type of instruction.

Expanding Opcode



- Again, remaining combinations will be used for the last format needed.

Expanding Opcode



- Here, 6 additional bits are necessary to accommodate the last 50 instructions.