ECE/CS 552: Instruction Sets – MIPS

© Prof. Mikko Lipasti

Lecture notes based in part on slides created by Mark Hill, David Wood, Guri Sohi, John Shen and Jim Smith

Instructions (252/354 Review)

Instructions are the “words” of a computer

Instruction set architecture (ISA) is its vocabulary

This defines most of the interface to the processor (not quite everything)

Implementations can and do vary

- Intel 486->Pentium->P6->Core Duo->Core i7

Instruction Sets

- MIPS/MIPS-like ISA used in 552
  - Simple, sensible, regular, easy to design CPU
- Most common: x86 (IA-32) and ARM
  - x86: Intel Pentium/Core i7, AMD Athlon, etc.
  - ARM: cell phones, embedded systems
- Others:
  - PowerPC (IBM servers)
  - SPARC (Sun)
- We won’t write programs in this course

Forecast

- Basics
- Registers and ALU ops
- Memory and load/store
- Branches and jumps
- Addressing modes

Basics

- C statement

  f = (g + h) – (i + j)

- MIPS instructions

  add t0, g, h

  add t1, i, j

  sub f, t0, t1

- Multiple instructions for one C statement

Opcode/Mnemonic: Specifies operation

Operands: Input and output data

Source

Destination

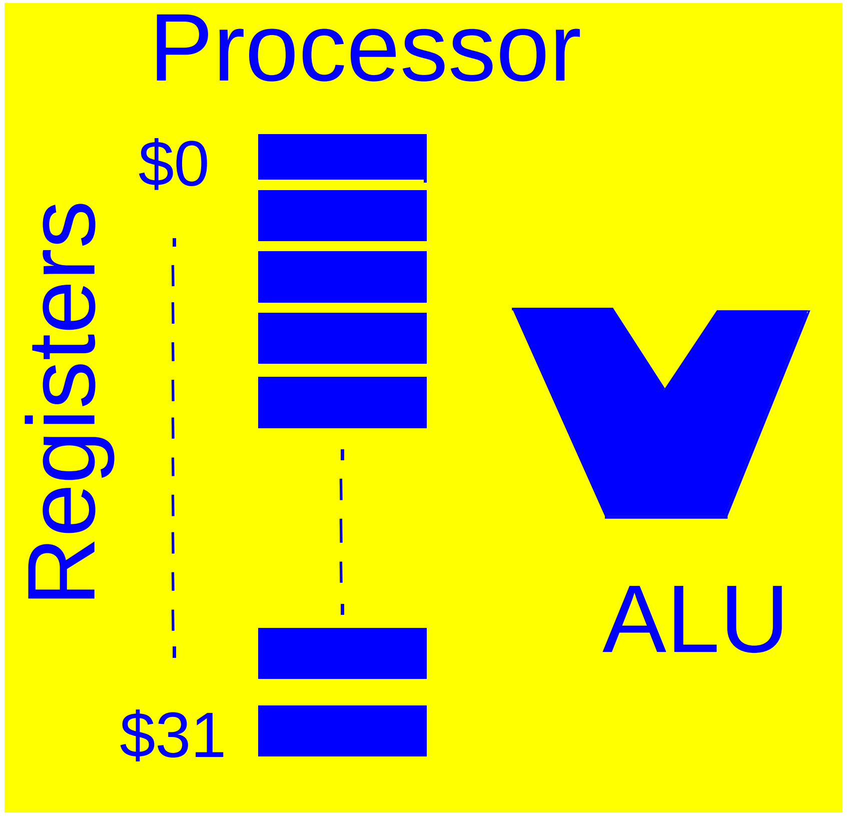
Why not bigger instructions?

- Why not "f = (g + h) – (i + j)" as one instruction?
- Church's thesis: A very primitive computer can compute anything that a fancy computer can compute – you need only logical functions, read and write memory, and data-dependent decisions
- Therefore, ISA selected for practical reasons:
  - Performance and cost, not computability
- Regularity tends to improve both
  - E.g. H/W to handle arbitrary number of operands is complex and slow and UNNECESSARY

Registers and ALU ops

- MIPS: Operands are registers, not variables
  - add $8, $17, $18
  - add $9, $19, $20
  - sub $16, $8, $9
- MIPS has 32 registers $0-$31
- $8 and $9 are temps, $16 is f, $17 is g, $18 is h, $19 is i and $20 is j
- MIPS also allows one constant called "immediate"
  - Later we will see immediate is restricted to 16 bits

Registers and ALU



ALU ops

- Some ALU ops:
  - add, addi, addu, addiu (immediate, unsigned)
  - sub …
  - mul, div – wider result
    - 32b x 32b = 64b product
    - 32b / 32b = 32b quotient and 32b remainder
  - and, andi
  - or, ori
  - sll, srl
- Why registers?
  - Short name fits in instruction word: $\log_2(32) = 5$ bits
- But are registers enough?

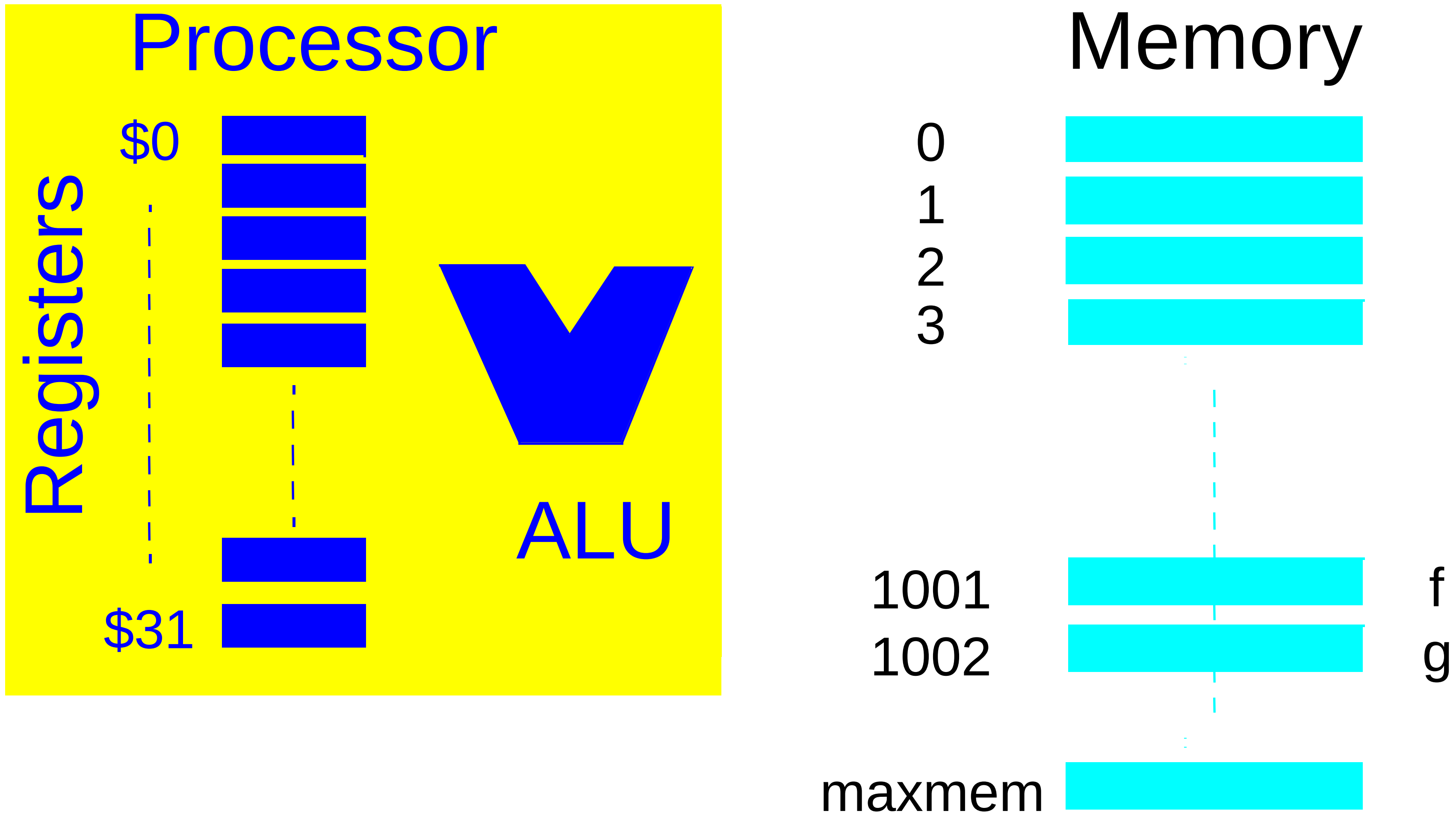
Memory and Load/Store

- Need more than 32 words of storage
- An array of locations M[j] indexed by j
- Data movement (on words or integers)
  - Load word for register <= memory

    ```
    lw $17, 1002 # get input g
    ```
  - Store word for register => memory

    ```
    sw $16, 1001 # save output f
    ```

Memory and load/store



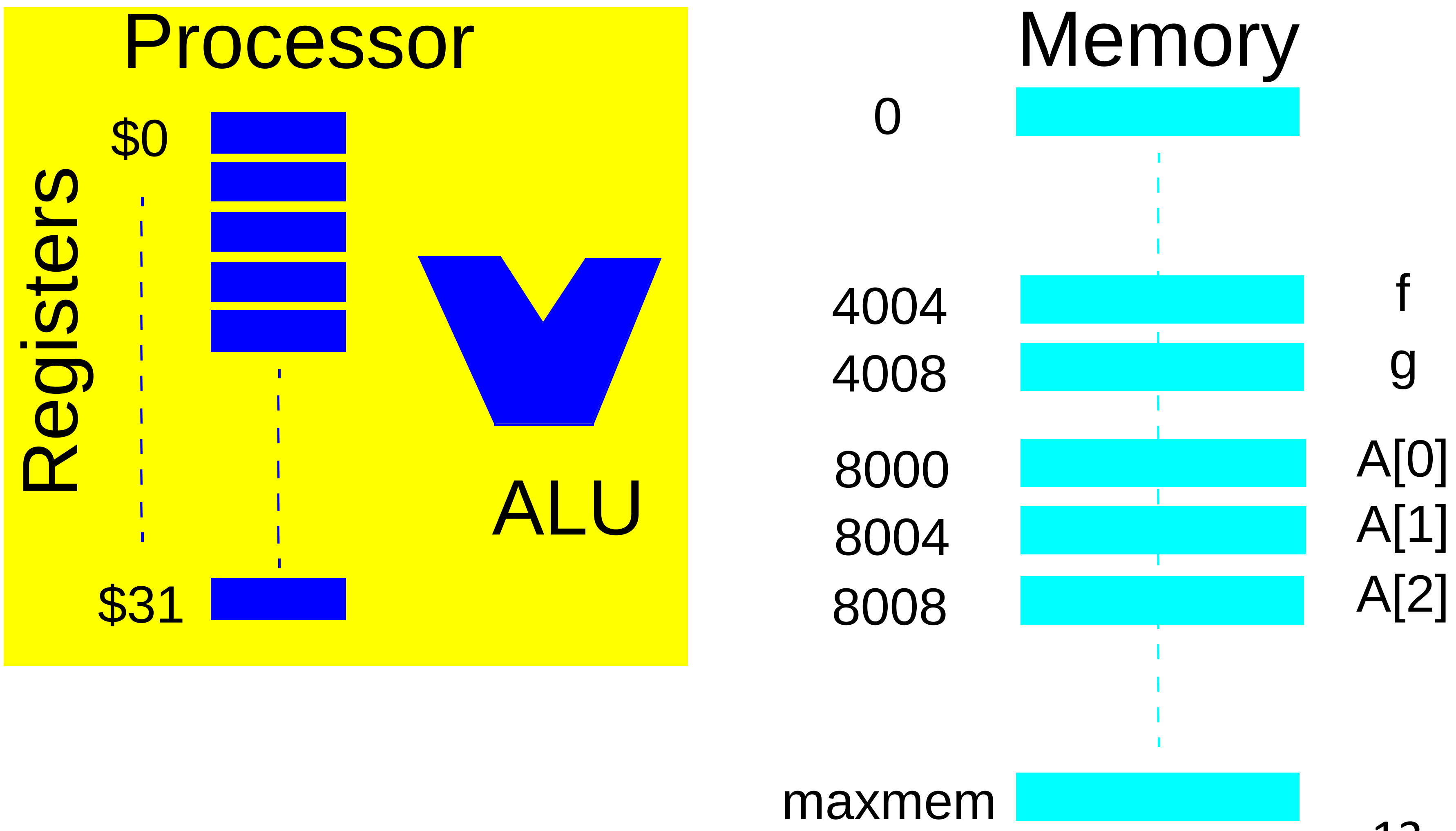
Memory and load/store

- Important for arrays

```
A[i] = A[i] + h
# $8 is temp, $18 is h, $21 is (i x 4)
# Astart is &A[0] is 0x8000
lw $8, Astart($21) # or 8000($21)
add $8, $18, $8
sw $8, Astart($21)
```

- MIPS has other load/store for bytes and halfwords

Memory and load/store



Branches and Jumps

```
While ( i != j) {
     j= j + i;
     i= i + 1;
}
```

```
# $8 is i, $9 is j
Loop:  beq $8, $9, Exit
       add $9, $9, $8
       addi $8, $8 , 1
       j    Loop
Exit:
```

Branches and Jumps

- What does beq do really?
  - read $, read $9, compare
  - Set PC = PC + 4 or PC = Target
- To do compares other than = or !=
  - E.g.

    ```
    blt $8, $9, Target # pseudoinstruction
    ```
  - Expands to:

    ```
    slt $1, $8, $9  # $1==($8<$9)==($8-$9)<0
    bne $1, $0, Target  # $0 is always 0
    ```

Branches and Jumps

- Other MIPS branches

  beq $8, $9, imm # if ($8==$9) PC = PC + imm<< 2 else PC += 4;

  bne …

  slt, sle, sgt, sge
- Unconditional jumps

  j addr # PC = addr

  jr $12 # PC = $12

  jal addr # $31 = PC + 4; PC = addr;

    (used for function calls)

MIPS Machine Language

- All instructions are 32 bits wide
- Assembly: add $1, $2, $3
- Machine language:

```
33222222222211111111110000000000
10987654321098765432109876543210
|000000|00010|00011|00001|00000|010000|
```

000000	00010	00011	00001	00000	010000
alu-rr	2	3	1	zero	add/signed

Instruction Format

- R-format

– Opc	rs	rt	rd	shamt	function
– 6	5	5	5	5	6

- Digression:
  - How do you store the number 4,392,976?
    - Same as add $1, $2, $3
- Stored program: instructions are represented as numbers
  - Programs can be read/written in memory like numbers
- Other R-format: addu, sub, …

Instruction Format

- Assembly: lw $1, 100($2)
- Machine: 100011 00010 00001 0000000001100100

100011	00010	00001	0000000001100100
lw	2	1	100 (in binary)

- I-format
  - Opc rs rt address/immediate
  - 6 5 5 16

Instruction Format

- I-format also used for ALU ops with immediates
  - addi $1, $2, 100
  - 001000 00010 00001 0000000001100100
- What about constants larger than 16 bits
  - Outside range: [-32768, 32767]?

  1100 0000 0000 0000 1111?

  lui $4, 12 # $4 == 0000 0000 1100 0000 0000 0000 0000 0000

  ori $4, $4, 15 # $4 == 0000 0000 1100 0000 0000 0000 1111
- All loads and stores use I-format

Instruction Format

- beq $1, $2, 7
  000100 00001 00010 0000 0000 0000 0111
  PC = PC + (0000 0111 << 2) # word offset
- Finally, J-format
  J address

Opcode	addr
6	26

- Addr is weird in MIPS:
  addr = 4 MSB of PC // addr // 00

Summary: Instruction Formats

R:	opcode	rs	rt	rd	shamt	function
	6	5	5	5	5	6
I:	opcode	rs	rt	address/immediate		
	6	5	5	16		
J:	opcode	addr				
	6	26				

- Instruction decode:
  - Read instruction bits
  - Activate control signals

Procedure Calls

- Calling convention is part of ABI
  - Caller
    - Save registers
    - Set up parameters
    - Call procedure
    - Get results
    - Restore registers
  - Callee
    - Save more registers
    - Do some work, set up result
    - Restore registers
    - Return
- Jal is special, otherwise just software convention

Procedure Calls

- Stack: parameters, return values, return address
- Stack grows from larger to smaller addresses (arbitrary)
- $29 is stack pointer; points just beyond valid data
- Push $2:

```
addi $29, $29, -4
sw $2, 4($29)
```

- Pop $2:

```
lw $2, 4($29)
addi $29, $29, 4
```

Procedure Example

```
Swap(int v[], int k) {
  int temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

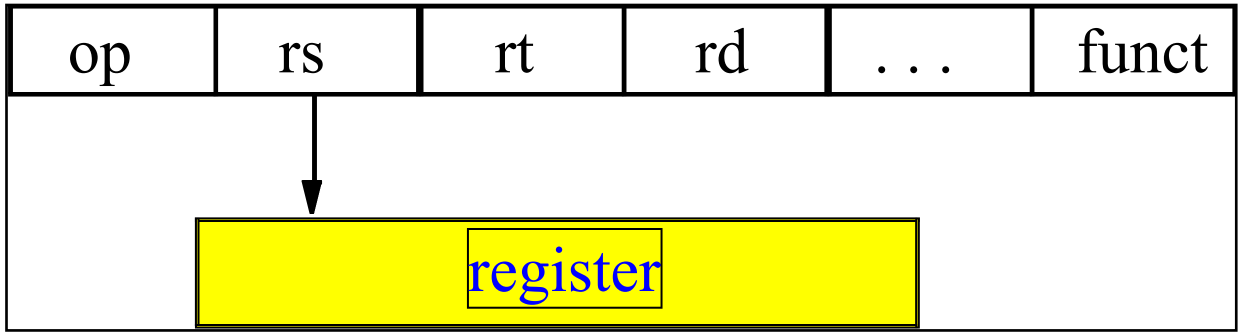
$4 is v[] & $5 is k -- 1st & 2nd incoming argument
$8, $9 & $10 are temporaries that callee can use w/o saving

```
swap:  add $9,$5,$5  # $9 = k+k
       add $9,$9,$9  # $9 = k*4
       add $9,$4,$9  # $9 = v + k*4 = &(v[k])
       lw  $8,0($9)  # $8 = temp = v[k]
       lw  $10,4($9) # $10 = v[k+1]
       sw  $10,0($9) # v[k] = v[k+1]
       sw  $8,4($9)  # v[k+1] = temp
       jr  $31       # return
```

Addressing Modes

- There are many ways of accessing operands
- Register addressing:

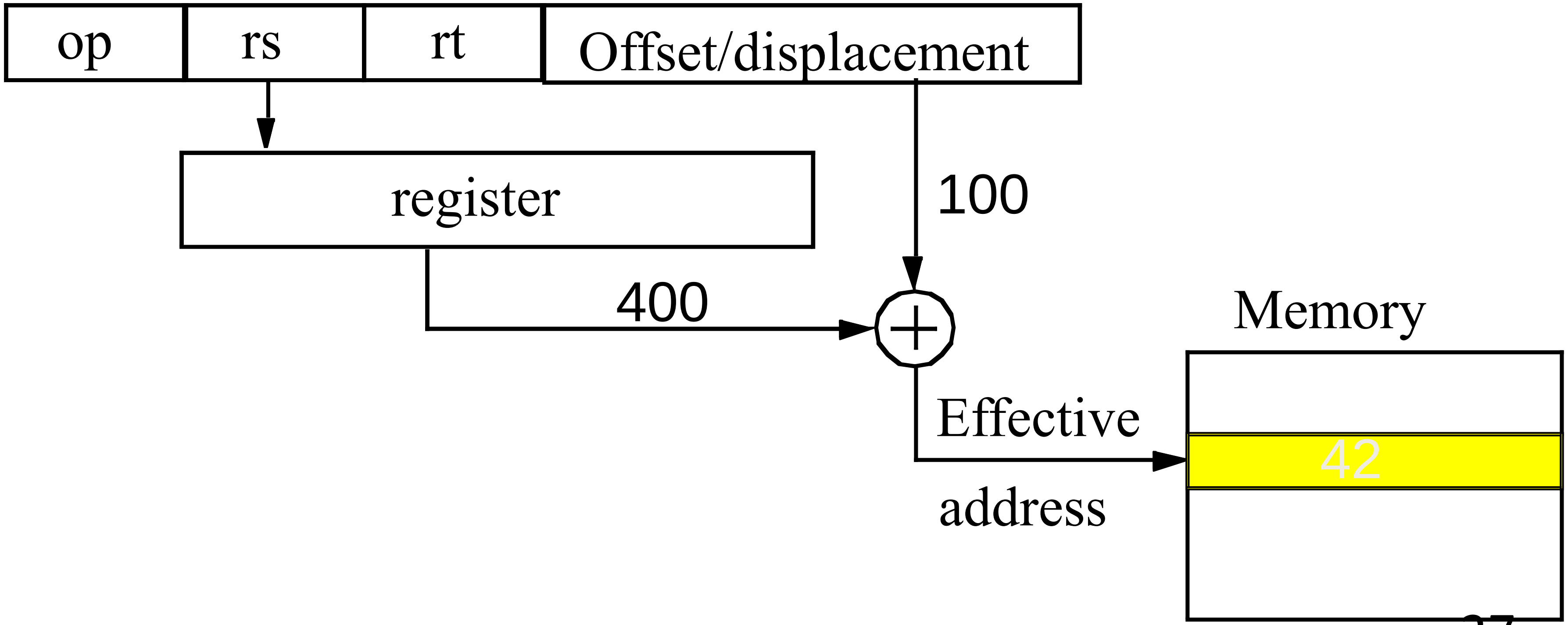
  add $1, $2, $3



Addressing Modes

- Base addressing (aka displacement)

  lw $1, 100($2) # $2 == 400, M[500] == 42



Addressing Modes

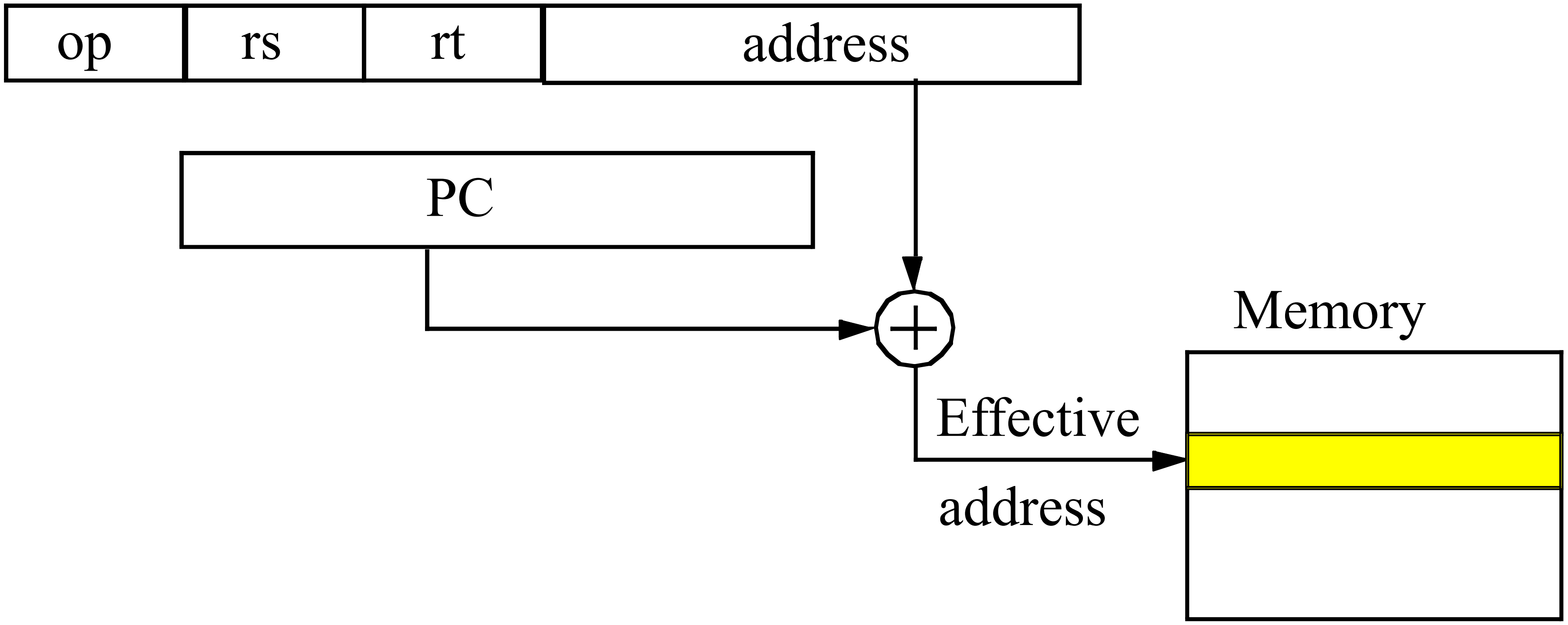
- Immediate addressing

  addi $1, $2, 100

op	rs	rt	immediate

Addressing Modes

- PC relative addressing
  beq $1, $2, 100 # if ($1==$2) PC = PC + 100



Summary

- Many options when designing new ISA
  - Backwards compatibility limits options
- Simple and regular makes designers’ life easier
  - Constant length instructions, fields in same place
- Small and fast minimizes memory footprint
  - Small number of operands in registers
- Compromises are inevitable
  - Pipelining should not be hindered
- Optimize for common case

ECE/CS 552: Instruction Sets – x86

Lecture notes based in part on slides created by Mark Hill, David Wood, Guri Sohi, John Shen and Jim Smith

Instruction Sets

- MIPS/MIPS-like ISA used in 552
  - Simple, sensible, regular, easy to design CPU
- Most common: x86 (IA-32) and ARM
  - x86: laptops, desktops, servers
  - ARM: smartphones, embedded systems
- Others:
  - PowerPC (IBM servers)
  - SPARC (Sun)
- Example of complex (CISC) ISA: x86

ISA Tradeoffs

- High-level language “semantic gap”
  - Motivated complex ISA
  - E.g. direct support for function call
    - Allocate stack frame
    - Push parameters
    - Push return address
    - Push stack pointer
    - In reverse order for return
  - Complex addressing modes
    - Arrays, pointers, indirect (pointers to pointers)

ISA Tradeoffs

- Compiler technology improved dramatically
- Closed "semantic gap" with software automation
- Enabled better optimization
  - Leaf functions don't need a stack frame
  - Redundant loads/stores to/from memory can be avoided
- No need for direct ISA support for high-level semantics

- "RISC" revolution in 1980s
  - IBM 801 project=>PowerPC, MIPS, SPARC, ARM

ISA Tradeoffs

- Minimize what?
  - Instrs/prog x cycles/instr x sec/cycle !!!
- If memory is limited, dense instructions are important
  - Older x86, IBM 360, DEC VAX: CISC ISA are dense
- In 1980s technology, simple ISAs made sense: RISC
  - As technology changes, computer design options change
- For high speed, pipelining and ease of pipelining is important
  - Even CISC can be pipelined effectively (ECE752)

- Legacy support, binary compatibility are key

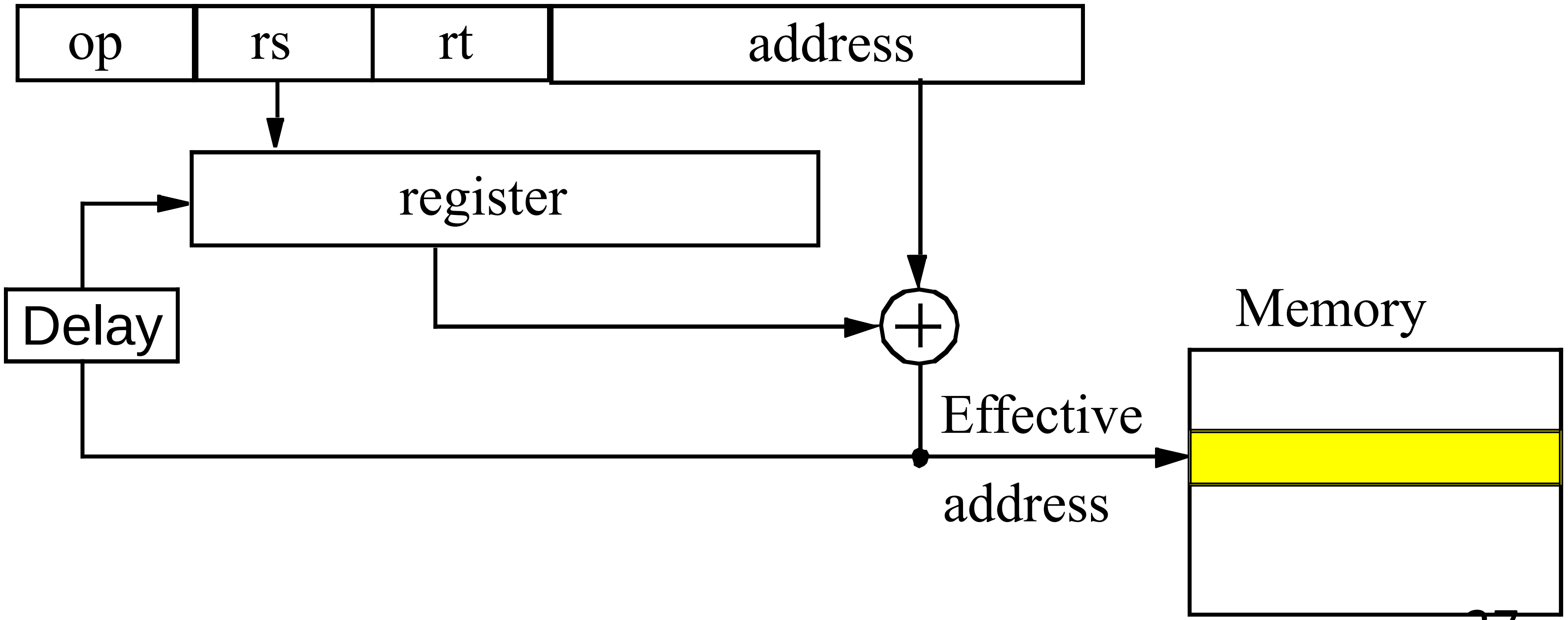
Addressing Modes

- Not found in MIPS:
  - Indexed: add two registers – base + index
  - Indirect: M[M[addr]] – two memory references
  - Autoincrement/decrement: add operand size
  - Autoupdate – found in PowerPC, PA-RISC
    - Like displacement, but update base register

Addressing Modes

- Autoupdate

  lwupdate $1,24($2) # $1 = M[$2+24]; $2 = $2 + 24



Addressing Modes

```
for(i=0; i < N, i += 1)
  sum += A[i];
# $7 is sum, $8 is &a[i], $9 is N,$2 is tmp, $3 is i*4
```

Inner loop:

```
    lw $2, 0($8)
    addi $8, $8, 4
    add $7, $7, $2
```

Or:

```
    lwupdate $2, 4($8)
    add $7, $7, $2
```

Where's the bug?

Before loop: sub $8, $8, 4

Some Intel x86 (IA-32) History

Year	CPU	Comment
1978	8086	16-bit with 8-bit bus from 8080; selected for IBM PC
1980	8087	Floating Point Unit
1982	80286	24-bit addresses, memory-map, protection
1985	80386	32-bit registers, flat memory addressing, paging
1989	80486	Pipelining
1992	Pentium	Superscalar
1995	Pentium Pro	Out-of-order execution, 1997 MMX
1999	P-III	SSE – streaming SIMD
2000	**AMD** Athlon	AMD64 or x86-64 64-bit extensions
2000+	P4, …, Haswell	SSE++, virtualization, security, transactions, etc.

Intel 386 Registers & Memory

- Registers
  - 8 32b registers (but backward 16b & 8b: EAX, AX, AH, AL)
  - 4 special registers: stack (ESP) & frame (EBP)
  - Condition codes: overflow, sign, zero, parity, carry
  - Floating point uses 8-element stack
- Memory
  - Flat 32b or segmented (rarely used)
  - Effective address =
    (base_reg + (index_reg x scaling_factor) + displacement)

Intel 386 ISA

- Two register instructions: src1/dst, src2

  reg/reg, reg/immed, reg/mem, mem/reg, mem/imm

- Examples

  mov EAX, 23 # 32b 2's C imm 23 in EAX

  neg [EAX+4] # M[EAX+4] = -M[EAX+4]

  faddp ST(7), ST # ST = ST + ST(7)

  jle label # PC = label if sign or zero flag set

Intel 386 ISA cont’d

- Decoding nightmare
  - Instructions 1 to 17 bytes
  - Optional prefixes, postfixes alter semantics
    - AMD64 64-bit extension: prefix byte
  - Crazy “formats”
    - E.g. register specifiers move around
  - But key 32b 386 instructions not terrible
  - Yet entire ISA has to correctly implemented

Current Approach

- Current technique used by Intel and AMD
  - Decode logic translates to RISC uops
  - Execution units run RISC uops
  - Backward compatible
  - Very complex decoder
  - Execution unit has simpler (manageable) control logic, data paths
- We use MIPS to keep it simple and clean
- Learn x86 later (if necessary)

Complex Instructions

- More powerful instructions not faster
- E.g. string copy
  - Option 1: move with repeat prefix for memory-to-memory move
    - Special-purpose
  - Option 2: use loads/stores to/from registers
    - Generic instructions
- Option 2 can be faster on same machine!

  (but which code is denser?)

Aside on "Endian"

- Big endian: MSB at address xxxxxx00
  - E.g. IBM, SPARC
- Little endian: MSB at address xxxxxx11
  - E.g. Intel x86
- Mode selectable
  - E.g. PowerPC, MIPS
- Causes headaches for
  - Ugly pointer arithmetic
  - Multibyte datatype transfers from one machine to another

Summary – x86

- Not regular
  - Instructions 1-17B in length, optional prefix bytes
- Not simple
  - Prefixes, many addressing modes, complex semantics
- High performance still possible
  - Requires designer cleverness
  - Translate to simple, easy to pipeline operations
  - Much more in ECE 752