KIẾN TRÚC MÁY TÍNH

ET4040

TS. Nguyễn Đức Minh

[Adapted from Computer Organization and Design, 4th Edition, Patterson & Hennessy, © 2008, MK]
[Adapted from Computer Architecture lecture slides, Mary Jane Irwin, © 2008, PennState University]

Tổ chức lớp

Số tín chỉ	3 (3-1-0-6)
Giảng viên	TS. Nguyễn Đức Minh
Văn phòng	611 TV TQB
Email	minh.nguyenduc1@hust.edu.vn
Website	http://ca.edabk.org
Sách	*Computer Org and Design*, 4th Ed., Patterson &Hennessy, ©2007
Thí nghiệm	3 bài
Bài tập	Theo chương, đề bài xem trên trang web

Điểm số

Điều kiện thi **Lab, Bài tập chương**

Bài thi giữa kỳ **30%**

- Bài tập lớn 20%
- Tiến trình 10%

Tích lũy, trừ qua trả lời câu hỏi trên lớp và đóng góp tổ chức lớp

Bài thi cuối kỳ **70%**

Lịch học

❖ Thời gian, địa điểm:

- ❑ Lý thuyết: 11 buổi x 135 phút / 1 buổi
- ❑ Bài tập lớn: 4 buổi x 135 phút / 1 buổi
- ❑ Thay đổi lịch (nghỉ, học bù) sẽ được thông báo trên website trước 2 ngày

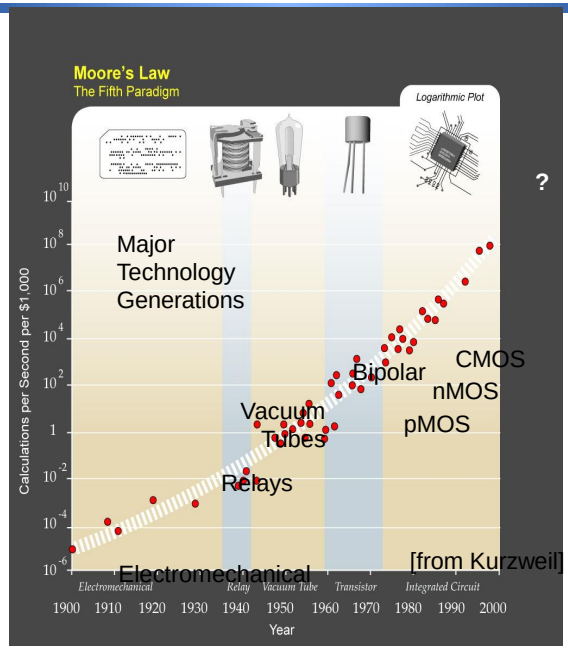
❖ Đăng ký học: ca.edabk.org

Outline

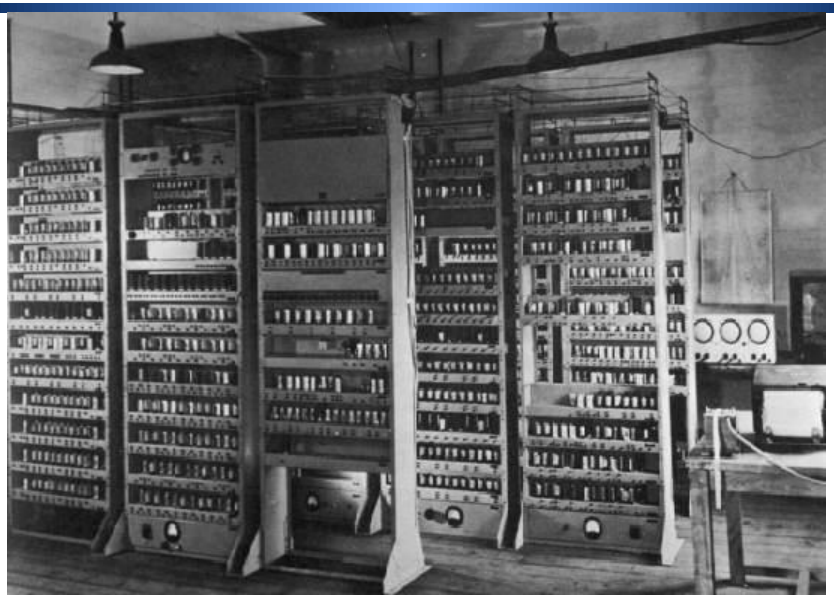
CHƯƠNG 1. GIỚI THIỆU CHUNG VỀ HỆ THỐNG MÁY TÍNH (3 LT+1 BT) (4LT)

1. **Xu hướng công nghệ**
2. **Ứng dụng**
3. **Phân loại**
4. Cấu trúc một hệ thống máy tính-Mô hình von Neumann, mô hình Havard
5. Khái niệm về hiệu năng hệ thống máy tính (Định nghĩa, Công thức tính, Các yếu tố ảnh hưởng; Chương trình kiểm chuẩn hiệu năng)

Phát triển của công nghệ thông tin



Bắt đầu

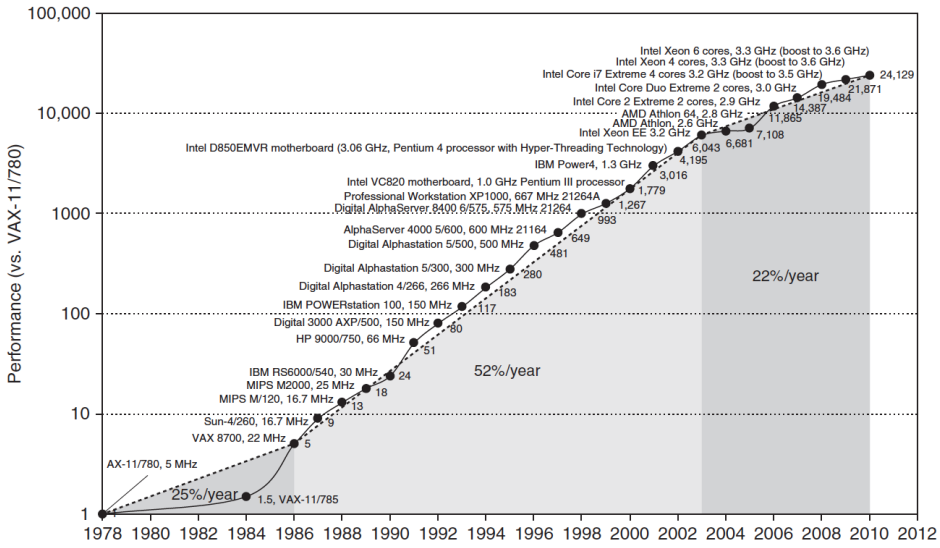


EDSAC, University of Cambridge, UK, 1949

Ngày nay

Máy tính có mặt khắp mọi nơi

Tăng trưởng hiệu năng máy tính



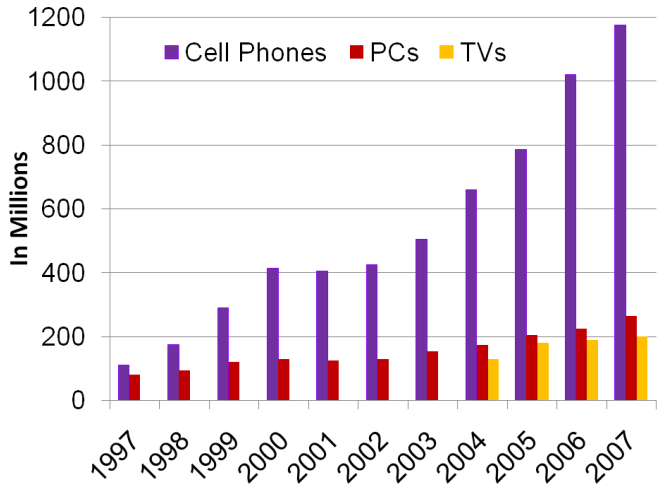
Ứng dụng

- Phương tiện giao thông
  - Khi máy tính trở nên rẻ hơn, nhỏ hơn và có hiệu suất cao hơn, nó được sử dụng trong ô tô, xe máy để tăng hiệu suất sử dụng nhiêu liệu, giảm ô nhiễm, tăng độ an toàn.
- Điện thoại di động, thiết bị viễn thông
  - Giúp con người giao tiếp dù ở bất kỳ đâu.
- Bản đồ gen
  - Máy tính trở nên rẻ và mạnh hơn 10-100 lần so với cách đây 10 năm cho phép trang bị các máy tính để phân tích và ánh xạ bản đồ gen người.
- WWW
  - Nhờ sự phổ biến của máy tính, các thiết bị mạng, Internet trở thành môi trường làm thế giới tràn ngập thông tin (thế giới thông tin).
- Bộ tìm kiếm
  - Google trở thành 1 động từ và 1 đế chế.

Các loại máy tính

- Máy tính để bàn (*eng*, *Desktop computers*)
  - Một người dùng; Chạy nhiều ứng dụng khác nhau; Đi kèm màn hình, bàn phím và chuột; Yêu cầu giá thành rẻ, hiệu năng cao
- Máy chủ (*eng, Servers*)
  - Nhiều người dùng đồng thời; Chạy các ứng dụng lớn; Truy cập qua mạng, Yêu cầu độ ổn định và an toàn cao.
- Siêu máy tính (*eng, Supercomputers*)
  - Chạy các ứng dụng khoa học và công nghệ cao cấp; Gồm hàng trăm/nghìn bộ xử lý, bộ nhớ và bộ lưu trữ dung lượng lớn; Yêu cầu hiệu năng cao và có giá thành cao.
- Máy tính nhúng (eng, *Embedded computers (processors)*)
  - Máy tính nằm bên trong một thiết bị khác, chạy 1 ứng dụng xác định trước.

Tăng trưởng doanh số điện thoại di động



Tăng trưởng điện thoại di đông >> Tăng trưởng máy tính để bàn

Đặc điểm của máy tính nhúng

- Ứng dụng trong nhiều lĩnh vực khác nhau
- Yêu cầu hiệu năng rất khác nhau
- Yêu cầu hiệu năng tối thiểu và vừa đủ. Ví dụ?
- Yêu cầu khắt khe về giá thành. Ví dụ?
- Yêu cầu khắt khe về năng lượng tiêu thụ. Ví dụ?
- Ít chấp nhận hỏng hóc. Ví dụ?

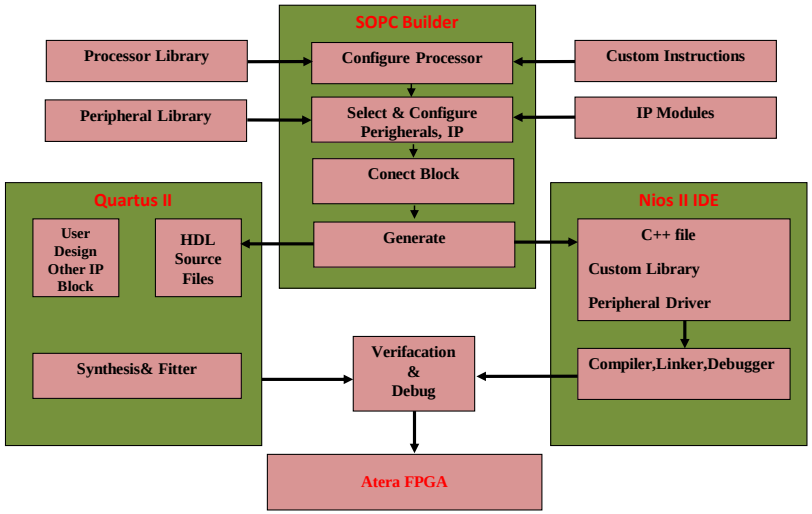
Mục tiêu môn học

Kiến thức về hệ thống máy tính:

- Giao diện giữa phần mềm và phần cứng
- Quá trình biên dịch chương trình phần mềm
- Cấu tạo và hoạt động của phần cứng máy tính
- Phương pháp đánh giá định lượng về hiệu năng máy tính
- Ảnh hưởng của các thành phần lên hiệu năng máy tính

- Kỹ sư phần mềm: tận dụng ưu điểm của phần cứng và lựa chọn phần cứng tối ưu
- Kỹ sư phần cứng: ảnh hưởng của phần cứng lên phần mềm

Mục tiêu môn học



KIẾN TRÚC MÁY TÍNH

Thành phần cơ bản của máy tính

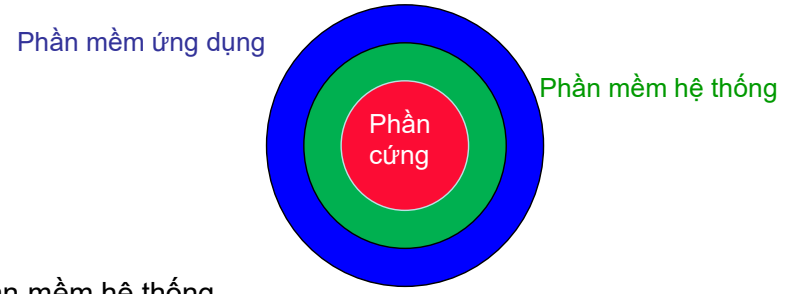
[Adapted from Computer Organization and Design, 4th Edition, Patterson & Hennessy, © 2008, MK]
[Adapted from Computer Architecture lecture slides, Mary Jane Irwin, © 2008, PennState University]

Outline

CHƯƠNG 1. GIỚI THIỆU CHUNG VỀ HỆ THỐNG MÁY TÍNH (3 LT+1 BT) (4LT)

1. Xu hướng công nghệ
2. Ứng dụng
3. Phân loại
4. **Cấu trúc một hệ thống máy tính-Mô hình von Neumann, mô hình Havard**
5. **Khái niệm về hiệu năng hệ thống máy tính (Định nghĩa, Công thức tính, Các yếu tố ảnh hưởng; Chương trình kiểm chuẩn hiệu năng)**

Phần mềm



- Phần mềm hệ thống
  - Hệ điều hành – giám sát, giao tiếp giữa phần cứng và phần mềm ứng dụng (như Linux, MacOS, Windows)
    - Điều khiển các hoạt động vào ra cơ bản
    - Cấp phát bộ nhớ
    - Cung cấp sự chia sẻ có bảo vệ giữa các ứng dụng
  - Bộ biên dịch – chuyển đổi các chương trình ở ngôn ngữ bậc cao (như C, Java) thành các câu lệnh phần cứng có thể thực hiện

Trình biên dịch

- High-level language program (in C)

```
swap (int v[], int k)
(int temp;
        temp = v[k];
        v[k] = v[k+1];
        v[k+1] = temp;
)
```

one-to-many → C compiler

- Assembly language program (for MIPS)

```
swap:  sll    $2, $5, 2
       add    $2, $4, $2
       lw     $15, 0($2)
       lw     $16, 4($2)
       sw     $16, 0($2)
       sw     $15, 4($2)
       jr     $31
```

one-to-one → assembler

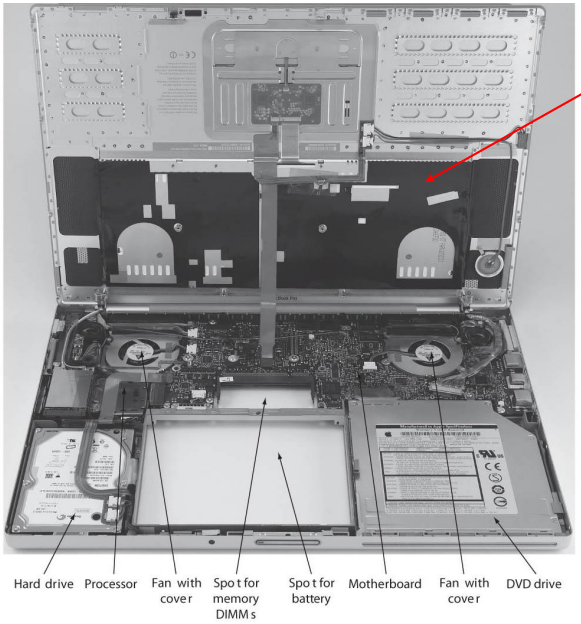
- Machine (object, binary) code (for MIPS)

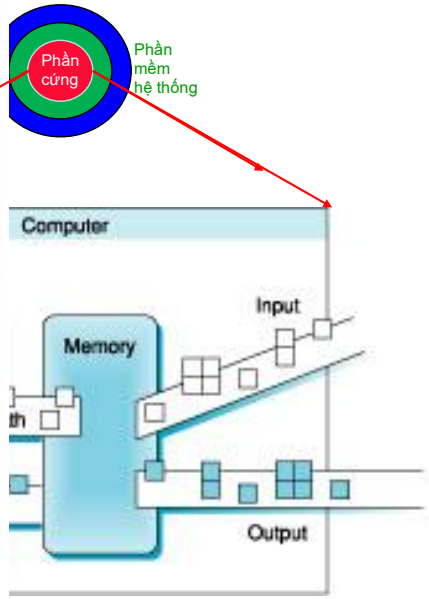
```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
, , ,
```


Ưu điểm của ngôn ngữ bậc cao

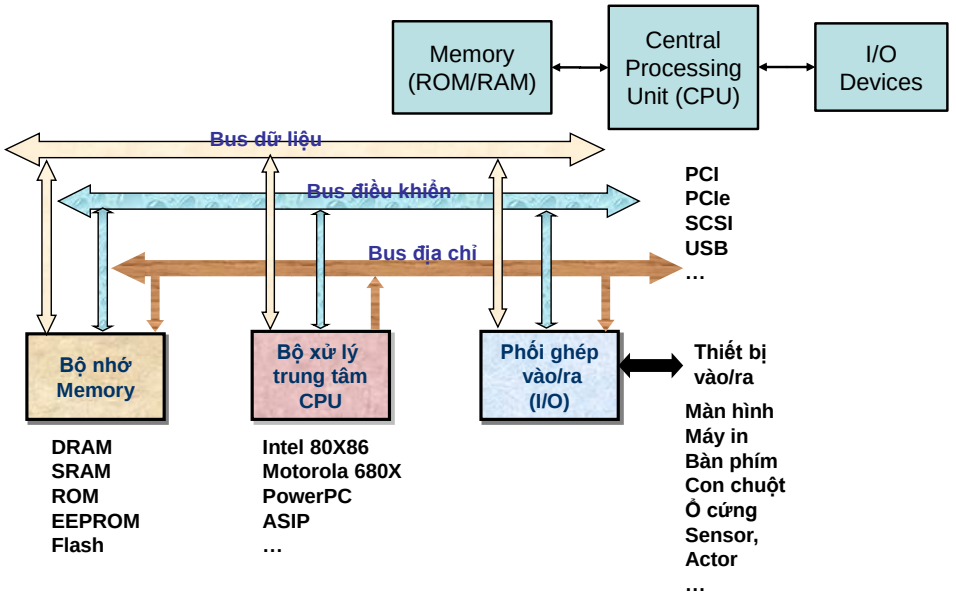
- Ngôn ngữ bậc cao
  - Chương trình được viết ở ngôn ngữ tự nhiên và phù hợp với từng ứng dụng (Ví dụ: Fotran, Lisp, Java)
  - Tăng năng suất lập trình viên – mã chương trình dễ hiểu, dễ gỡ lỗi, dễ kiểm tra
  - Tăng khả năng bảo trì chương trình
  - Chương trình độc lập với phần cứng sẽ thực hiện chương trình
  - Chương trình được tối ưu hóa cho từng loại phần cứng nhờ các thuật toán tối ưu trong trình biên dịch
- Ít chương trình còn được phát triển bằng hợp ngữ

Phần cứng

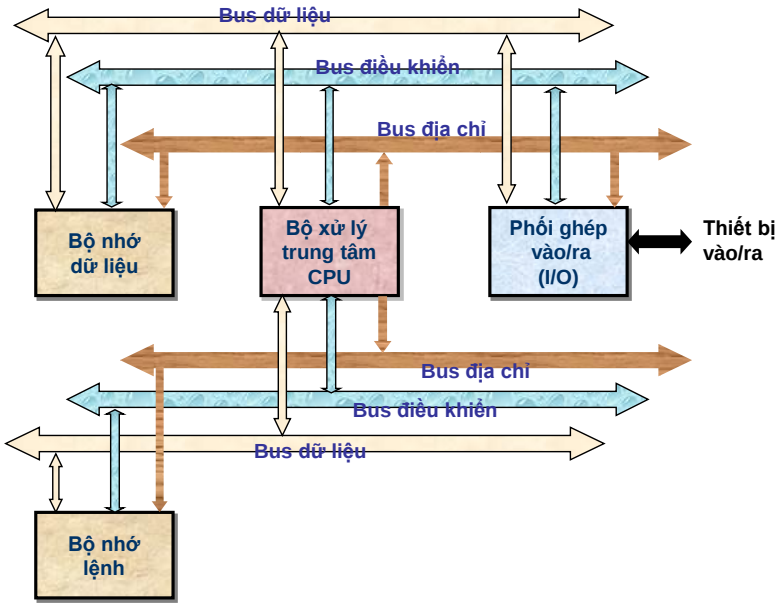




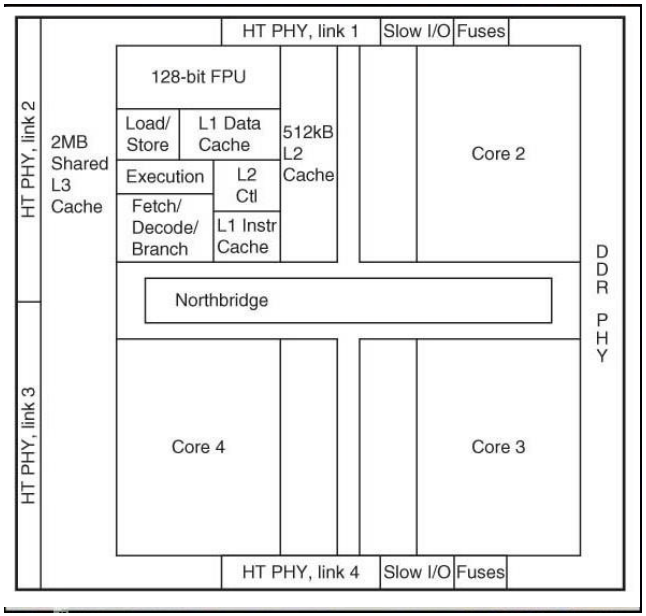
Kiến trúc vonNeumann



Kiến trúc Havard



Bộ xử lý đa nhân AMD's Barcelona



- 4 nhân, lệnh không theo thứ tự
- Đồng hồ: 1,9 GHz
- Công nghệ 65nm
- 3 mức bộ đệm (L1, L2, L3)
- Tích hợp bộ điều khiển cầu bắc

Kiến trúc tập lệnh (ISA)

- Kiến trúc tập lệnh (*eng, Instruction Set Architecture - ISA*), hay kiến trúc: là giao diện trừu tượng giữa phần cứng và các phần mềm ở lớp thấp nhất, bao gồm tất cả các thông tin cần thiết để viết chương trình ở ngôn ngữ máy như: các chỉ thị máy, thanh ghi, bản đồ bộ nhớ, phương pháp vào ra, …

- Giao diện nhị phân ứng dụng (*eng, Application Binary Interface – ABI*) bao gồm các chỉ thị máy cơ bản người dùng có thể sử dụng và các hàm hệ thống cấp thấp do hệ điều hành cung cấp.

Kết luận:

- Máy tính gồm các lớp phân cấp theo mức độ trừu tượng.
- Kiến trúc tập lệnh là một lớp then chốt trong hệ thống máy tính.
- Các triển khai phần cứng khác nhau tuân theo cùng chuẩn về kiến trúc tập lệnh có thể thực hiện cùng một phần mềm giống nhau.

Đánh giá và so sánh các máy tính

- Quyết định mua máy tính
  - Trong số các máy tính, máy nào có
    - hiệu năng tốt nhất?
    - giá thành rẻ nhất?
    - tỉ lệ giá thành/hiệu năng tốt nhất?
- Lựa chọn thiết kế máy tính
  - Trong các lựa chọn thiết kế, thiết kế nào
    - cho cải tiến tốt nhất về hiệu năng?
    - giá thành thấp nhất?
    - tỉ lệ giá thành/hiệu năngtốt nhất?
- Yêu cầu:
  - Căn cứ để so sánh
  - Thông số đánh giá
- Mục tiêu: nắm rõ sự
  - ảnh hưởng của các nhân tố trong kiến trúc máy tính tới hiệu năng toàn hệ thống;
  - vai trò quan trọng tương đối và giá thành của các nhân tố đó,

Hiệu năng (*eng. Performance*)

- Thời gian đáp ứng (thời gian thực thi) – là khoảng thời gian giữa thời điểm bắt đầu thực hiên và thời điểm hoàn thành một nhiệm vụ
  - Quan trọng đối với 1 người sử dụng yêu cầu hệ thống thức hiện 1 nhiệm vụ
- Thông lượng (dải thông) – là tổng số nhiệm vụ có thể được hoàn thành trong 1 khoảng thời gian
  - Quan trọng đối với người điều hành trung tâm dự liệu
- Cần các hệ đo lường khác nhau cho hiệu năng của máy tính cũng như cần 1 tập hợp các ứng dụng khác nhau để kiểm chuẩn các máy tính nhúng, máy tính để bàn (thường chú trọng đến thời gian đáp ứng) và các máy chủ (thường chú trọng đến thông lượng

Ví dụ 1.1 – Cải tiến hiệu năng

- Ảnh hưởng của bộ xử lý lên thời gian đáp ứng và thông lượng
- Nếu ta thay đổi cấu trúc máy tính như sau thì thời gian đáp ứng và thông lượng của máy tính thay đổi thế nào?
  - Thay thế bộ xử lý bằng bộ xử lý nhanh hơn
  - Bổ xung 1 bộ xử lý để thực hiện các nhiệm vụ tách biệt (như trong hệ thống tìm kiếm WWW)

Định nghĩa hiệu năng (tốc độ)

❖ Hiệu năng (tốc độ) của máy tính X:

$$\text{Performance}_x = \frac{1}{\text{Execution Time}_x}$$

❖ Máy tính X nhanh hơn máy tính Y, *n* lần:

$$\frac{\text{Performance}_x}{\text{Performance}_y} = \frac{\text{ExecutionTime}_y}{\text{ExecutionTime}_x} = n$$

❖ Để tối đa hóa hiệu năng, cần tối thiểu hóa thời gian thực hiện

❖ Giảm thời gian đáp ứng thường sẽ tăng thông lượng

Ví dụ 1.2 – So sánh hiệu năng

❖ Nếu máy tính A thực hiện 1 chương trình mất 10s và máy tính B chạy cũng chương trình đó mất 15s, máy tính A nhanh hơn máy tính B bao nhiêu lần?

Ví dụ 1.2 – So sánh hiệu năng

❖ Nếu máy tính A thực hiện 1 chương trình mất 10s và máy tính B chạy cũng chương trình đó mất 15s, máy tính A nhanh hơn máy tính B bao nhiêu lần?

❖ Tỉ lệ so sánh hiệu năng của A và B là:

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{ExecutionTime}_B}{\text{ExecutionTime}_A} = \frac{15}{10} = 1{,}5$$

➔ A nhanh hơn B 1,5 lần

Đo hiệu năng – Đo thời gian thực hiện

Có 3 loại thời gian dùng để tính hiệu năng

❖ Thời gian đáp ứng (thời gian đồng hồ, thời gian đã trôi qua):
- ❑ Tổng thời gian hoàn thành 1 nhiệm vụ
- ❑ Bao gồm: thời gian truy cập đĩa, bộ nhớ, thời gian vào ra, thời gian cho hệ điều hành

❖ Thời gian bộ xử lý (CPU time)
- ❑ Thời gian CPU người dùng
- ❑ Thời gian CPU hệ thống

Các yếu tố tính hiệu năng

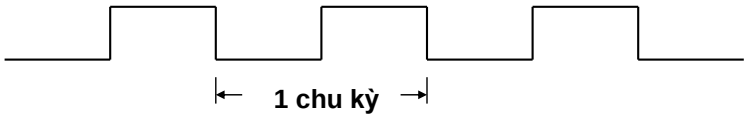
- Thời gian CPU – thời gian bộ xử lý dùng để thực hiện 1 nhiệm vụ
  - Không bao gồm thời gian chờ vào/ra hay thời gian thực hiện các chương trình khác
  - Thời gian CPU cho 1 chương trình, T_{cpu} được tính từ số chu kỳ đồng hồ CPU thực hiện chương trình P và thời gian 1 chu kỳ đồng hồ:

$$T_{cpu} = C \times T_c \text{ or } T_{cpu} = C / f_c$$

- Hiệu năng có thể cải thiện bằng cách giảm số chu kỳ 1 xung đồng hồ hoặc giảm số chu kỳ cần thiết để thực hiện chương trình

Xung nhịp đồng hồ

- CPU hoạt động đồng bộ theo đồng hồ



đồng hồ T_c

10 nsec clock cycle => 100 MHz clock rate

5 nsec clock cycle => 200 MHz clock rate

2 nsec clock cycle => 500 MHz clock rate

1 nsec (10^{-9}) clock cycle => 1 GHz (10^9) clock rate

500 psec clock cycle => 2 GHz clock rate

Ví dụ 1.3 – Cải thiện hiệu năng

- Máy tính A với xung đồng hồ 2GHz thực hiện 1 chương trình hết 10 giây. Để thực hiện chương trình đó trong 6 giây bằng máy tính B, ta cần tăng tốc độ xung đồng hồ của máy B. Tuy nhiên, tăng tốc độ xung đồng hồ cũng làm tăng số chu kỳ cần thiết lên 1,2 lần. Xác định tốc độ xung đồng hồ máy tính B.

Ví dụ 1.3 – Cải thiện hiệu năng

- Công thức để tính thời gian CPU, khi máy tính A thực hiện chương trình:

$$T_{cpu,A} = \frac{C_A}{f_{c,A}}$$

- Số chu kỳ máy tính A dùng để thực hiện chương trình:

$$C_A = T_{cpu,A} \times f_{c,A} = 10 \times 2 \times 10^9 = 20 \times 10^9 \text{ cycles}$$

- Số chu kỳ máy tính B dùng để thực hiện chương trình:

$$C_B = 1{,}2 \times C_A$$

- Tốc độ đồng hồ của máy tính B:

$$f_{c,B} = \frac{C_B}{T_{cpu,B}} = \frac{1{,}2 \times 20 \times 10^9}{6} = 4GHz$$

Số xung đồng hồ

- Số xung đồng hồ thực hiện 1 chương trình:

- Trong đó: $C = I \times CPI$
  - I là số chỉ thị máy cần thực hiện trong chương trình
  - CPI (*eng. Clock cycles per Instruction*) là số xung đồng hồ trung bình cần để thực thi 1 chỉ thị máy,
- CPI có thể dùng để so sánh các máy tính khác nhau cùng triển khai 1 kiến trúc tập lệnh.
- Ví dụ: có 3 loại lệnh A, B, C khác nhau trong 1 kiến trúc tập lệnh. Mỗi lệnh trong từng loại có CPI tương ứng:

	CPI for this instruction class		
	A	B	C
CPI	1	2	3

Ví dụ 1.4 – So sánh dựa trên CPI

- Máy tính A và B cùng triển khai 1 kiến trúc tập lệnh. Máy A có chu kỳ đồng hồ là 250ps, và CPI hiệu dụng cho 1 chương trình P là 2,0. Máy B có chu kỳ đồng hồ là 500ps, và CPI hiệu dụng cho cùng 1 chương trình P là 1,2. Máy tính nào nhanh hơn và nhanh hơn bao nhiêu?

Ví dụ 1.4 – So sánh dựa trên CPI

- Khi 2 máy tính A, B cùng thực hiện 1 chương trình, chúng cùng thực hiện I chỉ thị. Do đó:

$$T_{cpu,A} = I \times CPI_A \times T_{c,A} = I \times 2{,}0 \times 250ps = 500 \times I$$
$$T_{cpu,B} = I \times CPI_B \times T_{c,B} = I \times 1{,}2 \times 500ps = 600 \times I$$

- Vì vậy, máy A nhanh hơn máy B và:

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{T_{cpu,B}}{T_{cpu,B}} = \frac{600 \times I}{500 \times I} = 1{,}2$$

CPI hiệu dụng (trung bình)

- CPI hiệu dụng được tính bằng cách xét tất cả các lớp chỉ thị có trong chương trình và lấy trung bình với trọng số là tỉ lệ xuất hiện của lớp chỉ thị trong chương trình

$$CPI = \sum_{i=1}^{n} (CPI_i \times IC_i)$$

- Trong đó:
  - IC_i là tỉ lệ (%) số chỉ thị thuộc lớp i được thực thi
  - CPI_i là số chu kỳ (trung bình) cần để thực hiện 1 chỉ thị thuộc thuộc lớp i
  - N là số lớp chỉ thị
- CPI hiệu dụng phụ thuộc vào tỉ lệ chỉ thị trong một chương trình (tần suất động của các chỉ thị trong 1 hoặc nhiều chương trình)

Công thức hiệu năng

- Công thức hiệu năng cơ bản:

$$T_{cpu} = I \times CPI \times T_c \qquad T_{cpu} = \frac{I \times CPI}{f_c}$$

- Công thức trên phân tách 3 yếu tố ảnh hưởng đến hiệu năng máy tính
- Cho phép đo CPU được đo bằng cách chạy chương trình:
  - Tốc độ đồng hồ được cho trước
  - Số chỉ thị I được đo bằng cách dùng công cụ profilers/mô phỏng sự thực hiện chương trình mà không cần triển khai phần cứng
  - CPI phụ thuộc vào loại chỉ thị, triển khai phần cứng chi tiết
- Cho phép so sánh 2 triển khai phần cứng hoặc đánh giá lựa chọn giữa 2 thiết kế

Các yếu tố quyết định hiệu năng CPU

$$T_{cpu} = I \times CPI \times T_c$$

	I	CPI	T_c
Algorithm			
Programming language			
Compiler			
ISA			
Core organization			
Technology			

Các yếu tố quyết định hiệu năng CPU

$$T_{cpu} = I \times CPI \times T_c$$

	I	CPI	T_c
Algorithm	X	X	
Programming language	X	X	
Compiler	X	X	
ISA	X	X	X
Core organization		X	X
Technology			X

Ví dụ 1.5 – So sánh đoạn mã chương trình

- Người thiết kế một máy tính triển khai kiến trúc tập lệnh gồm 3 loại chỉ thị A, B, C được CPI như sau:

	A	B	C
CPI	1	2	3

- Với 1 câu lệnh ở ngôn ngữ bậc cao, người viết trình biên dịch có thể lựa chọn 2 đoạn chỉ thị máy gồm có tần suất các loại chỉ thị như sau:

Đoạn mã	**A**	**B**	**C**
1	2	1	2
2	4	1	1

- Đoạn mã nào gồm nhiều chỉ thị hơn? Đoạn mã nào nhanh hơn? Tính CPI của từng đoạn mã.

Ví dụ 1.5 – So sánh đoạn mã chương trình

Đoạn mã	A	B	C
1	2	1	2
2	4	1	1

	A	B	C
CPI	1	2	3

- Đoạn mã 1 dùng 5 chỉ thị, đoạn mã 2 dùng 6 chỉ thị
- Số xung đồng hồ để thực hiện mỗi đoạn mã được tính như sau:

$$C_1 = \sum_{i=1}^{3}(CPI_i \times I_{1,i}) = (1\times 2 + 2\times 1 + 3\times 2) = 10$$

$$C_2 = \sum_{i=1}^{3}(CPI_i \times I_{2,i}) = (1\times 4 + 2\times 1 + 3\times 1) = 9$$

  - Trong đó $I_{1,i}$, $I_{2,i}$ là số lượng chỉ thị loại i trong đoạn mã 1 và 2 tương ứng
- Như vậy đoạn mã 1 chậm hơn đoạn mã 2, mặc dù dùng ít chỉ thị hơn

Ví dụ 1.6 – Cải tiến hiệu năng

- Cho một máy tính thực hiện 1 chương trình gồm 4 loại chỉ thị máy có các thông số về tần suất và CPI như sau:

Op	Tần suất (IC_i)	CPI_i	IC_i x CPI_i
ALU	50%	1	
Load	20%	5	
Store	10%	3	
Branch	20%	2	
			Σ =

- Nếu ta có bộ đệm dữ liệu làm giảm thời gian nạp (Load) xuống 2 chu kỳ, máy tính sẽ nhanh lên bao nhiêu lần?
- Nếu ta có khối dự báo rẽ nhánh cho phép tiết kiệm 1 chu kỳ khi rẽ nhánh, hiệu năng sẽ thế nào?
- Nếu ta có 2 khối ALU thực hiện 2 chỉ thị ALU đồng thời?

Ví dụ 1.6 – Cải tiến hiệu năng

Op	Tần suất (IC_i)	CPI_i	IC_i x CPI_i			
ALU	50%	1	0,5	0,5	0,5	0,25
Load	20%	5	1,0	0,4	1,0	1,0
Store	10%	3	0,3	0,3	0,3	0,3
Branch	20%	2	0,4	0,4	0,2	0,4
		Σ =	2,2	1,6	2,0	1,95

- Nếu ta có bộ đệm dữ liệu làm giảm thời gian nạp (Load) xuống 2 chu kỳ, máy tính sẽ nhanh lên bao nhiêu lần?

  T_{cpu} = 1,6 x I x T_c vì vậy 2,2/1,6 means 37,5% faster
- Nếu ta có khối dự báo rẽ nhánh cho phép tiết kiệm 1 chu kỳ khi rẽ nhánh, hiệu năng sẽ thế nào?

  T_{cpu} = 2,0 x I x T_c vì vậy 2,2/2,0 means 10% faster
- Nếu ta có 2 khối ALU thực hiện 2 chỉ thị ALU đồng thời?

  T_{cpu} = 1,95 x I x T_c vì vậy 2,2/1,95 means 12,8% faster

Chương trình đo kiểm chuẩn

- Benchmarks – là tập hợp các chương trình tạo nên 1 "workload" được chọn để đo hiệu năng
- SPEC (System Performance Evaluation Cooperative) tạo ra 1 tập các benchmark chuẩn bắt đầu từ SPEC89. Chuẩn mới nhất là SPEC CPU2006 gồm 12 chương trình số nguyên (CINT2006) và 17 chương trình số thực (CFP2006)

  www.spec.org

- Ngoài ra, còn có các tập các workload để đo kiểm năng lượng (SPECpower_ssj2008), mail (SPECmail2008), hay đa phương tiện (mediabench), …

Các chương trình đo kiểm

Integer benchmarks		FP benchmarks	
gzip	compression	wupwise	Quantum chromodynamics
vpr	FPGA place & route	swim	Shallow water model
gcc	GNU C compiler	mgrid	Multigrid solver in 3D fields
mcf	Combinatorial optimization	applu	Parabolic/elliptic pde
crafty	Chess program	mesa	3D graphics library
parser	Word processing program	galgel	Computational fluid dynamics
eon	Computer visualization	art	Image recognition (NN)
perlbmk	perl application	equake	Seismic wave propagation simulation
gap	Group theory interpreter	facerec	Facial image recognition
vortex	Object oriented database	ammp	Computational chemistry
bzip2	compression	lucas	Primality testing
twolf	Circuit place & route	fma3d	Crash simulation fem
		sixtrack	Nuclear physics accel
		apsi	Pollutant distribution

SPEC CINT2006 on Barcelona (T_c = 0,4 x 10^{-9})

Name	Ix10^9	CPI	ExTime	RefTime	SPEC ratio
perl	2 118	0,75	637	9 770	15,3
bzip2	2 389	0,85	817	9 650	11,8
gcc	1 050	1,72	724	8 050	11,1
mcf	336	10,00	1 345	9 120	6,8
go	1 658	1,09	721	10 490	14,6
hmmer	2 783	0,80	890	9 330	10,5
sjeng	2 176	0,96	837	12 100	14,5
libquantum	1 623	1,61	1 047	20 720	19,8
h264avc	3 102	0,80	993	22 130	22,3
omnetpp	587	2,94	690	6 250	9,1
astar	1 082	1,79	773	7 020	9,1
xalancbmk	1 058	2,70	1 143	6 900	6,0
Geometric Mean					**11,7**

So sánh và tổng kết hiệu năng

- Tổng kết hiệu năng cho 1 tập tiêu chuẩn thành một số duy nhất:
  - Thời gian thực hiện được chuẩn hóa thành SPECRatio (tỉ số lớn có nghĩa là nhanh hơn)
  - Lấy trung bình nhân của các SPECRatio:

$$GM = \sqrt[n]{\prod_{i=1}^{n} \text{SPEC ratio}_i}$$

Các thông số hiệu năng khác

- Năng lượng tiêu thụ là một thông số quan trọng, đặc biệt với thị trường hệ nhúng (thời lượng pin là quan trọng với hệ thống)

Target Load %	Performance (ssj_ops)	Average Power (Watts)
100 %	231,86 7	295
90%	211 ,282	286
80%	185,80 3	275
70%	163,42 7	265
60%	140,16 0	256
50%	118,32 4	246
40%	92,03 5	233
30%	70,50 0	222
20%	47,12 6	206
10%	23,06 6	180
0%	0	141
Overall Sum	1,283, 590	2,605
Σ ssj_ops / Σ power =		493

Luật Amdahl

- Khi thực hiện cải tiến một đặc điểm của hệ thống, tác dụng của việc cải tiến bị giới hạn bởi đặc điểm được cải tiến

$$T_{cpu} \text{ sau khi cải tiến} = \frac{T_{cpu} \text{ ảnh hưởng bởi sự cải tiến}}{\textit{Tỉ lệ} \text{ cải tiến}} + T_{cpu} \text{ không ảnh hưởng}$$

- Luật cơ bản để tính toán định lượng sự cải tiến
- Khi cải tiến, cần chú trọng đến các trường hợp thông dụng
- Đặt ra giới hạn số lượng bộ xử lý hoạt động song song

Ví dụ 1.7 - Luật Amdahl

- 1 chương trình thực hiện bởi 1 máy tính trong 100 giây. Trong đó, 80 giây được dùng để thực hiện phép nhân. Vậy cần phải tăng tốc độ phép nhân lên mấy lần để có thể tăng tốc độ thực hiện chương trình lên 5 lần?

$$T_{cpu} \text{ sau khi cải tiến} = \frac{T_{cpu} \text{ ảnh hưởng bởi sự cải tiến}}{\textit{Tỉ lệ} \text{ cải tiến}} + T_{cpu} \text{ không ảnh hưởng}$$

$$20\ giây = \frac{80\ giây}{n} + (100\text{-}80)$$

- Không thể cải tiến để tăng tốc được 5 lần

Kết luận chương

- Hệ thống máy tính được xây dựng **từ phân cấp các lớp trừu tượng**. Các chi tiết triển khai lớp dưới bị che khuất khỏi lớp trên.
- **Kiến trúc tập lệnh** – lớp giao tiếp giữa phần cứng và phần mềm mức thấp – là lớp trừu tượng quan trọng trong hệ thống máy tính.
- Phần cứng máy tính gồm **5 thành phần**: **đường dữ liệu, khối điều khiển, bộ nhớ, khối vào, và khối ra**. 5 thành phần đó kết nối với nhau bằng hệ thống bus theo mô hình **vonNeumann** hoặc mô hình **Havard**.
- Phương pháp đánh giá hiệu năng một hệ thống máy tính là dùng **thời gian thực hiện 1 chương trình**. Thời gian thực hiện chương trình được tính bằng công thức:

$$T_{cpu} = I \times CPI \times T_c$$

Computer Architecture 1

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolić, © 2020, UC Berkeley]

What you need to know about this class

- **Course information**
- Course Website:

https://sites.google.com/site/kimhuetadtvtbk/cac-mon-giang-day/kien-truc-may-tinh

https://cs61c.org/sp20/#lectures

- Textbooks: Average 15 pages of reading/week
  - Patterson & Hennessy, Computer Organization and Design, RISC-V edition
  - Kernighan & Ritchie, The C Programming Language 2nd Edition
  - Barroso & Holzle, The Datacenter as a Computer 3rd

What you need to know about this class

- **Course Grading**
- Homework, Assignments, mini-tests(15%)
- Project, midterm exam (15%)
- Final exam (70%)

- **Extra Score: EPA!**
- Effort
  - Completing all assignments and homework in time
- Participation
  - Asking/Answering Qs in TEAMS/ offline class & making it interactive
- Altruism
  - Helping others in class
  - Writing software, tutorials that help others learn
- EPA! can bump students up to the next grade level

What is expected to this course ?

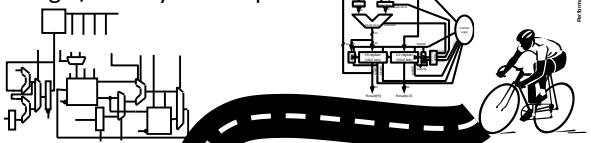
- To explain what's inside the revolutionary machine, unraveling the software below your program and the hardware under the covers of your computer
- To understand the aspects of both the hardware and software that affect program performance
- By the time you complete this course, you will be able to answer the following questions:
  - ✓How are programs written in a high-level language, such as C, Java translated into the language of hardware
  - ✓How does the hardware execute the resulting program?
  - ✓What determines the performance of a program?
  - ✓What techniques can be used by a programmer/hardware designers to improve the performance

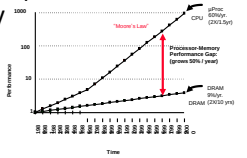
Road map

Chapter 1: Introduction

Chapter 2: ISA MIPS – RISC V

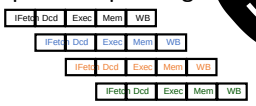
Chapter 3: Single/multicycle Datapath

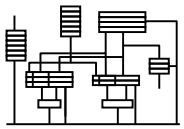




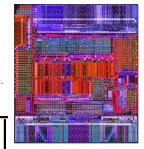
Course information is described in more detail in the syllabus

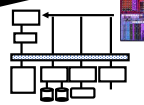
Chapter 4: Pipelining





Chapter 5: Memory Systems



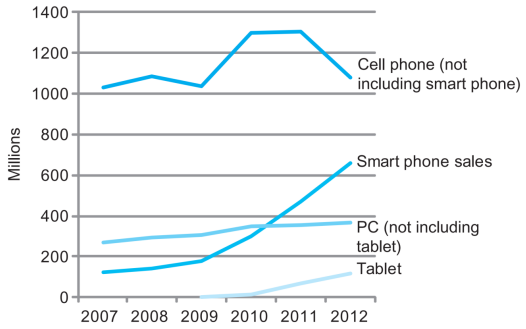


Chapter 6: Parallel processor

Introduction to Computer Abstractions and Technology

- Introduction
- Great Ideas in Computer Architecture

Introduction



Hardware advances have allowed programmers to create wonderfully useful software, which explains why computers are omnipresent
- *Computers in automobiles*
- *Cell phones*
- *Human genome project*
- *World Wide Web*
- *Search engines*

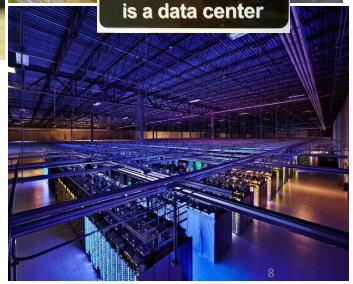
Smart phones represent the recent growth in the cell phone industry, and they passed PCs in 2011.
Tablets are the fastest growing category, nearly doubling between 2011 and 2012.
Recent PCs and traditional cell phone categories are relatively flat or declining

Welcome to the Post-PC Era

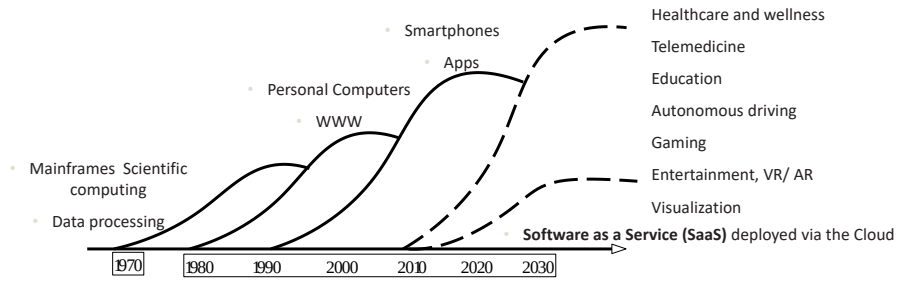
Embedded computes in Network Edge Devices

Replacing the PC is the Personal Mobile Devices (PMD)

Taking over from the conventional server is **Cloud Computing**, which relies upon giant datacenters that are now known as *Warehouse Scale Computers* (WSCs).
Companies like Amazon and Google build these WSCs containing 100,000 servers

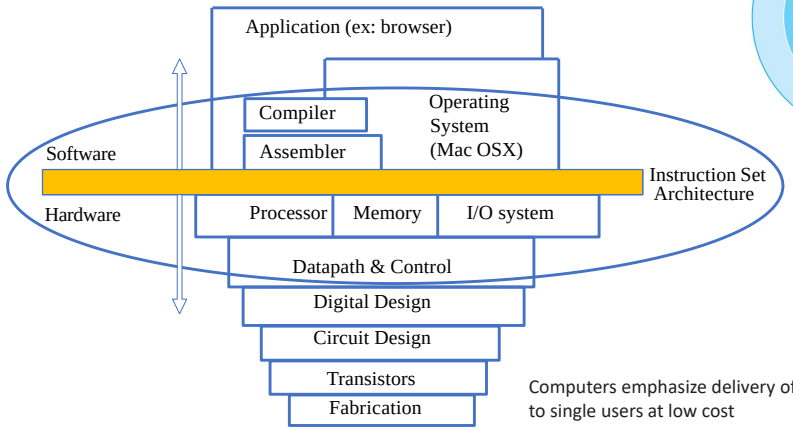


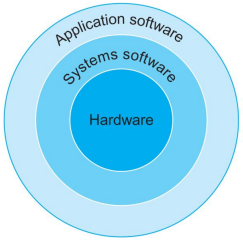
Why is Computer Architecture Exciting Today?



- Number of deployed devices continues to grow
  - Diversification of needs, architectures
  - Machine learning is common for most domains

Thinking about machine structure in Computer Architecture 1

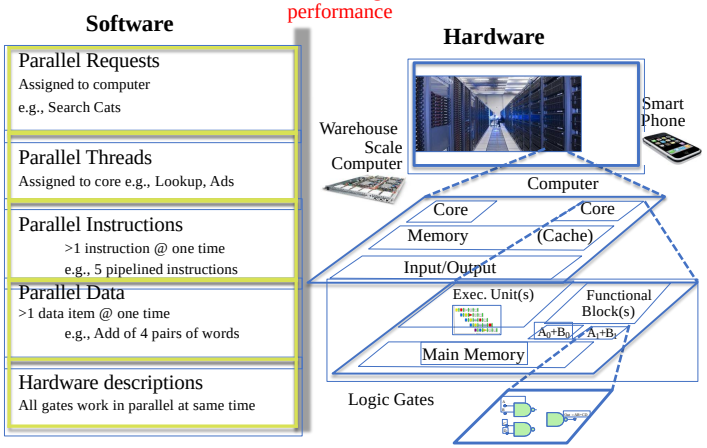




Computers emphasize delivery of good performance to single users at low cost

Thinking about machine structure in Advanced Computer Arch – CA 2

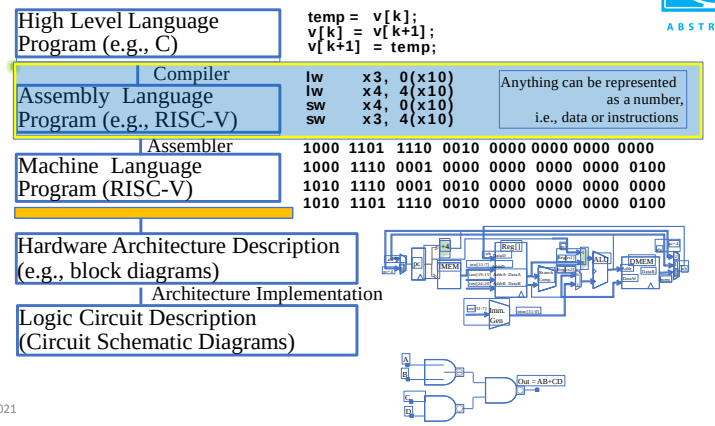
Harness Parallelism & achieve high performance



6 Great Ideas in Computer Architecture

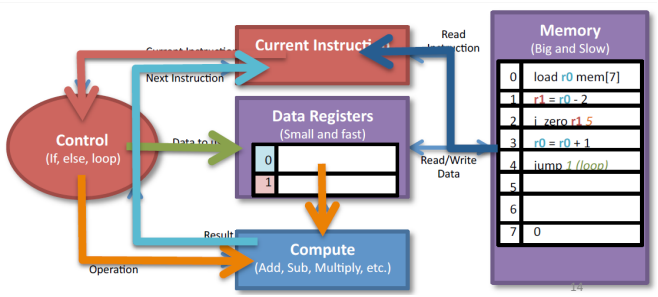
- Abstraction (Layers of Representation/Interpretation)
- Moore's Law
- Principle of Locality/Memory Hierarchy
- Parallelism
- Performance Measurement & Improvement
- Dependability via Redundancy

Great Idea #1: Abstraction (Layers of Representation/Interpretation)

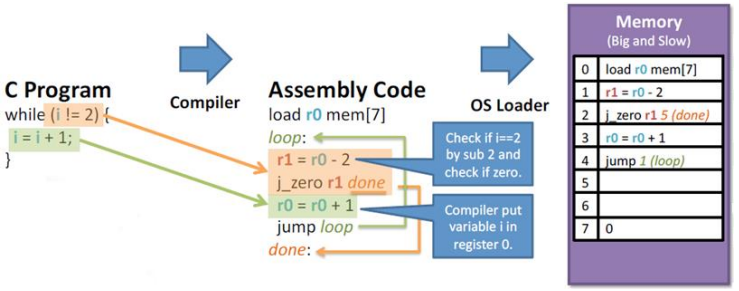


Below your program: Let's walk through the program

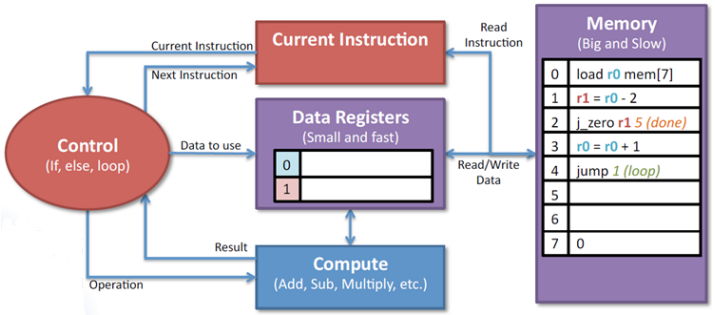
- What will the processor do?
  - **1. Load the instruction**
  - **2. Figure out what operation to do**
  - **3. Figure out what data to use**
  - **4. Do the computation**
  - **5. Figure out next instruction**
- Repeat this over and over and over…



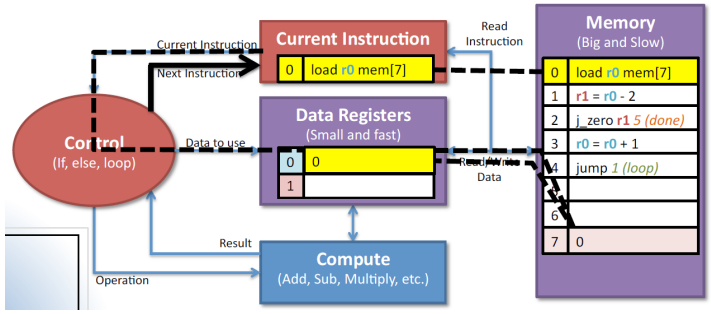
Example



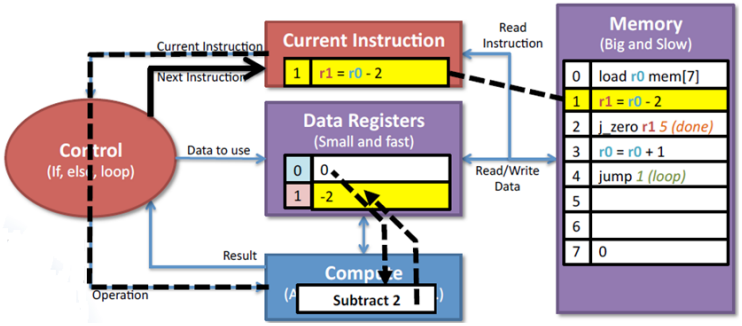
Basic processor: Memory, Control, and Compute



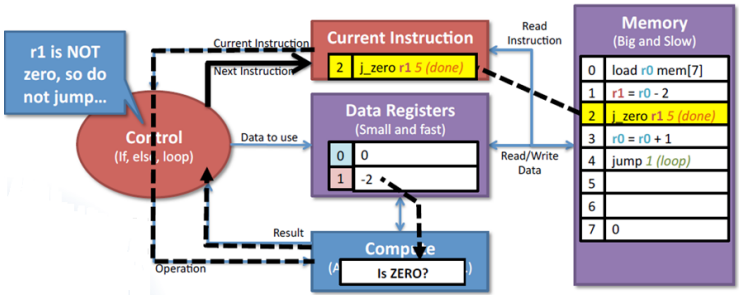
1: Load r0 (i) from memory (location 7)



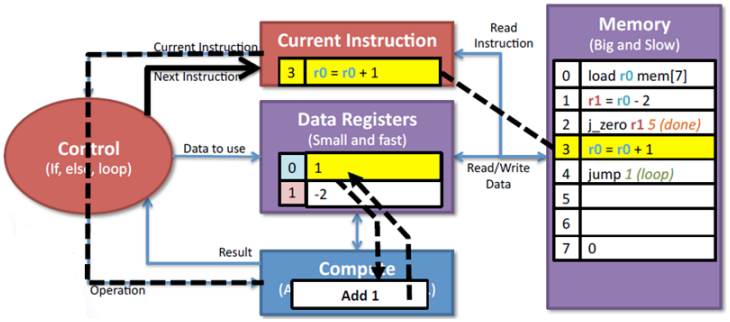
2: Subtract 2 from r0(i) to see if it is 2



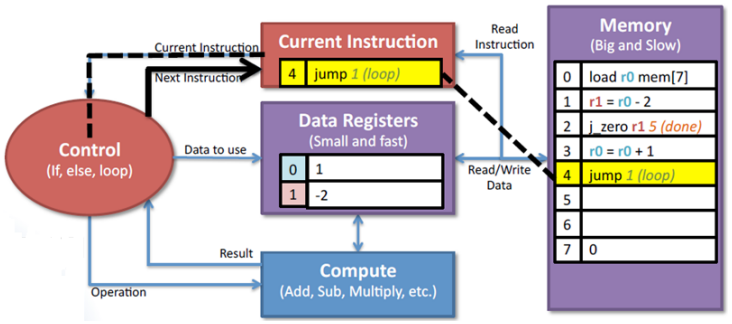
3: Check if r1 is zero 0, and jump to done if it is



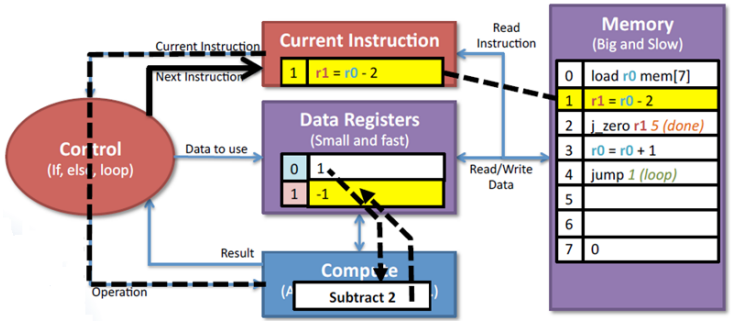
4: Increment r0 (i)



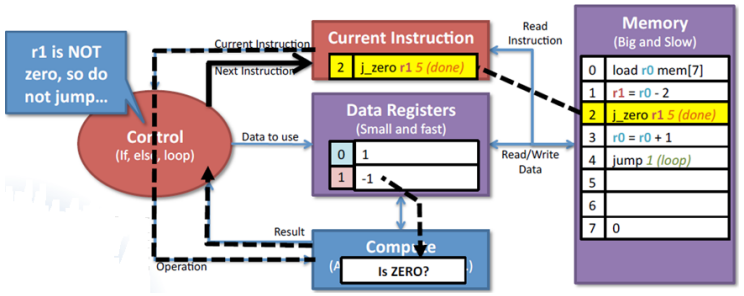
5: Continue the loop



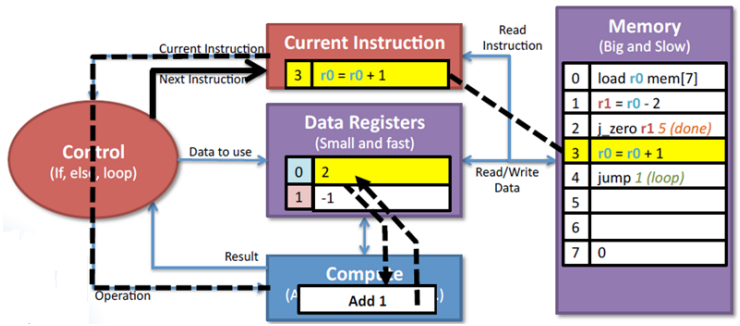
6: Subtract 2 from r0(i) to see if it is 2



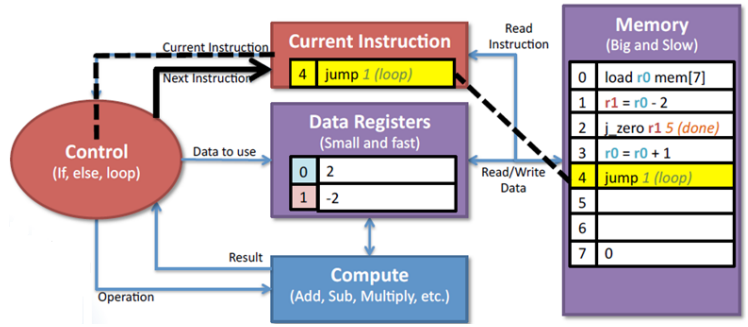
7: Check if r1 is zero 0, and jump to done if it is



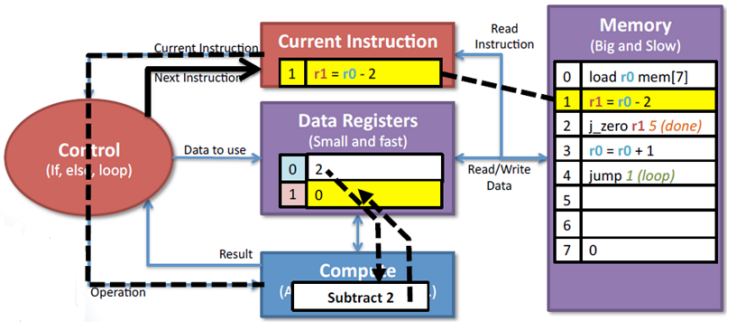
8: Increment r0 (i)



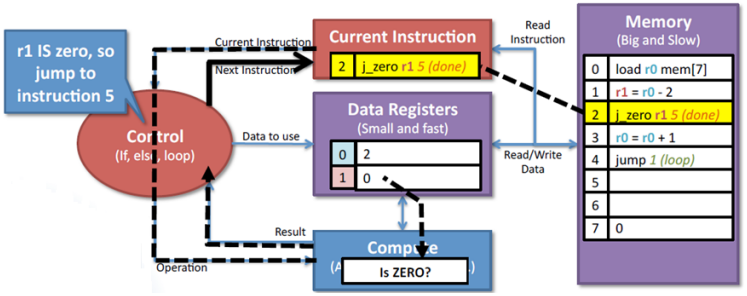
9: Continue the loop



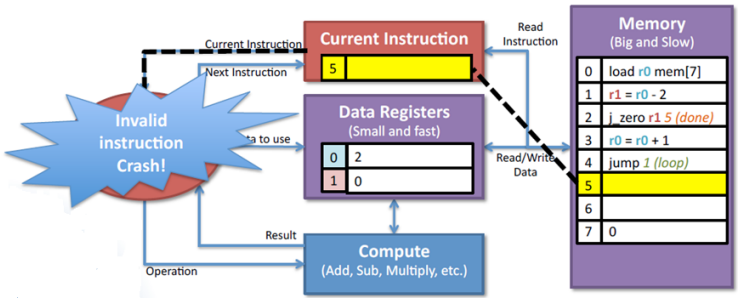
10: Subtract 2 from r0(i) to see if it is 2



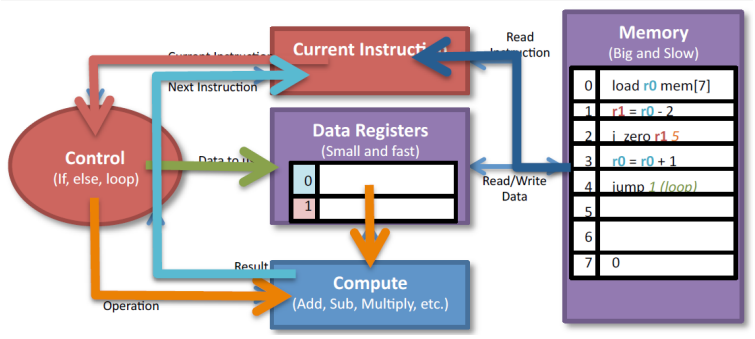
11: Check if r1 is zero 0, and jump to done if it is



12: Crash because the instruction 5 is invalid!



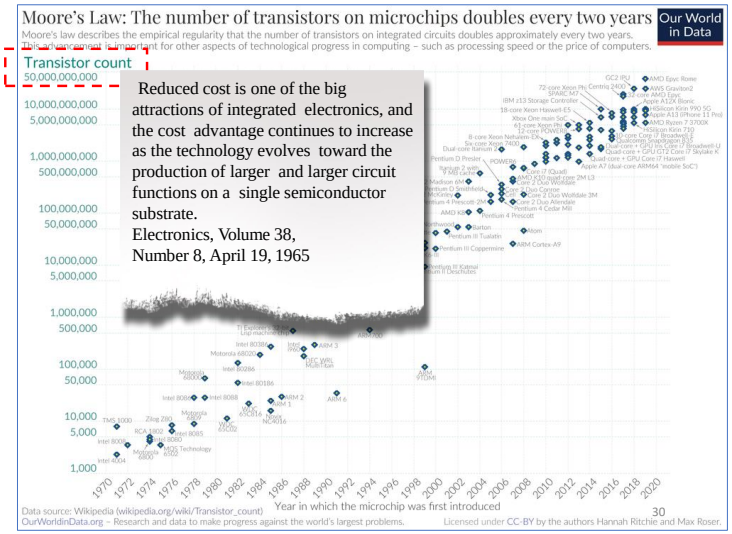
This course is about understanding the **details**, **corner cases**, **performance**, and how this all comes together in a RISC V processor.



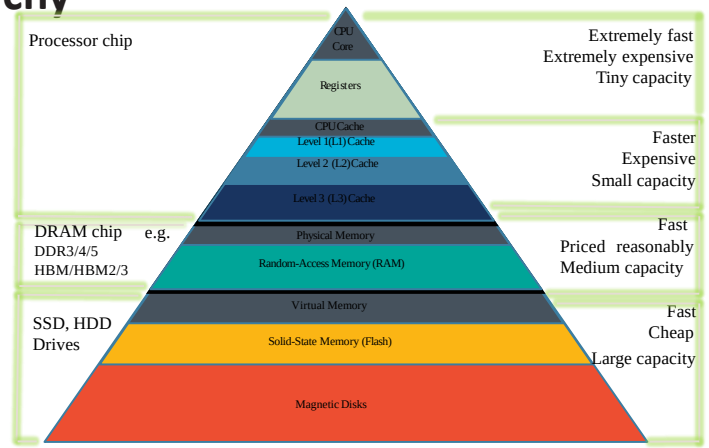
Great Idea #2: Moore's Law

It states that integrated circuit resources double every 18–24 months

Gordon Moore
Intel Cofounder
B.S. Cal 1950!

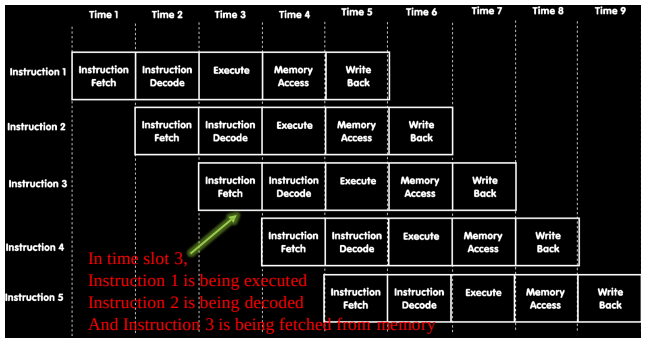


Great Idea #3 : Principle of Locality/Memory Hierarchy



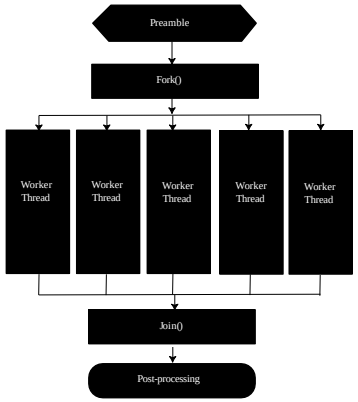
Great Idea #4: Parallelism (1/3)

Performance via Pipelining

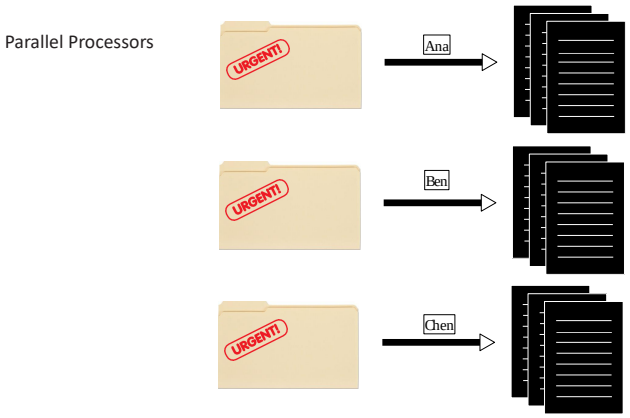


Great Idea #4: Parallelism (2/3)

Parallel Threads



Great Idea #4: Parallelism (3/3)



Caveat! Amdahl's Law

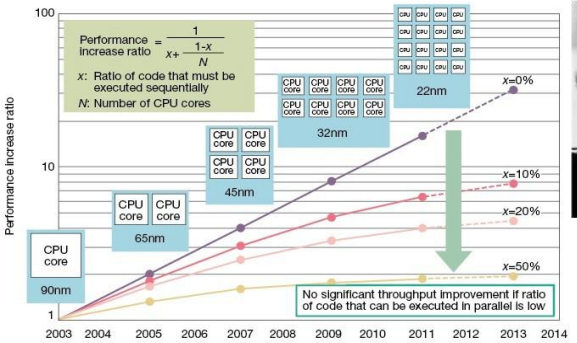


Fig 3 Amdahl's Law an Obstacle to Improved Performance Performance will not rise in the same proportion as the increase in CPU cores. Performance gains are limited by the ratio of software processing that must be executed sequentially. Amdahl's Law is a major obstacle in boosting multicore microprocessor performance. Diagram assumes no overhead in parallel processing. Years shown for design rules based on Intel planned and actual technology. Core count assumed to double for each rule generation.

Gene Amdahl
Computer Pioneer

Great Idea #5: Performance Measurement and Improvement

- How hardware and software affect performance ?

Hardware or software component	How this component affects performance
Algorithm	Determines both the number of source level statements and the number of I/O operations executed
Programming language, compiler, and architecture	Determines the number of computer instructions for each source level statement
Processor and memory system	Determines how fast instructions can be executed
I/O system (hardware and operating system)	Determines how fast I/O operations may be executed

Great Idea #5: Performance Measurement and Improvement

- Matching application to underlying hardware to exploit:
  - Locality
  - Parallelism
  - Special hardware features, like specialized instructions (e.g., matrix manipulation)
- Latency/Throughput
  - How long to set the problem up and complete it (or how many tasks can be completed in given time)
  - How much faster does it execute once it gets going
  - Latency is all about time to finish

Great Idea #6: Dependability via Redundancy

- Redundancy so that a failing piece doesn't make the whole system fail
- Applies to everything from datacenters to storage to memory to instructors
  - Redundant datacenters so that can lose 1 data center, but Internet service stays online
  - Redundant disks so that can lose 1 disk but not lose data (Redundant Arrays of Independent Disks/RAID)
  - Redundant memory bits of so that can lose 1 bit but no data (Error Correcting Code/ECC Memory)

Homework

- Reading Technologies for Building Processors and Memory (Section 1.5 #page 24) and writing the report

Computer Architecture 1

Computer Organization and Design
THE HARDWARE/SOFTWARE INTERFACE

[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolíc, © 2020, UC Berkeley]

Instructions: Language of the Computer

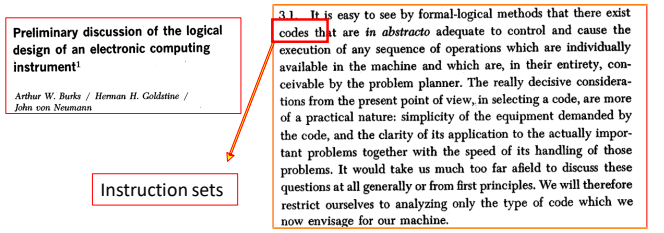
- Introductions
- Operations of the Computer Hardware
- Operands of the Computer Hardware
- Logical Operations
- Instructions for Making Decisions

Introduction

- To demonstrate how easy it is to pick up other instruction sets, we will also take a quick look at two other popular instruction sets.
  - MIPS is an elegant example of the instruction sets designed since the 1980s. In several respects, RISC-V follows a similar design.
  - The Intel x86 originated in the 1970s, but still today powers both the PC and the Cloud of the post-PC era.

Operations of the Computer Hardware

- Instruction set for a particular architecture (e.g. RISC-V) is represented by the Assembly language
- Each line of assembly code represents one instruction for the computer



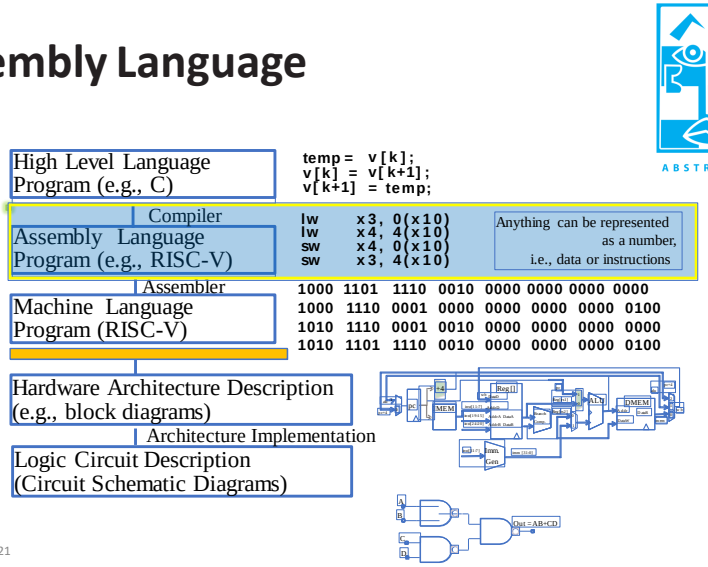
Preliminary discussion of the logical design of an electronic computing instrument[1]

Arthur W. Burks / Herman H. Goldstine / John von Neumann

3.1. It is easy to see by formal-logical methods that there exist codes that are *in abstracto* adequate to control and cause the execution of any sequence of operations which are individually available in the machine and which are, in their entirety, conceivable by the problem planner. The really decisive considerations from the present point of view, in selecting a code, are more of a practical nature: simplicity of the equipment demanded by the code, and the clarity of its application to the actually important problems together with the speed of its handling of those problems. It would take us much too far afield to discuss these questions at all generally or from first principles. We will therefore restrict ourselves to analyzing only the type of code which we now envisage for our machine.

Instruction sets

Assembly Language



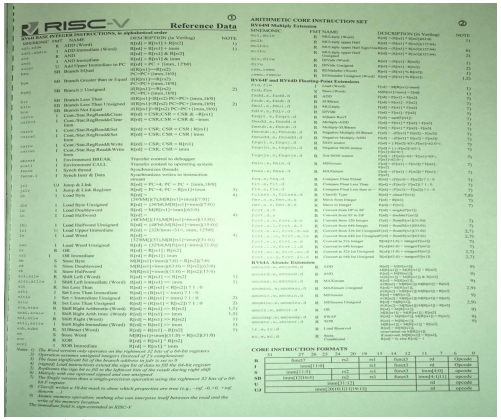
Assembly Language

- Basic job of a CPU: execute lots of *instructions*.
- Instructions are the primitive operations that the CPU may execute.
  - Like a sentence: operations (verbs) applied to operands (objects) processed in sequence …
- Different CPUs implement different sets of instructions. The set of instructions implemented in a particular CPU is an *Instruction Set Architecture* (*ISA*).
  - Examples: ARM (cell phones), Intel x86 (i9, i7, i5, i3), IBM Power, IBM/Motorola PowerPC (old Macs), MIPS, RISC-V, …

RISC-V Architecture

- New open-source, license-free ISA spec
  - Supported by growing shared software ecosystem
  - Appropriate for all levels of computing system, from microcontrollers to supercomputers
  - 32-bit, 64-bit, and 128-bit variants (we're using 32-bit in class, textbook uses 64-bit)
- Why RISC-V instead of Intel 80x86?
  - RISC-V is simple, elegant. Don't want to get bogged down in gritty details.
  - RISC-V has exponential adoption

RISC V – GREEN CARD



https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf

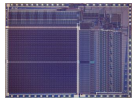
https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf

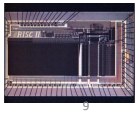
RISC V Origin https://cs61c.org/resources/

- Started in Summer 2010 to support open research and teaching at UC Berkeley
  - Lineage can be traced to RISC-I/II projects (1980s)
  - As the project matured, it migrated to RISC-V foundation (www.riscv.org)
- Many commercial and research projects based on RISC-V, open-source and proprietary
- Widely used in education
- Read more:
  - https://riscv.org/risc-v-history/
  - https://riscv.org/risc-v-genealogy/

RISC-I



RISC-II



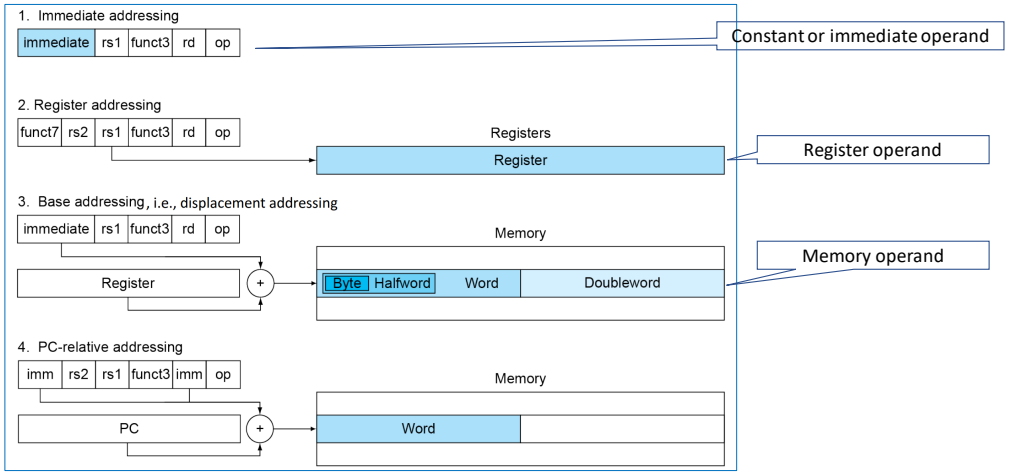
RISC V Instructions

Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

Arithmetic : + , / , * , -
Logical: AND, OR, XOR
Data transfer: Load, store
Shift : >>, <<
Condition branch: if … then
Unconditional branch: jump to loop

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
Data transfer	Load doubleword	`ld x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Doubleword from memory to register
	Store doubleword	`sd x5, 40(x6)`	`Memory[x6 + 40] = x5`	Doubleword from register to memory
	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	`x5 = Memory[x6]`	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	`Memory[x6] = x5; x7 = 0/1`	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate

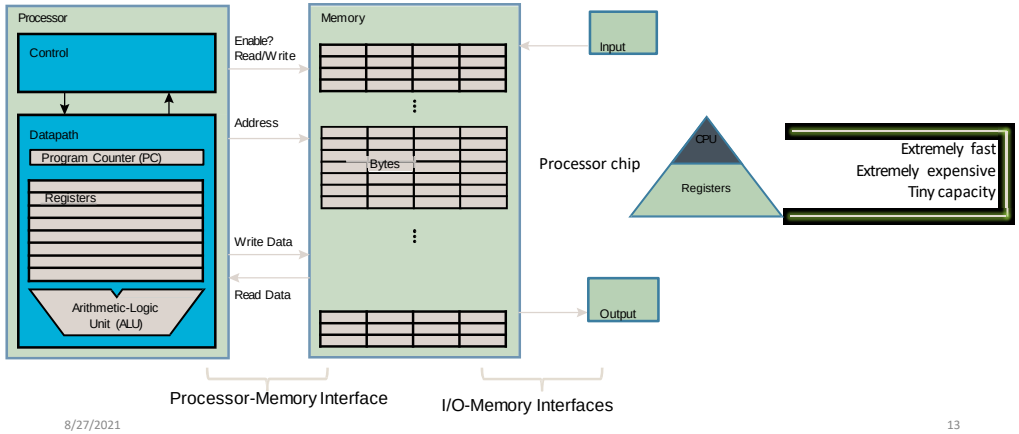
Operand of the Computer Hardware: RISC-V Addressing Summary



Assembly Variables: Registers (1/3)

- Elements of hardware: registers
- Unlike HLL like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Assembly operands are registers
  - Limited number of special locations built directly into the hardware
  - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they're very fast (faster than 0.25ns)
  - Recall light is $3x10^8$m/s = 0.3m/ns = 30cm/ns = 10cm/0.3ns!!!… where 0.3ns is the clock period of a 3.33GHz computer

Aside: Registers are Inside the Processor



Assembly Variables: Registers (2/3)

- Drawback: Since registers are in hardware, there is a predetermined number of them
  - Solution: RISC-V code must be very carefully put together to efficiently use registers
- 32 registers in RISC-V. Why 32?
  - Smaller is faster, but too small is bad. Goldilocks principle ("This porridge is too hot; This porridge is too cold; this porridge is just right")
- Each RISC-V register is 32 bits wide (in RV32 variant)
  - Groups of 32 bits called a word in RV32
  - P&H textbook uses the 64-bit variant RV64

Assembly Variables: Registers (3/3)

- Registers are numbered from 0 to 31
  - Referred to by number x0–x31
- X0 is special, always holds value zero
  - So only 31 registers able to hold variable values
- Each register can be referred to by number or name
  - Will add names later

Register	ABI Name	Description	Saver
x0	zero	hardwired zero	-
x1	ra	return address	Caller
x2	sp	stack pointer	Callee
x3	gp	global pointer	-
x4	tp	thread pointer	-
x5-7	t0-2	temporary registers	Caller
x8	s0 / fp	saved register / frame pointer	Callee
x9	s1	saved register	Callee
x10-11	a0-1	function arguments / return values	Caller
x12-17	a2-7	function arguments	Caller
x18-27	s2-11	saved registers	Callee
x28-31	t3-6	temporary registers	Caller

C, Java variables vs. registers

- In C (and most high-level languages) variables declared first and given a type. E.g.,

```
int fahr, celsius;
char a, b, c, d, e;
```

- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match int and char variables).
- In assembly language, the registers have no type
  - Operation determines how register contents are treated

Comments in Assembly

- Make your code more readable: comments!
  - Hash (#) is used for RISC-V comments anything from hash mark to end of line is a comment and will be ignored
- This is just like the C99 //

```
Note: Different from C.
```

- C comments have format `/* comment */` so they can span many lines
- In assembly language, each statement (called an Instruction), executes exactly one of a short list of simple commands
  - Unlike in C (and most other high-level languages), each line of assembly code contains at most 1 instruction
  - Instructions are related to operations (=, +, -, *, /) in C or Java

RISC-V Addition and Subtraction (1/4)

- Syntax of Instructions:

```
one two, three, four
add x1, x2, x3
```

- where:

```
one = operation by name
two = operand getting result (“destination,” x1)
three = 1st operand for operation (“source1,” x2)
four = 2nd operand for operation (“source2,” x3)
```

- Syntax is rigid: `1 operator, 3 operands`
- Why? Keep hardware simple via regularity

Addition and Subtraction of Integers (2/4)

- Addition in Assembly
  - Example: **`add x1,x2,x3 (in RISC-V)`**
  - Equivalent to: **`a = b + c (in C)`**
  - where C variables ⇔ RISC-V registers are: **`a ⇔ x1, b ⇔ x2, c ⇔ x3`**
- Subtraction in Assembly
  - Example: **`sub x3,x4,x5 (in RISC-V)`**
  - Equivalent to: **`d = e - f (in C)`**
  - where C variables ⇔ RISC-V registers are:

  `d ⇔ x3, e ⇔ x4, f ⇔ x5`

Addition and Subtraction of Integers (3/4)

- How to do the following C statement?

```
a = b + c + d - e;
```

- Break into multiple instructions

```
add x10, x1, x2 # a_temp = b + c
add x10, x10, x3 # a_temp = a_temp + d
sub x10, x10, x4 # a = a_temp - e
```

- Notice: A single line of C may break up into several lines of RISC-V.
- Notice: Everything after the hash mark on each line is ignored (comments).

Addition and Subtraction of Integers (4/4)

- How do we do this?

```
f = (g + h) - (i + j);
```

- Use intermediate temporary register

```
add x5, x20, x21 # a_temp = g + h
add x6, x22, x23 # b_temp = i + j
sub x19, x5, x6 # f = (g + h)- (i + j)
```


Immediates

- There is no Subtract Immediate in RISC-V: Why? There are **add** and **sub**, but no **addi** counterpart
- Limit types of operations that can be done to absolute minimum if an operation can be decomposed into a simpler
- where RISC-V registers **`x3,x4`** are associated with C variables f, g, respectively

```
addi x3,x4,-10 (in RISC-V)
f = g – 10 in C
```

Register Zero

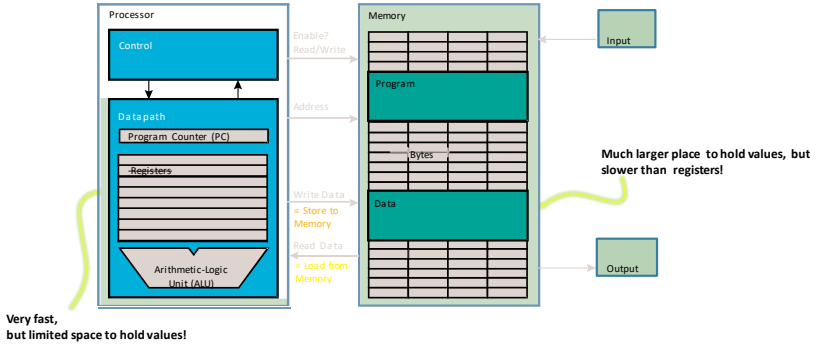
- One particular immediate, the number zero (0), appears very often in code.
- So the register zero (x0) is 'hard-wired' to value 0; e.g.

```
add x3,x4,x0   # in RISC V
f = g    # in C
```

- where RISC-V registers **x3,x4** are associated with C variables **f, g**
- Defined in hardware, so an instruction

```
add x0,x3,x4 # will not do anything!
```

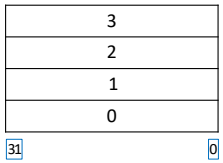

Data Transfer: Load from and Store to memory



RISC-V (32)

Memory Addresses are in Bytes

- Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type)–works fine if everything is a multiple of 8 bits
  - 8 bit chunk is called a byte (1 word = 4 bytes)
  - Memory addresses are really in bytes, not words
- Word addresses are 4 bytes apart
  - Word address is same as address of rightmost byte – least-significant byte (i.e. Little-endian convention)



Least-significant byte in a word

15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	0

31 24 23 16 15 8 7 0

Least-significant byte gets the smallest address

Homework_ week 2

- Exercise 2.1 – 2.11 (pg. 162 -164 in P&H textbook)

Big Endian vs. Little Endian

The adjective endian has its origin in the writings of 18th century writer Jonathan Swift. In the 1726 novel Gulliver's Travels, he portrays the conflict between sects of Lilliputians divided into those breaking the shell of a boiled egg from the big end or from the little end. He called them the "Big-Endians" and the "Little-Endians".

The order in which BYTES are stored in memory
Bits always stored as usual (E.g., 0xC2=0b 1100 0010)

Consider the number 1025 as we typically write it:

BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

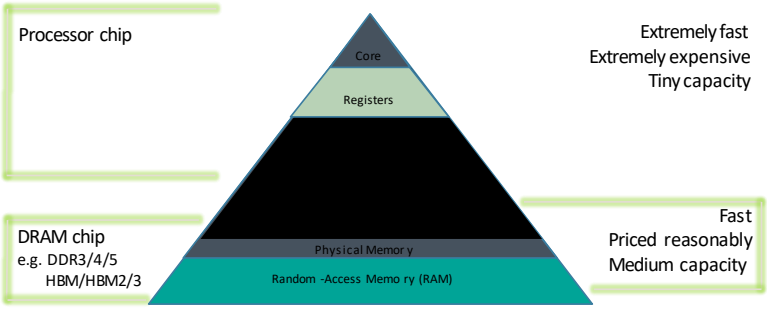
Big Endian			
ADDR3	ADDR2	ADDR1	ADDR0
BYTE0	BYTE1	BYTE2	BYTE3
10000000	00000100	00000000	00000000

Examples
Names in China or Hungary (e.g., Nikolić Bora)
Java Packages: (e.g., org.mypackage.HelloWorld)
Dates in ISO 8601 YYYY-MM-DD (e.g., 2020-09-07)
Eating Pizza crust first

Little Endian			
ADDR3	ADDR2	ADDR1	ADDR0
BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

Examples
Names in the US (e.g., BoraNikolić)
Internet names (e.g., cs.berkeley.edu)
Dates written in Europe DD/MM/YYYY (e.g., 07/09/2020)
Eating Pizza skinny part first

Speed of Registers vs. Memory



- Registers: 32 words (128 Bytes)
- Memory (DRAM): Billions of bytes (2 GB to 64 GB on laptop) and physics dictates that **Smaller is Faster**
- How much faster are registers than DRAM??
  - About 50-500 times faster! (in terms of latency of one access - tens of ns)
  - But subsequent words come every few ns

Load from Memory to Register

- C code

```
int  A[100];  g = h + A[3];
```

Data flow

- Using Load Word (lw) in RISC-V:

```
lw  x10,12(x15) # Reg x10 gets A[3]
add x11,x12,x10 # g = h + A[3]
# x15 – base register (pointer to A[0])  12 – offset in bytes
# Offset must be a constant known at assembly time
```


Store from Register to Memory

- C code

```
int   A[100];  A[10] = h + A[3];
```

- Using Store Word (sw) in RISC-V:

```
lw    x10,12(x15) # Temp reg x10 gets A[3]
add x10,x12,x10   # Temp reg x10 gets h + A[3]
sw    x10,40(x15) # A[10] = h + A[3]
```

Data flow

- Note:

```
x15 – base register (pointer) 12,40 – offsets in bytes
x15+12 and x15+40 must be multiples of 4
```

Loading and Storing Bytes

- In addition to word data transfers (lw, sw), RISC-V has byte data transfers:
  load byte: lb
  store byte: sb
- Same format as lw, sw. E.g.,

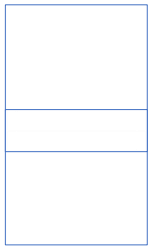
```
lb x10,3(x11)
```

-> contents of memory location with address = sum of "3" + contents of register x11 is copied to the low byte position of register x10.

RISC-V also has "unsigned byte" loads (`lbu`) which zero extends to fill register. Why no unsigned store byte '`sbu`'?

Example: What is in x12 ?

```
addi x11,x0,0x3F5
sw x11,0(x5)
lb x12,1(x5)
```



Substituting addi

- The following two instructions:

```
lw  x10,12(x15) # Temp reg x10 gets A[3]
add x12,x12,x10  # reg x12 = reg x12 + A[3]
```

- Replace addi:

```
addi x12, value # value in A[3]
```

- But involve a load from memory!
- Add immediate is so common that it deserves its own instruction!

RV32 So Far…

- Addition/subtraction

```
add rd, rs1, rs2   sub rd, rs1, rs2
```

- Add immediate

```
addi rd, rs1, imm
```

- Load/store

```
lw rd, rs1, imm
lb rd,rs1, imm
lbu rd,rs1, imm
sw rs1, rs2, imm
sb rs1, rs2, imm
```

Computer Decision Making

- Based on computation, do something different
- In programming languages: if-statement
- RISC-V: if-statement instruction is

```
beq reg1,reg2,L1
```

- means: go to statement labeled L1 if (value in reg1) == (value in reg2) ….otherwise, go to next statement
- beq stands for branch if equal
- Other instruction: bne for branch if not equal

Types of Branches

- Branch – change of control flow
- Conditional Branch – change control flow depending on outcome of comparison
  - branch *if* equal (**beq**) or branch *if not* equal (**bne**)
  - Also branch if less than (**blt**) and branch if greater than or equal (**bge**)
  - And unsigned versions (**bltu**, **bgeu**)
- Unconditional Branch – always branch
  - a RISC-V instruction for this: *jump* (**j**), as in
    j label

Example *if* Statement

- Assuming translations below, compile if block

f → **x10** g → **x11** h → **x12**
i → **x13** j → **x14**

```
if (i == j)
  f = g + h;
```

```
bne x13,x14,Exit
add x10,x11,x12
Exit:
```

- May need to negate branch condition

Example *if-else* Statement

- Assuming translations below, compile

f → **x10** g → **x11** h → **x12**
i → **x13** j → **x14**

```
if (i == j)
  f = g + h;
Else
f = g – h;
```

```
bne x13,x14,Else
add x10,x11,x12
j Exit
Else:sub x10,x11,x12
Exit:
```

Magnitude Compares in RISC-V

- General programs need to test < and > as well.
- RISC-V magnitude-compare branches:

"Branch on Less Than"

Syntax: **`blt reg1,reg2, Label`**

Meaning: **`if (reg1 < reg2) goto Label;`**

"Branch on Less Than Unsigned"

Syntax: **`bltu reg1,reg2, Label`**

Meaning:
```
if (reg1 < reg2)// treat registers
                   as unsigned integers
    goto label;
```

Also "Branch on Greater or Equal" **`bge`** and **`bgeu`**

Note: No '**`bgt`**' or '**`ble`**' instructions

Loops in C/Assembly

- There are three types of loops in C:
- while
- do … while
- for
- Each can be rewritten as either of the other two, so the same branching method can be applied to these loops as well.
- Key concept: Though there are multiple ways of writing a loop in RISC-V, the key to decision- making is conditional branch

C Loop Mapped to RISC-V Assembly

```
int A[20];
int sum = 0;
for (int i=0; i < 20; i++)
sum += A[i];
Assume that the base address of A, sum
    and i correspond to registers x9,x10
    and x11
```

```
    add x9, x8, x0 # x9=&A[0]
    add x10, x0, x0 # sum
    add x11, x0, x0 # i
    addi x13,x0, 20 # x13
Loop:
    bge x11,x13,Done
    lw x12, 0(x9)    # x12 A[i]
    add x10,x10,x12  # sum+=
    addi x9, x9,4    # &A[i+1]
    addi x11,x11,1   # i++
    j Loop
Done:
```


C Loop Mapped to RISC-V Assembly

```
Here is a traditional loop in C:
while (save[i] == k)
i += 1;
Assume that i and k correspond to registers x22 and x24 and the base
of the array save is in x25. What is the RISC-V assembly code
corresponding to this C code?
```

Computer Architecture 1

Computer Organization and Design
THE HARDWARE/SOFTWARE INTERFACE

[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolíc, © 2020, UC Berkeley]

RV32 So Far…

- Add/sub

```
add rd, rs1, rs2
sub rd, rs1, rs2
```

- Add immediate

```
addi rd, rs1, imm
```

- Load/store

```
lw  lb
lbu rd,
sw  rs1, rs2, imm
sb  rs1, rs2, imm
```

- Branching

```
beq rs1, rs2, Label
bne rs1, rs2, Label
bge rs1, rs2, Label
blt rs1, rs2, Label
bgeu rs1, rs2, Label
bltu rs1, rs2, Label
j Label
```


RISC-V Logical Instructions

- Useful to operate on fields of bits within a word
  - e.g., characters within a word (8 bits)
- Operations to pack /unpack bits into words
- Called logical operations

Logical operations	C operators	Java operators	RISC-V instructions
Bit-by-bit AND	&	&	**and**
Bit-by-bit OR	|	|	**or**
Bit-by-bit XOR	^	^	**xor**
Shift left logical	<<	<<	**sll**
Shift right logical	>>	>>	**srl**

RISC-V Logical Instructions

- Always two variants
  - Register: **`and x5, x6, x7 # x5 = x6 & x7`**
  - Immediate: **`andi x5, x6, 3 # x5 = x6 & 3`**
- Used for 'masks'
  - **`andi`** with **`0000 00FF`**$_{\text{hex}}$ isolates the least significant byte
  - **`andi`** with **`FF00 0000`**$_{\text{hex}}$ isolates the most significant byte
- There is no logical NOT in RISC-V
  - Use **`xor`** with **`11111111`**$_{\text{two}}$
  - Remember - simplicity…

Logical Shifting

- Shift Left Logical (**sll**) and immediate (**slli**):

```
slli x11,x12,2 #x11=x12<<2
```

  - Store in **x11** the value from **x12** shifted by 2 bits to the left (they fall off end), inserting 0's on right; << in C.
  - Before: $0000\ 0002_{hex}$

    $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$
  - After: $0000\ 0008_{hex}$

    $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1000_{two}$
  - What arithmetic effect does shift left have?
- Shift Right: **srl** is opposite **shift; >>**

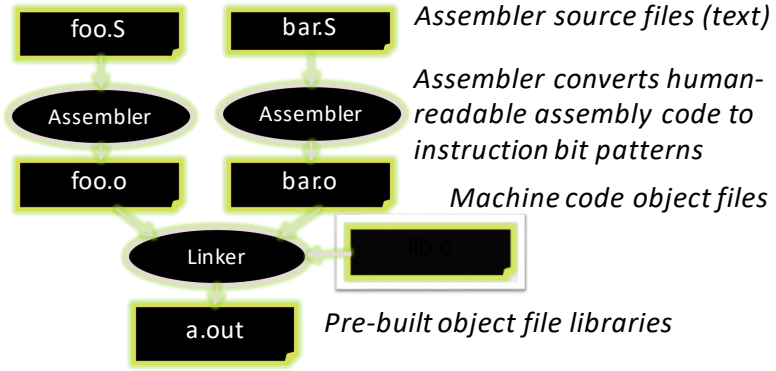
Arithmetic Shifting

- Shift right arithmetic (**sra**, **srai**) moves *n* bits to the right (insert high-order sign bit into empty bits)
- For example, if register x10 contained

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_{two} = -25_{ten}$
- If execute **srai x10, x10, 4**, result is:

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
- Unfortunately, this is NOT same as dividing by 2^n
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0

Assembler to Machine Code (More Later in Course)



Assembler source files (text)

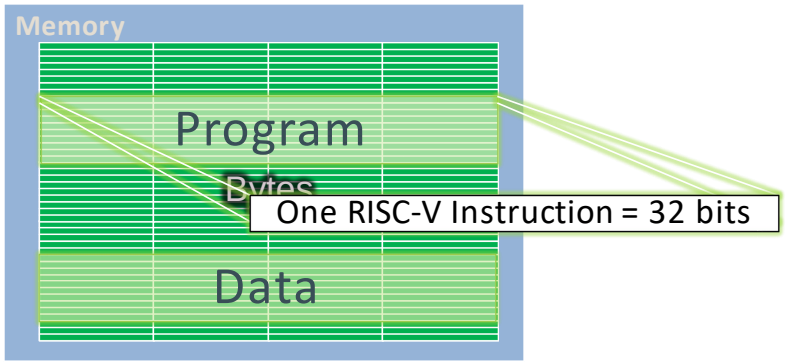
Assembler converts human-readable assembly code to instruction bit patterns

Machine code object files

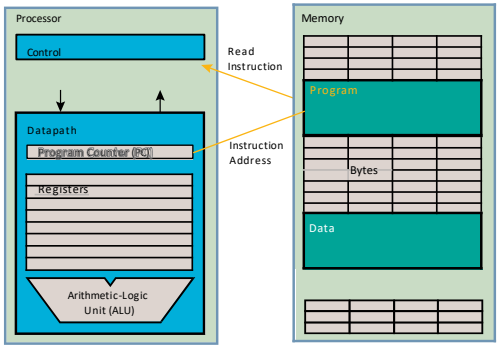
Pre-built object file libraries

Machine code executable file

How Program is Stored



Program Execution



- **PC** (program counter) is a register internal to the processor that holds byte address of next instruction to be executed

Helpful RISC-V Assembler Features

- Symbolic register names
  - E.g., **`a0-a7`** for argument registers (**`x10-x17`**) for function calls
  - E.g., **`zero`** for **`x0`**
- Pseudo-instructions
  - Shorthand syntax for common assembly idioms
  - E.g., **`mv rd, rs`** = **`addi rd, rs, 0`**
  - E.g., **`li rd, 13`** = **`addi rd, x0, 13`**
  - E.g., **`nop`** = **`addi x0, x0, 0`**

RISC V Function calls

```
main() {
   int i,j,k,m;
   ...
   i = mult(j,k); ...
   m = mult(i,i); ...

/* really dumb mult function */
int mult (int mcand, int mlier){
   int product = 0;
   while (mlier > 0)  {
     product = product + mcand;
     mlier = mlier -1; }
   return product;
   }
```

What information mus compiler/programmer keep track of?

What instructions can accomplish this?

Six Fundamental Steps in Calling a Function

1. Put arguments in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put return value in a place where calling code can access it and restore any registers you used; release local storage
6. Return control to point of origin, since a function can be called from several points in a program

RISC-V Function Call Conventions

- Registers faster than memory, so use them
- **a0–a7** (**x10–x17**): eight *argument* registers to pass parameters and two return values (**a0–a1**)
- **ra**: one *return address* register to return to the point of origin (**x1**)
- Also **s0–s1** (**x8–x9**) and **s2–s11** (**x18–x27**): saved registers (more about those later)

Instruction Support for Functions (1/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
    }
          int sum(int x, int y) {
          return x+y;
    }
```

RISC-V

```
address (shown in decimal)
    1000
    1004
    1008
    1012
    1016
    …
    2000
    2004
```

In RISC-V, all instructions are 4 bytes, and stored in memory just like data. So, here we show the addresses of where the programs are stored.

Instruction Support for Functions (2/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
    }
          int sum(int x, int y) {
          return x+y;
    }
```

RISC-V

```
address (shown in decimal)
    1000 mv a0,s0            # x = a
    1004 mv a1,s1            # y = b
    1008 addi ra,zero,1016   #ra=1016
    1012 j    sum            #jump to sum
    1016 …                   # next inst.
    …
    2000 sum: add a0,a0,a1
    2004 jr   ra #new instr."jump reg"
```


Instruction Support for Functions (3/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
    }
          int sum(int x, int y) {
          return x+y;
    }
```

RISC-V

- Question: Why use **jr** here? Why not use **j**?
- Answer: **sum** might be called by many places, so we can't return to a fixed place. The calling proc to **sum** must be able to say "return here" somehow.

```
    …
2000 sum: add a0,a0,a1
2004 jr   ra #new instr."jump reg"
```

Instruction Support for Functions (4/4)

- Single instruction to jump and save return address: jump and link (**`jal`**)
- Before:

```
1008 addi ra,zero,1016      # ra=1016
1012 j sum                  # goto sum
```

- After

```
1008 jal sum  # ra=1012,goto sum
```

- Why have a **`jal`**?
  - Make the common case fast: function calls very common
  - Reduce program size
  - Don't have to know where code is in memory with **`jal`**!

RISC-V Function Call Instructions

- Invoke function: *jump and link* instruction (**`jal`**) (really should be **`laj`** *"link and jump"*)
  - "link" means form an *address* or *link* that points to calling site to allow function to return to proper address
  - Jumps to address and simultaneously saves the address of the following instruction in register ra

```
jal FunctionLabel
```

- Return from function: *jump register* instruction (**`jr`**)
  - Unconditional jump to address specified in register: **`jr ra`**
  - Assembler shorthand: **`ret = jr ra`**

Summary of Instruction Support

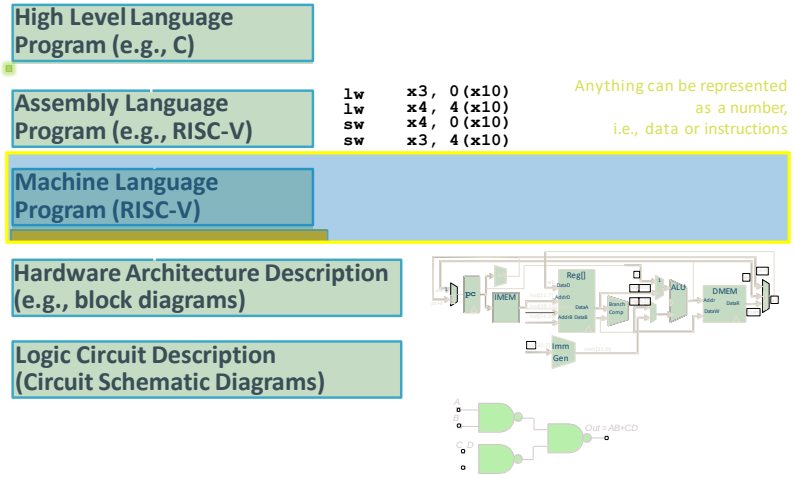
Actually, only two instructions:

- `jal rd, Label` – jump-and-link
- `jalr rd, rs, imm` – jump-and-link register

`j, jr` and `ret` are pseudoinstructions!

- `j: jal x0, Label`

RISC-V Instruction Representation



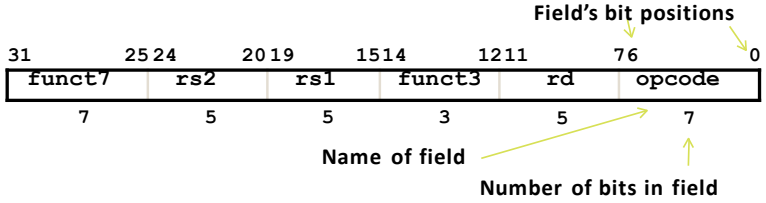
Instructions as Numbers (1/2)

- Most data we work with is in words (32-bit chunks):
  - Each register is a word
  - `lw` and `sw` both access memory one word at a time
- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so assembler string "`add x10,x11,x0`" is meaningless to hardware
  - RISC-V seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words also
    - Same 32-bit instructions used for RV32, RV64, RV128

Instructions as Numbers (2/2)

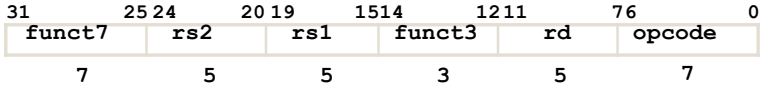
- One word is 32 bits, so divide instruction word into "fields"
- Each field tells processor something about instruction
- We could define different fields for each instruction, but RISC-V seeks simplicity, so define six basic types of instruction formats:
  - R-format for register-register arithmetic operations
  - I-format for register-immediate arithmetic operations and loads
  - S-format for stores
  - B-format for branches (minor variant of S-format)
  - U-format for 20-bit upper immediate instructions
  - J-format for jumps (minor variant of U-format)

R-Format Instruction Layout



- 32-bit instruction word divided into six fields of varying numbers of bits each: 7+5+5+3+5+7 = 32
- Examples
  - `opcode` is a 7-bit field that lives in bits 6-0 of the instruction
  - `rs2` is a 5-bit field that lives in bits 24-20 of the instruction

R-Format Instructions opcode/funct Fields



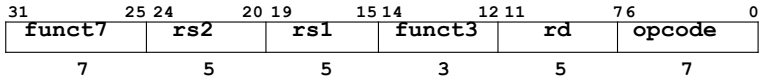
opcode: partially specifies what instruction it is

- Note: This field is equal to 0110011_{two} for all R-Format register-register arithmetic instructions

`funct7+funct3`: combined with **`opcode`**, these two fields describe what operation to perform

- **Question: You have been professing simplicity, so why aren't `opcode` and `funct7` and `funct3` a single 17- bit field?**
  - **We'll answer this later**

R-Format Instructions Register Specifiers



`rs1` (Source Register #1): specifies register containing first operand

`rs2` : specifies second register operand

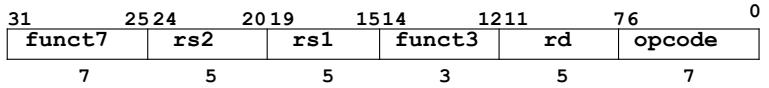
`rd` (Destination Register): specifies register which will receive result of computation

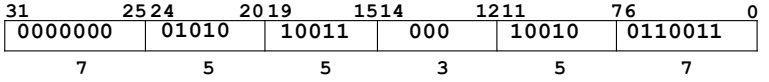
Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (**`x0`**-**`x31`**)

R-Format Example

- RISC-V Assembly Instruction:

```
add  x18,x19,x10
```





`add` `rs2=10` `rs1=19` `add` `rd=18` `Reg-Reg OP`

Your Turn

- What is correct encoding of **`add x4, x3, x2`** ?

1) $4021\,8233_{hex}$
2) $0021\,82b3_{hex}$
3) $4021\,82b3_{hex}$
4) $0021\,8233_{hex}$
5) $0021\,8234_{hex}$

31 25 24 20 19 15 14 12 11 7 6 0

0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	110	rd	0110011	or
						and

All RV32 R-format Instructions

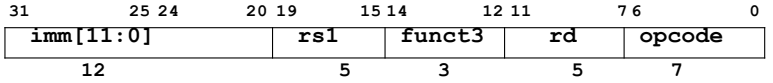
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110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

Different encoding in funct7 + funct3 selects different operations
Can you spot two new instructions?

I-Format Instructions

- What about instructions with immediates?
  - Compare:
    - `add rd, rs1, rs2`
    - `addi rd, rs1, imm`
  - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  - Ideally, RISC-V would have only one instruction format (for simplicity): unfortunately, we need to compromise
- Define new instruction format that is mostly consistent with R-format
  - Notice if instruction has immediate, then uses at most 2 registers (one source, one destination)

I-Format Instruction Layout

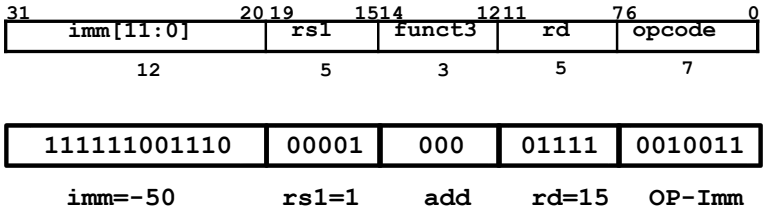


- Only one field is different from R-format, **`rs2`** and **`funct7`** replaced by 12-bit signed immediate, **`imm[11:0]`**
- Remaining fields (**`rs1`**, **`funct3`**, **`rd`**, **`opcode`**) same as before
- **`imm[11:0]`** can hold values in range [-2048_{ten}, $+2047_{ten}$]
- Immediate is always sign-extended to 32-bits before use in an arithmetic operation
- We'll later see how to handle immediates > 12 bits

I-Format Example

- RISC-V Assembly Instruction:

  `addi x15,x1,-50`



All RV32 I-format Arithmetic Instructions

imm[11:0]	rs1	000	rd	0010011	addi
imm[11:0]	rs1	010	rd	0010011	slti
imm[11:0]	rs1	011	rd	0010011	sltiu
imm[11:0]	rs1	100	rd	0010011	xori
imm[11:0]	rs1	110	rd	0010011	ori
imm[11:0]	rs1	111	rd	0010011	andi
0000000 shamt	rs1	001	rd	0010011	slli
0000000 shamt	rs1	101	rd	0010011	srli
0100000 shamt	rs1	101	rd	0010011	srai

One of the higher-order immediate bits is used to distinguish "shift right logical" (SRLI) from "shift right arithmetic" (SRAI)

"Shift-by-immediate" instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

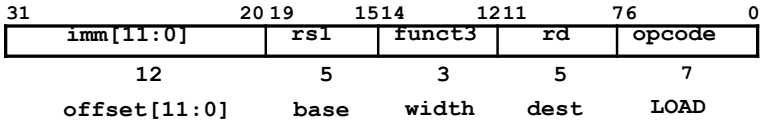
Xác định mã máy (Hệ hex) cho các lệnh sau

- add x1, x2, x3 : 0x003100B3
- addi x8, x6, -10: 0xFF630413
- ori x7, x10, 0x04: 0x00456393
- slli x5, x12, 2: 0x00261293

0000 000	0 0011	0001 0	000	0000 1	011 0011	add

1111 1111 0110	0011 0	000	0100 0	001 0011	addi
0000 0000 0100	0101 0	110	0011 1	001 0011	ori
0000 000 0 0010	0110 0	001	0010 1	001 0011	slli

Load Instructions are also I-Type

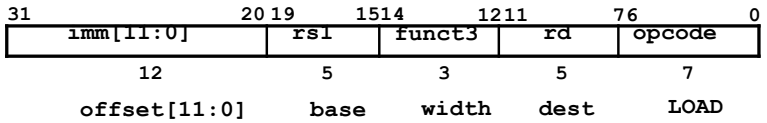


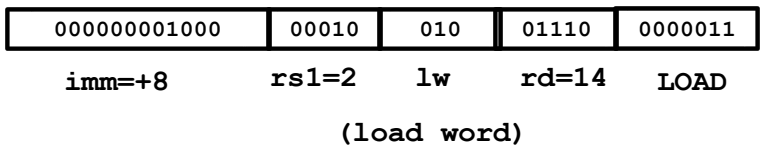
- The 12-bit signed immediate is added to the base address in register `rs1` to form the memory address
  - This is very similar to the add-immediate operation but used to create address not to create final result
- The value loaded from memory is stored in register `rd`

I-Format Load Example

- RISC-V Assembly Instruction:

  `lw x14, 8(x2)`





(load word)

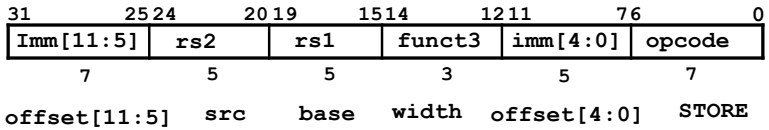
All RV32 Load Instructions

imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	001	rd	0000011	lh
imm[11:0]	rs1	010	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu

funct3 field encodes size and 'signedness' of load data

- **lbu** is "load unsigned byte"
- **lh** is "load halfword", which loads 16 bits (2 bytes) and sign- extends to fill destination 32-bit register
- **lhu** is "load unsigned halfword", which zero-extends 16 bits to fill destination 32-bit register
- There is no '**lwu**' in RV32, because there is no sign/zero extension needed when copying 32 bits from a memory location into a 32- bit register

S-Format Used for Stores

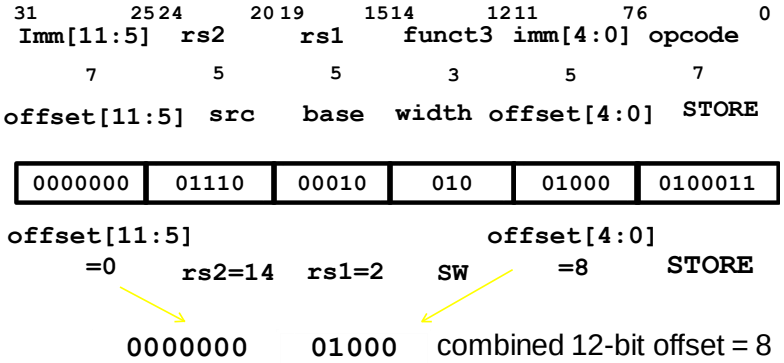


- Store needs to read two registers, **rs1** for base memory address, and **rs2** for data to be stored, as well immediate offset!
- Can't have both **rs2** and immediate in same place as other instructions!
- Note that stores don't write a value to the register file, ***no rd***!
- RISC-V design decision is to move low 5 bits of immediate to where **rd** field was in other instructions – keep **rs1**/**rs2** fields in same place
  - Register names more critical than immediate bits in hardware design

S-Format Example

- RISC-V Assembly Instruction:

```
sw x14, 8(x2)
```



All RV32 Store Instructions

- Store byte, halfword, word

Imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	**sb**
Imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	**sh**
Imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	**sw**
			width			

RISC-V Conditional Branches

- E.g., **beq x1, x2, Label**
- Branches read two registers but don't write to a register (similar to stores)
- How to encode label, i.e., where to branch to?

Branching Instruction Usage

- Branches typically used for loops (**`if-else`**, **`while`**, **`for`**)
  - Loops are generally small (< 50 instructions)
  - Function calls and unconditional jumps handled with jump instructions (J-Format)
- **Recall:** Instructions stored in a localized area of memory (Code/Text)
  - Largest branch distance limited by size of code
  - Address of current instruction stored in the program counter (PC)

PC-Relative Addressing

PC-Relative Addressing: Use the **`immediate`** field as a two's-complement offset to PC

- Branches generally change the PC by a small amount
- Can specify $\pm 2^{11}$ 'unit' addresses from the PC
- (We will see in a bit that we can encode 12-bit offsets as immediates)

- Why not use byte as a unit of offset from PC?
  - Because instructions are 32-bits (4-bytes)
  - We don't branch into middle of instruction

Scaling Branch Offset

- One idea: To improve the reach of a single branch instruction, multiply the offset by four bytes before adding to PC
- This would allow one branch instruction to reach $\pm 2^{11} \times$ 32-bit instructions either side of PC
  - Four times greater reach than using byte offset

Branch Calculation

- If we don't take the branch:
  `PC = PC + 4` (i.e., next instruction)
- If we do take the branch:
  `PC = PC + immediate*4`

- Observations:
  - **`immediate`** is number of instructions to jump (remember, specifies words) either forward (+) or backwards (–)

RISC-V Feature, n × 16-bit Instructions

- Extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 16-bits in length
- To enable this, RISC-V scales the branch offset by 2 bytes even when there are no 16-bit instructions
- Reduces branch reach by half and means that ½ of possible targets will be errors on RISC-V processors that only support 32-bit instructions (as used in this class)
- RISC-V conditional branches can only reach $\pm 2^{10}$ × 32-bit instructions on either side of PC

RISC-V B-Format for Branches

0011111	01011	01010	001	01000	1100111
imm[12:6]	rs2	rs1	funct3	imm[5:1]	opcode

- B-format is mostly same as S-Format, with two register sources (**rs1**/**rs2**) and a 12-bit immediate **imm[12:1]**
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode *even* 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

Branch Example, Determine Offset

- RISC-V Code:

```
Loop:  beq   x19,x10,End
       add   x18,x18,x10
       addi  x19,x19,-1
       j     Loop
End:   # target instruction
```

Count instructions from branch: 1, 2, 3, 4

- Branch offset =

4 × 32-bit instructions = 16 bytes

- (Branch with offset of 0, branches to itself)

Branch Example, Determine Offset

- RISC-V Code:

```
Loop:  beq   x19,x10,End
       add   x18,x18,x10
       addi  x19,x19,-1
       j     Loop
End:   # target instruction
```

Count instructions from branch: 1, 2, 3, 4

```
???????  01010  10011  000  ?????  1100011
 imm     rs2=10 rs1=19  BEQ  imm    BRANCH
```

Branch Example, Determine Offset

- RISC-V Code:

```
Loop:  beq  x19,x10,End
       add  x18,x18,x10
       addi x19,x19,-1
       j    Loop
End:   # target instruction
```

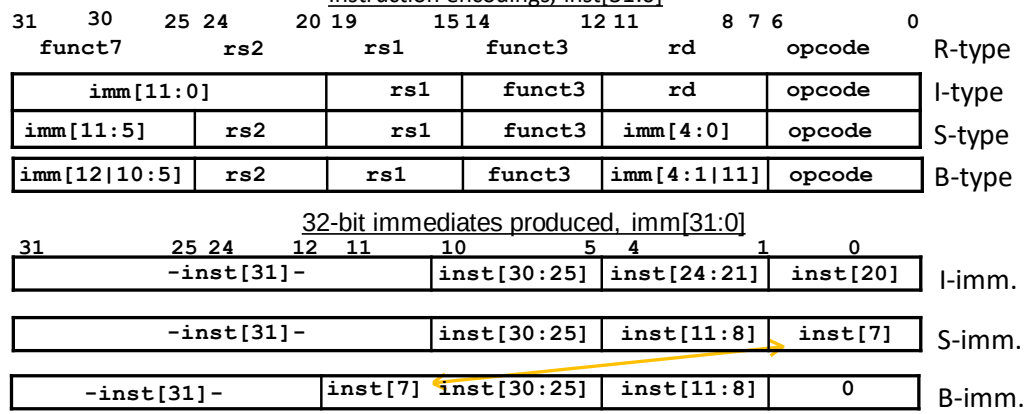
Offset = 16 bytes = 8 x 2

1 2 3 4

```
                                        01000
???????  01010  10011  000  ?????  1100011
imm      rs2=10 rs1=19  BEQ  imm    BRANCH
```


RISC-V Immediate Encoding

Instruction encodings, inst[31:0]



32-bit immediates produced, imm[31:0]

Upper bits sign-extended from `inst[31]` always

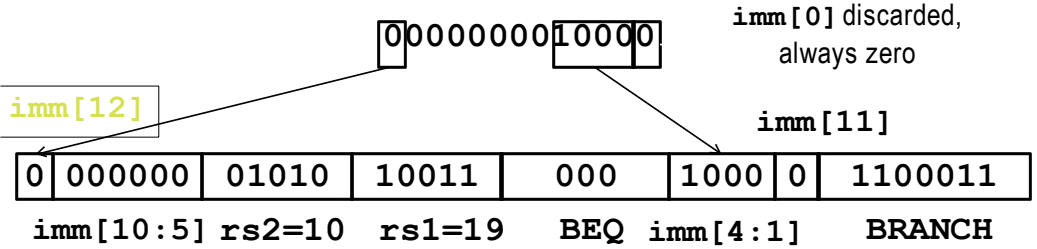
Only bit 7 of instruction changes role in immediate between S and B

Branch Example, Complete Encoding

`beq x19,x10, offset = 16 bytes`

13-bit immediate, `imm[12:0]`, with value 16

`imm[0]` discarded, always zero



All RISC-V Branch Instructions

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	bgeu

Questions on PC-addressing

- Does the value in branch immediate field change if we move the code?
  - If moving individual lines of code, then yes
  - If moving all of code, then no ('position-independent code')
- What do we do if destination is > 2^{10} instructions away from branch?
  - Other instructions save us

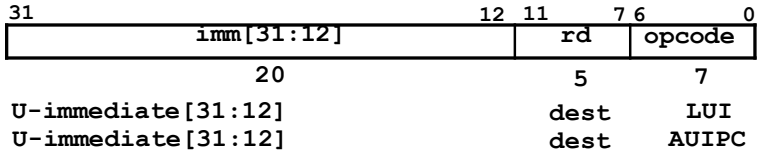
Questions on PC-addressing

- Does the value in branch immediate field change if we move the code?
  - If moving individual lines of code, then yes
  - If moving all of code, then no ('position-independent code')
- What do we do if destination is > 2^{10} instructions away from branch?
  - Other instructions save us

```
beq x10,x0,far

# next instr     →  bne x10,x0,next
                    j   far
                next: # next instr
```


U-Format for "Upper Immediate" Instructions



- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
  - **`lui`** – Load Upper Immediate
  - **`auipc`** – Add Upper Immediate to PC

LUI to Create Long Immediates

- LUI writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
- Together with an **`addi`** to set low 12 bits, can create any 32-bit value in a register using two instructions (**`lui`**/**`addi`**).

```
lui x10, 0x87654        # x10 = 0x87654000
addi x10, x10, 0x321    # x10 = 0x87654321
```

One Corner Case

How to set **0xDEADBEEF**?

```
lui x10, 0xDEADB      # x10 = 0xDEADB000
addi x10, x10, 0xEEF # x10 = 0xDEADAEEF
```

addi 12-bit immediate is always sign-extended, if top bit is set, will subtract -1 from upper 20 bits

Solution

How to set **0xDEADBEEF**?

```
LUI x10, 0xDEADC   # x10 = 0xDEADC000

ADDI x10, x10, 0xEEF  # x10 =
                      #0xDEADBEEF
```

Pre-increment value placed in upper 20 bits, if sign bit will be set on immediate in lower 12 bits.

Assembler pseudo-op handles all of this:

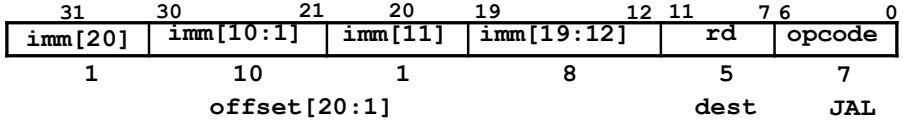
```
li x10, 0xDEADBEEF # Creates two
                   #instructions
```


AUIPC

- Adds upper immediate value to PC and places result in destination register
- Used for PC-relative addressing

```
Label: AUIPC x10, 0 # Puts address of
                    # Label in x10
```


J-Format for Jump Instructions



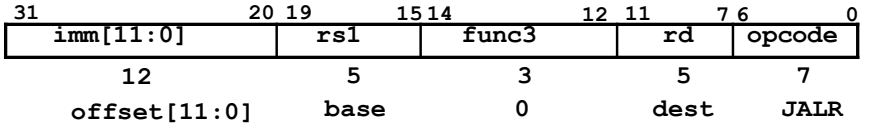
- **jal** saves PC+4 in register **rd** (the return address)
  - Assembler "**j**" jump is pseudo-instruction, uses JAL but sets **rd=x0** to discard return address
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

Uses of JAL

```
# j pseudo-instruction
j Label = jal x0, Label # Discard return address

# Call function within 2^18 instructions of PC
jal ra, FuncName
```


JALR Instruction (I-Format)

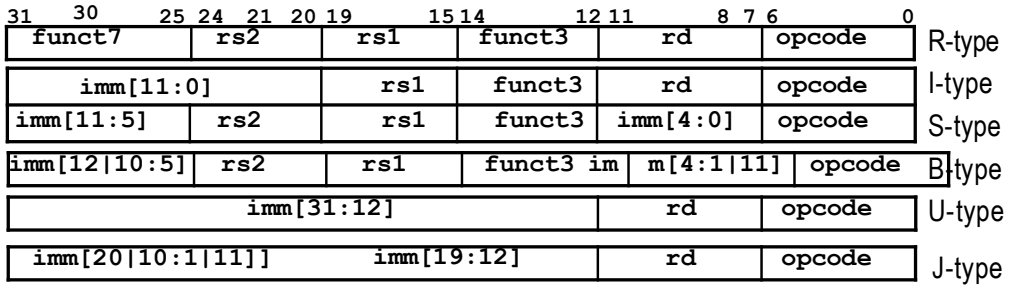


- **`jalr rd, rs, immediate`**
  - Writes PC+4 to rd (return address)
  - Sets PC = **`rs + immediate`**
  - Uses same immediates as arithmetic and loads
    - ***no*** multiplication by 2 bytes
    - In contrast to branches and **`jal`**

Uses of JALR

```
# ret and jr psuedo-instructions
ret = jr ra = jalr x0, ra, 0
# Call function at any 32-bit absolute address
lui x1, <hi20bits>
jalr ra, x1, <lo12bits>
# Jump PC-relative with 32-bit offset
auipc x1, <hi20bits>
jalr x0, x1, <lo12bits>
```


Summary of RISC-V Instruction Formats



Complete RV32I ISA

Open Reference Card ¨

Base Integer Instructions:RV32I

Category	Name	Fmt	RV32I Base
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt
	Shift Right Logical	R	SRL rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm
	Add Upper Imm to PC	U	AUIPC rd,imm
Logical	XOR	R	XOR rd,rs1,rs2
	XOR Immediate	I	XORI rd,rs1,imm
	OR	R	OR rd,rs1,rs2
	OR Immediate	I	ORI rd,rs1,imm
	AND	R	AND rd,rs1,rs2
	AND Immediate	I	ANDI rd,rs1,imm
Compare	Set <	R	SLT rd,rs1,rs2
	Set < Immediate	I	SLTI rd,rs1,imm
	Set < Unsigned	R	SLTU rd,rs1,rs2
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm
Branches	Branch =	B	BEQ rs1,rs2,imm
	Branch ≠	B	BNE rs1,rs2,imm
	Branch <	B	BLT rs1,rs2,imm
	Branch ≥	B	BGE rs1,rs2,imm
	Branch < Unsigned	B	BLTU rs1,rs2,imm
	Branch ≥ Unsigned	B	BGEU rs1,rs2,imm
Jump & Link&L		J	JAL rd,imm
	Jump & Link Register	I	JALR rd,rs1,imm
Loads	Load Byte	I	LB rd,rs1,imm
	Load Halfword	I	LH rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm
	Load Half Unsigned	I	LHU rd,rs1,imm
	Load Word	I	LW rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm
	Store Halfword	S	SH rs1,rs2,imm
	Store Word	S	SW rs1,rs2,imm
Synch	Synch thread	I	FENCE
Environment	CALL	I	ECALL
	BREAK	I	EBREAK

Not in 61C

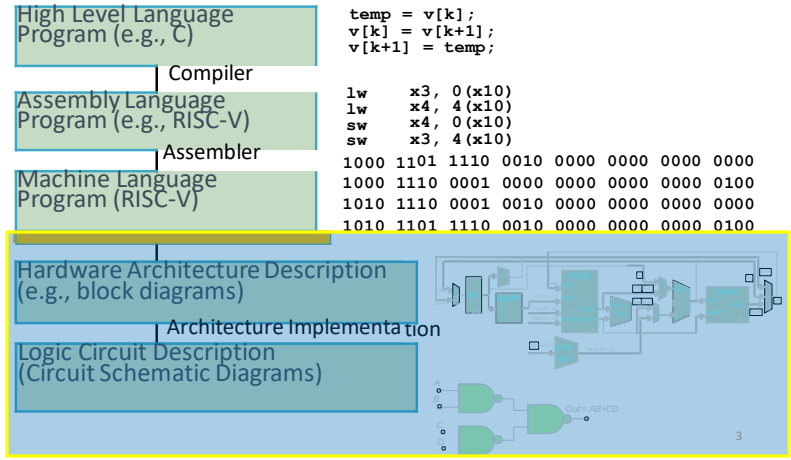
Computer Architecture 1

Computer Organization and Design
THE HARDWARE/SOFTWARE INTERFACE

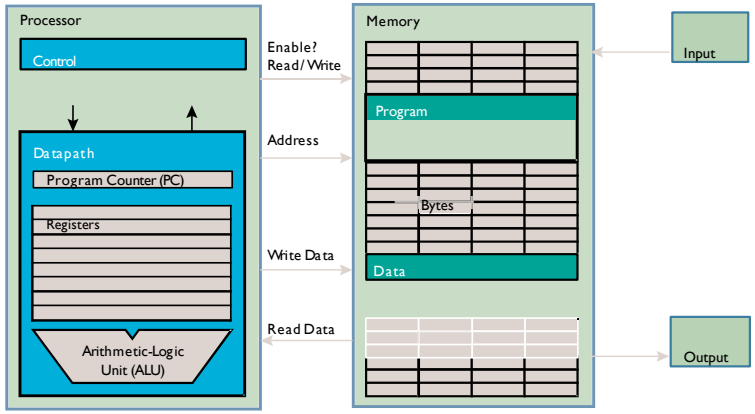
[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolić, © 2020, UC Berkeley]

RISC-V Processor Design

Great Idea #1: Abstraction (Levels of Representation/Interpretation)



Our Single-Core Processor So Far…



The CPU

- Processor (CPU): the active part of the computer that does all the work (data manipulation and decision-making)
- Datapath: portion of the processor that contains hardware necessary to perform operations required by the processor (the brawn)
- Control: portion of the processor (also in hardware) that tells the datapath what needs to be done (the brain)

Need to Implement All RV32I Instructions

RISC-V

Open Reference Card

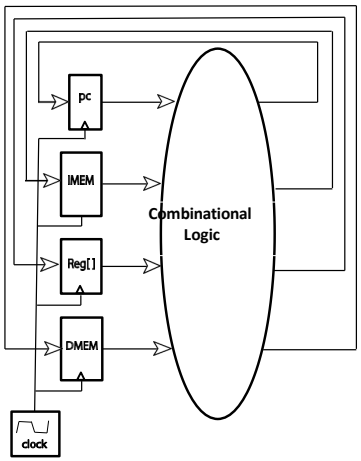
Base Integer Instructions: RV32I

Category	Name	Fmt	RV32I Base	Category	Name	Fmt	RV32I Base
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2	Loads	Load Byte	I	LB rd,rs1,imm
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt		Load Halfword	I	LH rd,rs1,imm
	Shift Right Logical	R	SRL rd,rs1,rs2		Load Byte Unsigned	I	LBU rd,rs1,imm
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt		Load Half Unsigned	I	LHU rd,rs1,imm
	Shift Right Arithmetic	R	SRA rd,rs1,rs2		Load Word	I	LW rd,rs1,imm
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	Stores	Store Byte	S	SB rs1,rs2,imm
Arithmetic	ADD	R	ADD rd,rs1,rs2		Store Halfword	S	SH rs1,rs2,imm
	ADD Immediate	I	ADDI rd,rs1,imm		Store Word	S	SW rs1,rs2,imm
	SUBtract	R	SUB rd,rs1,rs2	Branches	Branch =	B	BEQ rs1,rs2,imm
	Load Upper Imm	U	LUI rd,imm		Branch ≠	B	BNE rs1,rs2,imm
	Add Upper Imm to PC	U	AUIPC rd,imm		Branch <	B	BLT rs1,rs2,imm
Logical	XOR	R	XOR rd,rs1,rs2		Branch ≥	B	BGE rs1,rs2,imm
	XOR Immediate	I	XORI rd,rs1,imm		Branch < Unsigned	B	BLTU rs1,rs2,imm
	OR	R	OR rd,rs1,rs2		Branch ≥ Unsigned	B	BGEU rs1,rs2,imm
	OR Immediate	I	ORI rd,rs1,imm	Jump & Link	J&L	J	JAL rd,imm
	AND	R	AND rd,rs1,rs2		Jump & Link Register	I	JALR rd,rs1,imm
	AND Immediate	I	ANDI rd,rs1,imm				
Compare	Set <	R	SLT rd,rs1,rs2	Synch	Synch thread	I	FENCE
	Set < Immediate	I	SLTI rd,rs1,imm				
	Set < Unsigned	R	SLTU rd,rs1,rs2	Environment	CALL	I	ECALL
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm		BREAK	I	EBREAK

Not in 61C

Building a RISC-V Processor

One-Instruction-Per-Cycle RISC-V Machine



- On every tick of the clock, the computer executes one instruction
- Current state outputs drive the inputs to the combinational logic, whose outputs settles at the values of the state before the next clock edge
- At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle

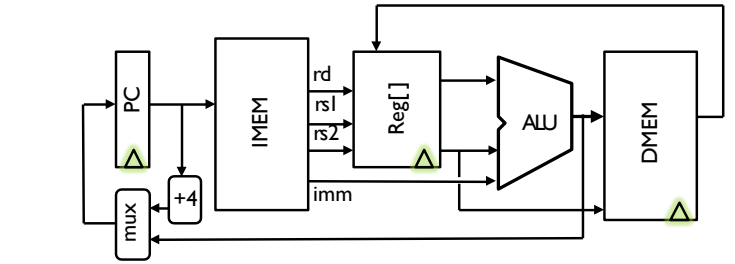
Stages of the Datapath : Overview

- Problem: a single, "monolithic" block that "executes an instruction" (performs all necessary operations beginning with fetching the instruction) would be too bulky and inefficient
- Solution: break up the process of "executing an instruction" into stages, and then connect the stages to create the whole datapath
  - smaller stages are easier to design
  - easy to optimize (change) one stage without touching the others (modularity)

Five Stages of the Datapath

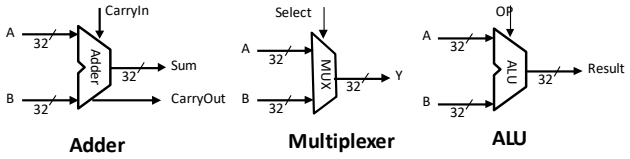
- **Stage 1: *Instruction Fetch (IF)***
- **Stage 2: *Instruction Decode (ID)***
- **Stage 3: *Execute (EX) - ALU* (Arithmetic-Logic Unit)**
- **Stage 4: *Memory Access (MEM)***
- **Stage 5: *Write Back to Register (WB)***

Basic Phases of Instruction Execution



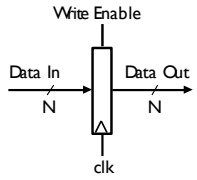
Datapath Components: Combinational

- Combinational elements



- Storage elements + clocking methodology
- Building blocks

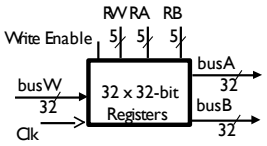
Datapath Elements: State and Sequencing (1/3)



- Register
- Write Enable:
  - Low (or deasserted) (0): Data Out will not change
  - Asserted (1): Data Out will become Data In on positive edge of clock

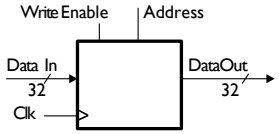
Datapath Elements: State and Sequencing (2/3)

- Register file (regfile, RF) consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
- Clock input (Clk)
  - Clk input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid ⇒ busA or busB valid after "access time."



Datapath Elements: State and Sequencing (3/3)

- "Magic" Memory
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is found by:
  - For Read: Address selects the word to put on Data Out
  - For Write: Set Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block: Address valid ⇒ Data Out valid after "access time"



State Required by RV32I ISA (1/2)

Each instruction during execution reads and updates the state of :
(1) Registers, (2) Program counter, (3) Memory

- Registers (**`x0..x31`**)
  - Register file (*regfile*) **`Reg`** holds 32 registers x 32 bits/register: **`Reg[0]..Reg[31]`**
  - First register read specified by **`rs1`** field in instruction
  - Second register read specified by **`rs2`** field in instruction
  - Write register (destination) specified by *rd* field in instruction
  - **`x0`** is always 0 (writes to **`Reg[0]`** are ignored)
- Program Counter (**`PC`**)
  - Holds address of current instruction

State Required by RV32I ISA (2/2)

- Memory (**`MEM`**)
  - Holds both instructions & data, in one 32-bit byte-addressed memory space
  - We'll use separate memories for instructions (**`IMEM`**) and data (**`DMEM`**)
    - *These are placeholders for instruction and data caches*
  - Instructions are read (*fetched*) from instruction memory (assume **`IMEM`** read-only)
  - Load/store instructions access data memory

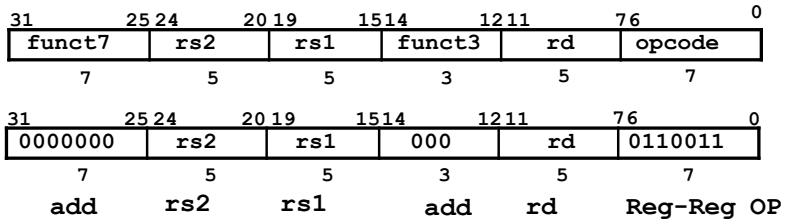
R-Type Add Datapath

Review: R-Type Instructions

3 1 30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0
R-format : ALU					
[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]
7	5	5	3	5	7
func7	rs2	rs1	func3	rd	opcode
0000000	rs2	rs1	000 : ADD	rd	0110011:OP-R
0100000	rs2	rs1	000 : SUB	rd	0110011:OP-R
0000000	rs2	rs1	001 : SLL	rd	0110011:OP-R
0000000	rs2	rs1	010 : SLT	rd	0110011:OP-R
0000000	rs2	rs1	011 : SLTU	rd	0110011:OP-R
0000000	rs2	rs1	100 : XOR	rd	0110011:OP-R
0000000	rs2	rs1	101 : SRL	rd	0110011:OP-R
0100000	rs2	rs1	101 : SRA	rd	0110011:OP-R
0000000	rs2	rs1	110 : OR	rd	0110011:OP-R
0000000	rs2	rs1	111 : AND	rd	0110011:OP-R

- E.g. Addition/subtraction **`add rd, rs1, rs2`**
  `R[rd] = R[rs1] + R[rs2]`
  `sub rd, rs1, rs2`
  `R[rd] = R[rs1] - R[rs2]`

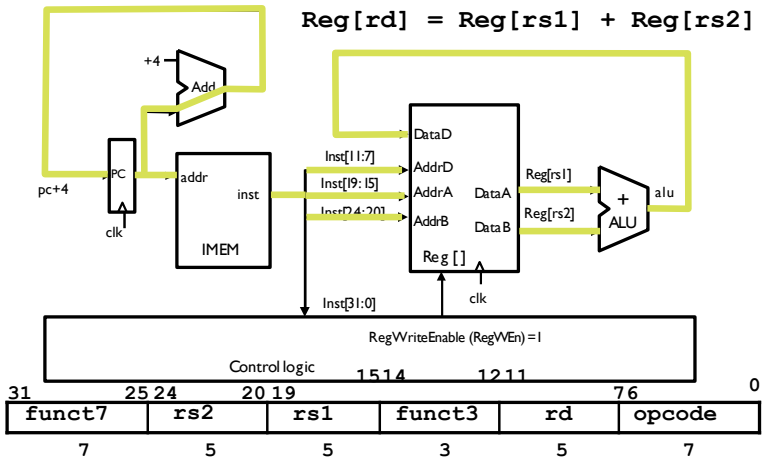
Implementing the add instruction



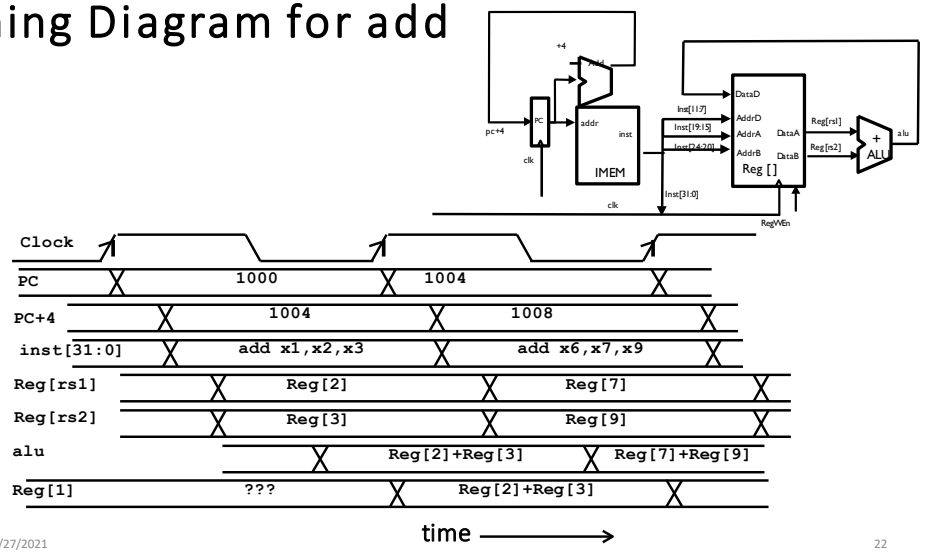
`add rd, rs1, rs2`

- Instruction makes two changes to machine's state:
  - **`Reg[rd] = Reg[rs1] + Reg[rs2]`**
  - **`PC = PC + 4`**

Datapath for add



Timing Diagram for add



Sub Datapath

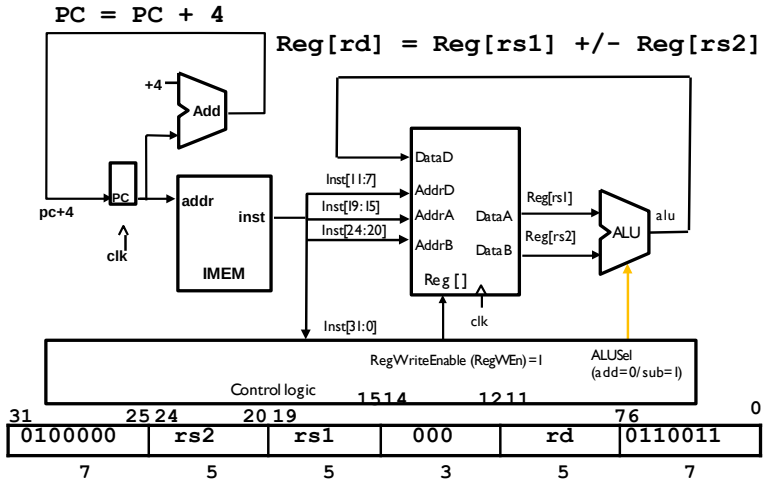
Implementing the sub instruction

0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub

`sub rd, rs1, rs2`

- Almost the same as add, except now have to subtract operands instead of adding them
- **`inst[30]`** selects between add and subtract

Datapath for add/sub



Implementing Other R-Format Instructions

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

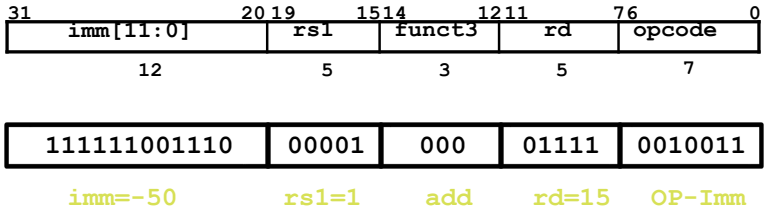
All implemented by decoding funct3 and funct7 fields and selecting appropriate ALU function

Datapath With Immediates

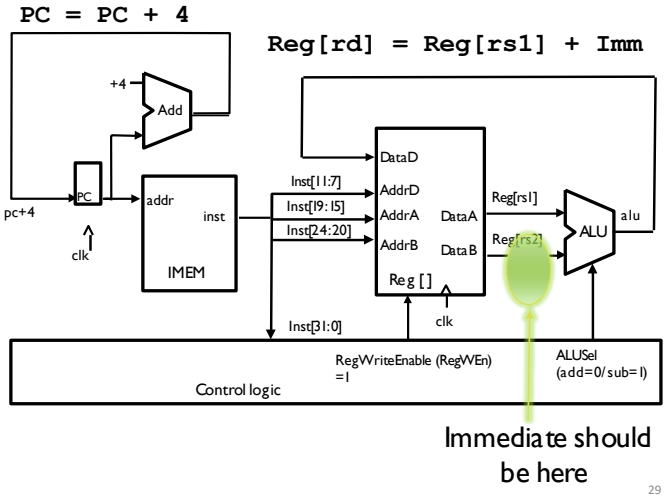
Implementing I-Format - addi instruction

- RISC-V Assembly Instruction:

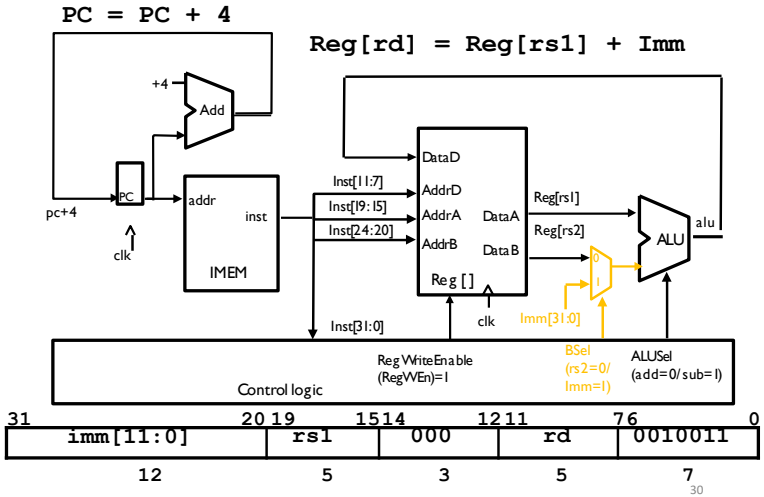
```
addi   x15,x1,-50
```



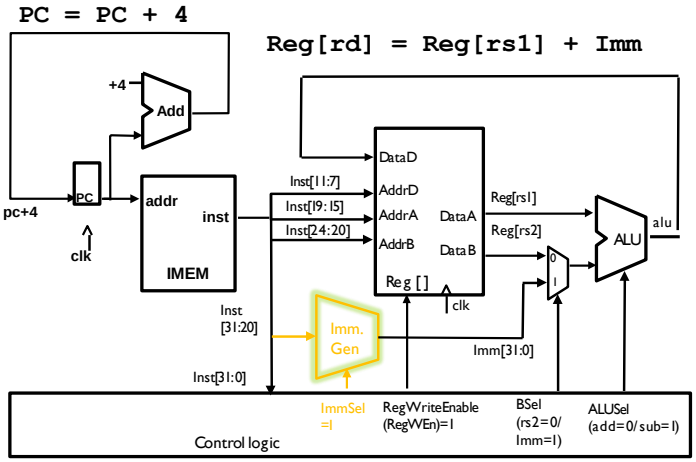
Datapath for add/sub



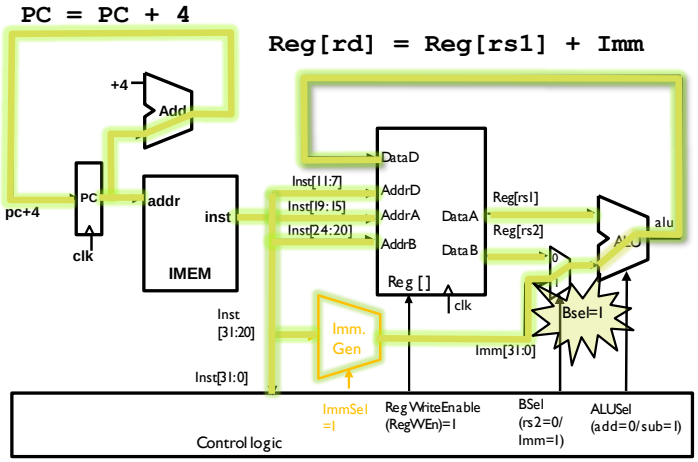
Adding **addi** to Datapath



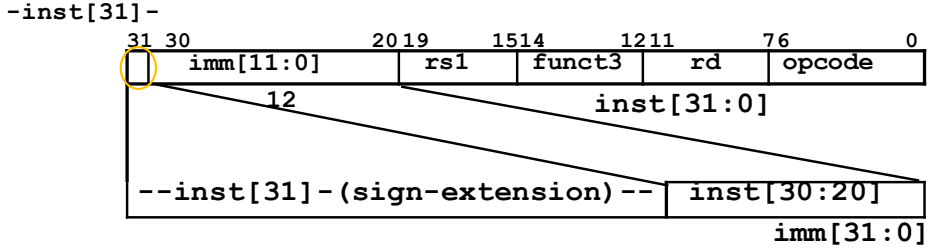
Adding **addi** to Datapath

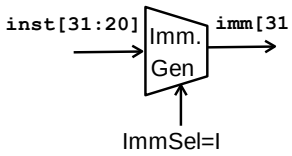


Adding **addi** to Datapath



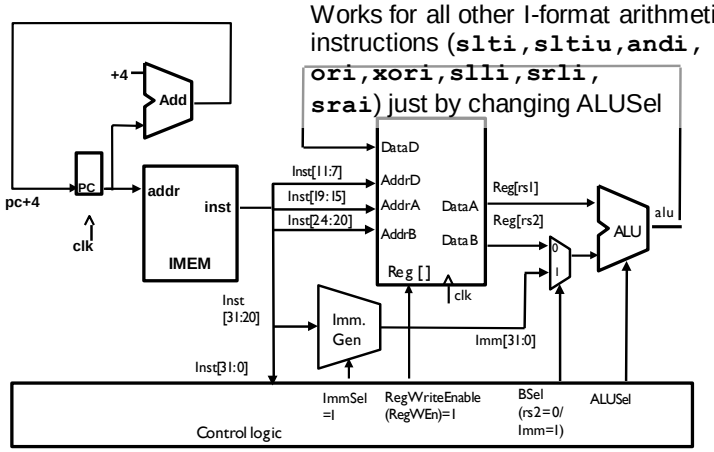
I-Format Immediates





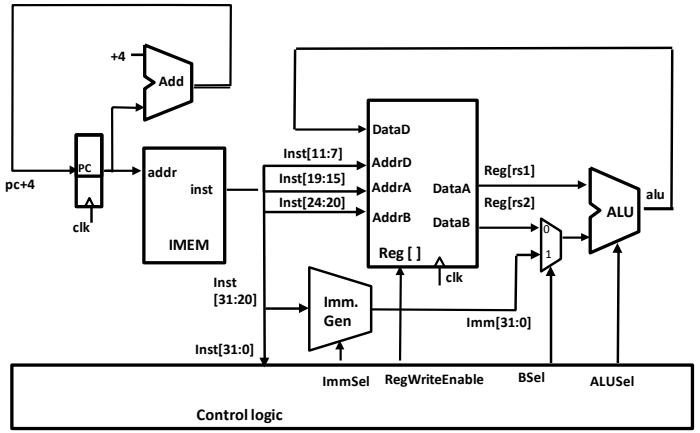
- High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
- Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])

Adding **addi** to Datapath



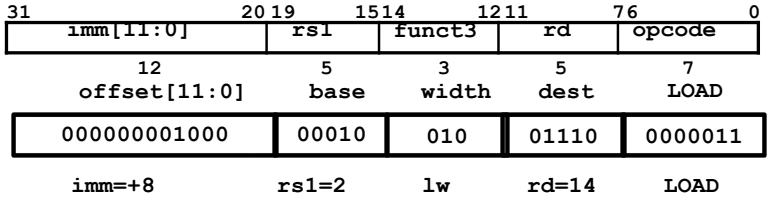
Supporting Loads

R+I Arithmetic/Logic Datapath



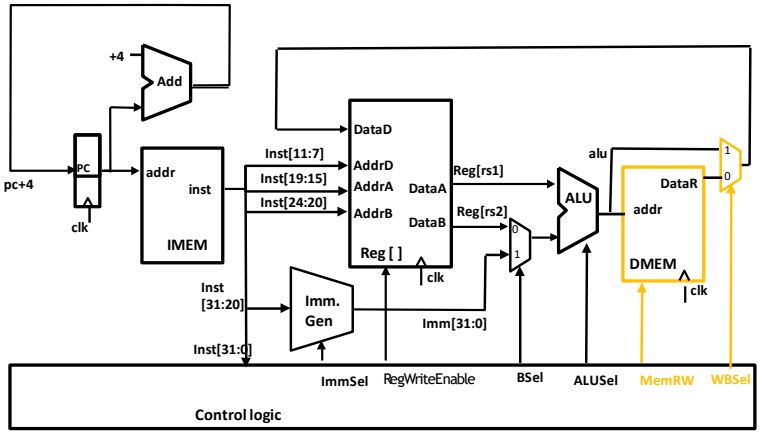
Add `lw`

- RISC-V Assembly Instruction (I-type): **`lw x14, 8(x2)`**

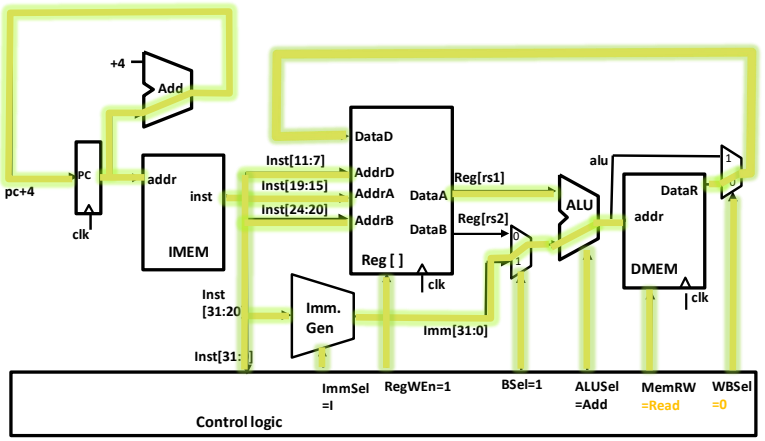


- The 12-bit signed immediate is added to the base address in register **`rs1`** to form the memory address
  - This is very similar to the add-immediate operation but used to create address not to create final result
- The value loaded from memory is stored in register **`rd`**

R+I Arithmetic/Logic Datapath



R+I Arithmetic/Logic Datapath



All RV32 Load Instructions

imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	001	rd	0000011	lh
imm[11:0]	rs1	010	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu

- funct3 field encodes size and 'signedness' of load data

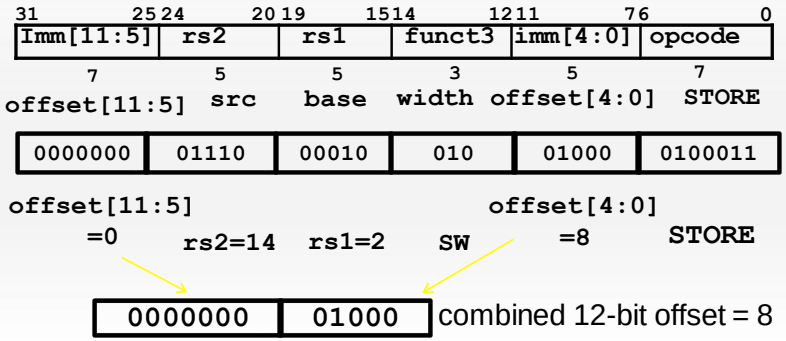
- Supporting the narrower loads requires additional logic to extract the correct byte/halfword from the value loaded from memory, and sign- or zero-extend the result to 32 bits before writing back to register file.
  - It is just a mux + a few gates

Datapath for Stores

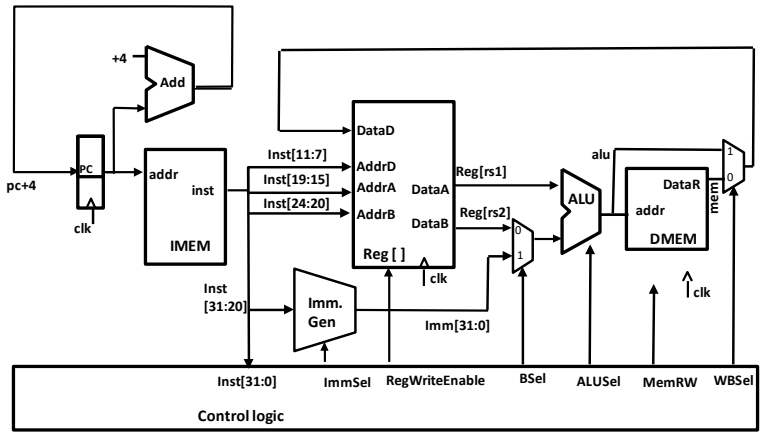
Adding **sw** instruction

sw: Reads two registers, rs1 for base memory address, and rs2 for data to be stored, as well immediate offset!

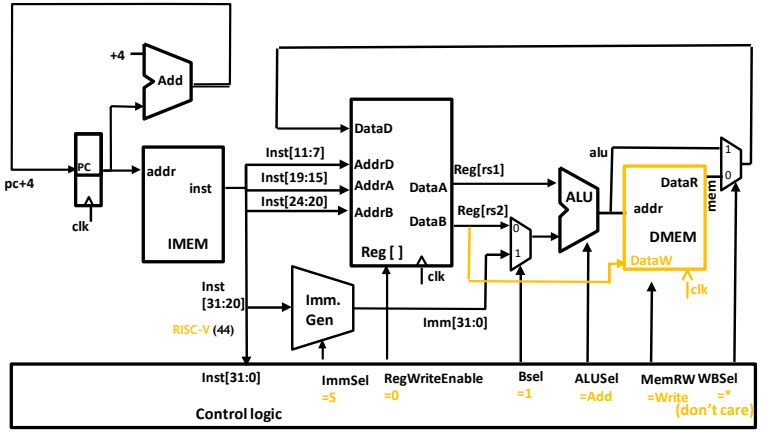
```
sw x14, 8(x2)
```



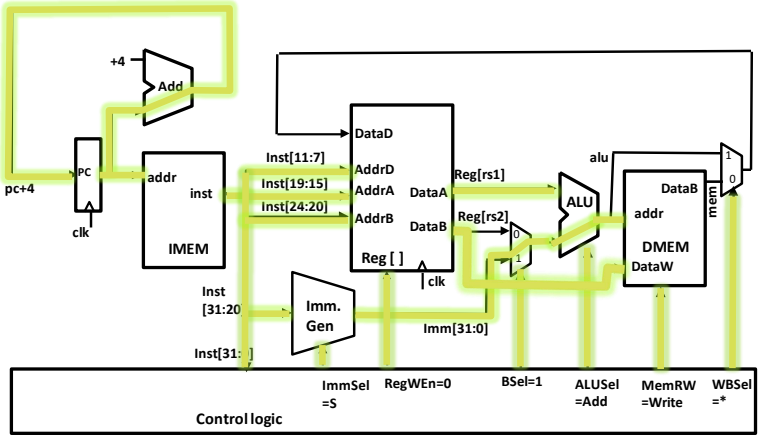
Datapath with **lw**



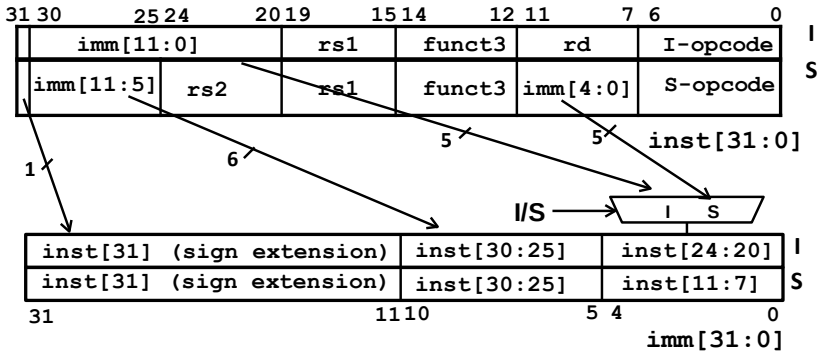
Adding `sw` to Datapath



Adding `sw` to Datapath



I+S Immediate Generation



- Just need a 5-bit mux to select between two positions where low five bits of immediate can reside in instruction
- Other bits in immediate are wired to fixed positions in instruction

All RV32 Store Instructions

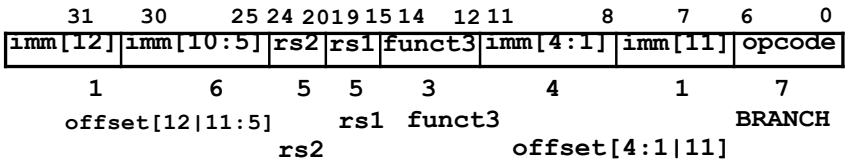
- Store byte writes the low byte to memory
- Store halfword writes the lower two bytes to memory

Imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	sb
Imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	sh
Imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw

width

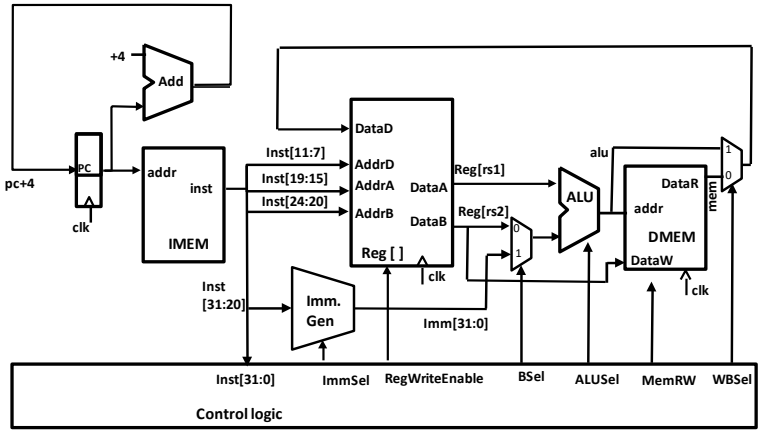
Implementing Branches

RISC-V B-Format for Branches



- B-format is mostly same as S-Format, with two register sources (**`rs1`**/**`rs2`**) and a 12-bit immediate **`imm[12:1]`**
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

Datapath So Far



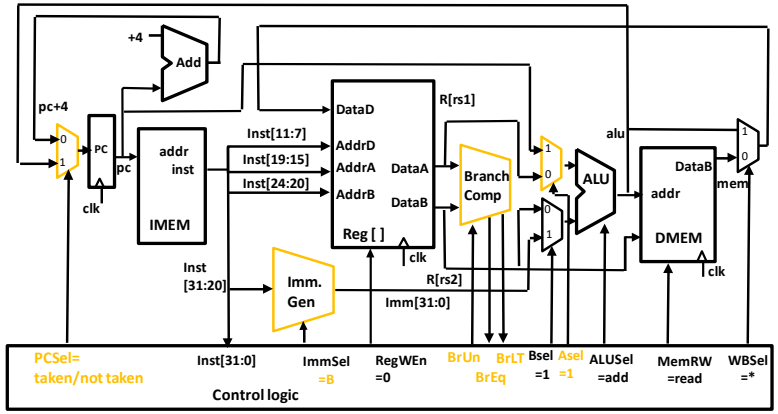
To Add Branches

- Different change to the state:

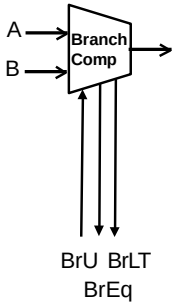
$$\texttt{PC} = \begin{cases} \texttt{PC + 4}, & \texttt{branch not taken} \\ \texttt{PC + immediate}, & \texttt{branch taken} \end{cases}$$

- Six branch instructions: **`beq, bne, blt, bge, bltu, bgeu`**
- Need to compute **`PC + immediate`** and to compare values of **`rs1`** and **`rs2`**
  - But have only one ALU – need more hardware

Adding Branches



Branch Comparator



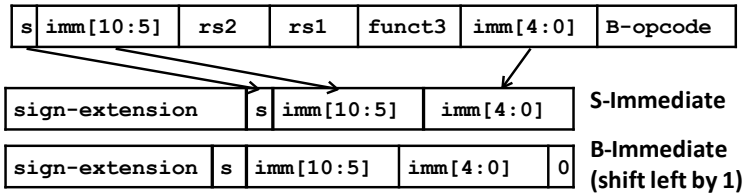
- **`BrEq = 1`**, if A=B
- **`BrLT = 1`**, if A<B
- **`BrUn`** = 1 selects unsigned comparison for **`BrLT`**, 0=signed

`BGE` branch: A >= B, if $\overline{A<B}$

$\overline{A<B}$ = !(A<B)

Branch Immediates (In Other ISAs)

- 12-bit immediate encodes PC-relative offset of -4096 to +4094 bytes in multiples of 2 bytes
- Standard approach: Treat immediate as in range -2048..+2047, then shift left by 1 bit to multiply by 2 for branches



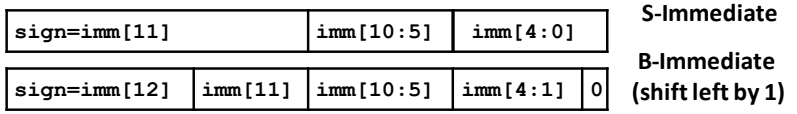
Each instruction immediate bit can appear in one of two places in output immediate value – so need one 2-way mux per bit

Branch Immediates (In RISC-V ISAs)

- 12-bit immediate encodes PC-relative offset of -4096 to +4094 bytes in multiples of 2 bytes

NOTE : Page. 116 (RISC V Textbook -> get extra information)

- RISC-V approach: keep 11 immediate bits in fixed position in output value, **and rotate LSB of S-format to be bit 12 of B-format**

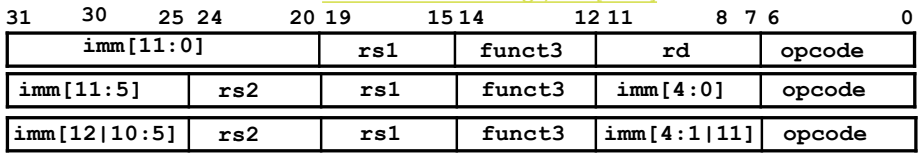


Only one bit changes position between S and B, so only need a single-bit 2-way mux

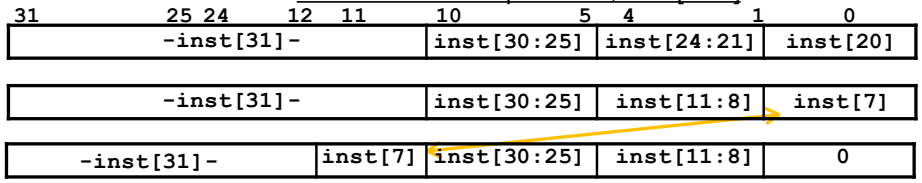
The RISC-V architects wanted to support the possibility of instructions that are only 2 bytes long, so the branch instructions represent the number of *halfwords* between the branch and the branch target

RISC-V Immediate Encoding

Instruction encodings, inst[31:0]



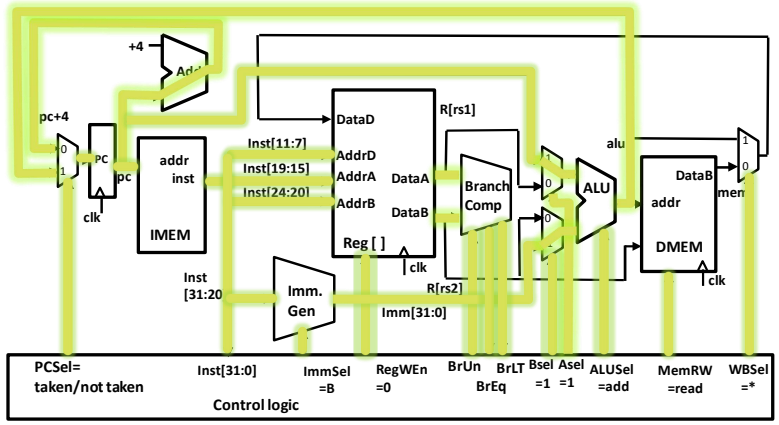
32-bit immediates produced, imm[31:0]



Upper bits sign-extended from `inst[31]` always

Only bit 7 of instruction changes role in immediate between S and B

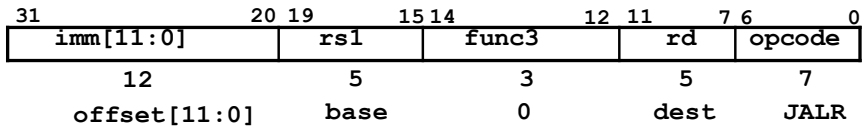
Lighting Up Branch Path



RISC-V (58)

Adding JALR to Datapath

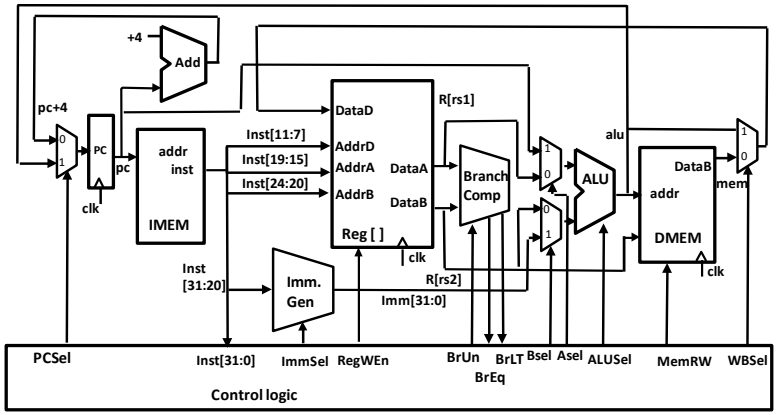
Let's Add **JALR** (I-Format)



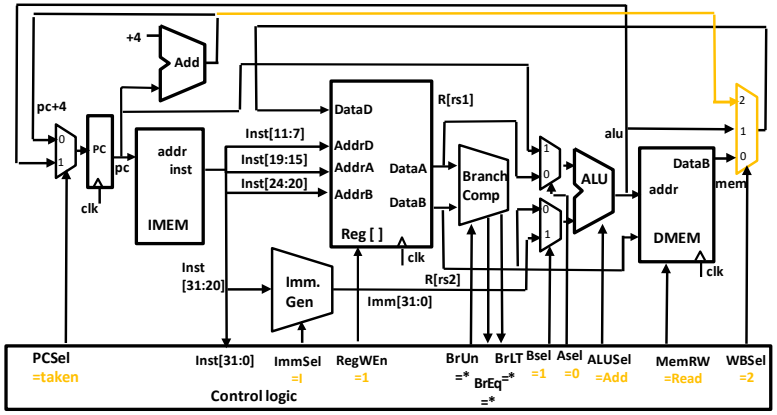
- **`JALR rd, rs, immediate`**
- Two changes to the state
  - Writes PC+4 to **`rd`** (return address)
  - Sets PC = **`rs1 + immediate`**
  - Uses same immediates as arithmetic and loads
    - ***no*** multiplication by 2 bytes
    - LSB is ignored

RISC-V (60)

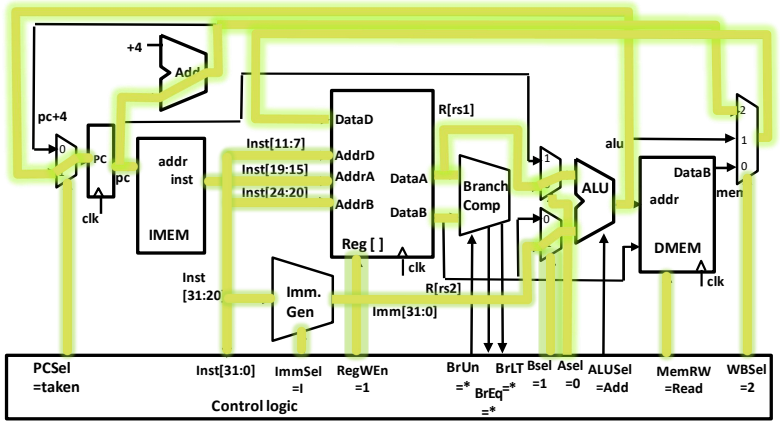
Datapath So Far, With Branches



Datapath With **JALR**

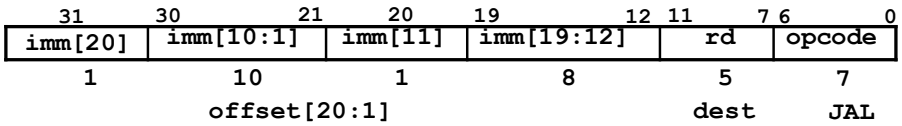


Datapath With **JALR**



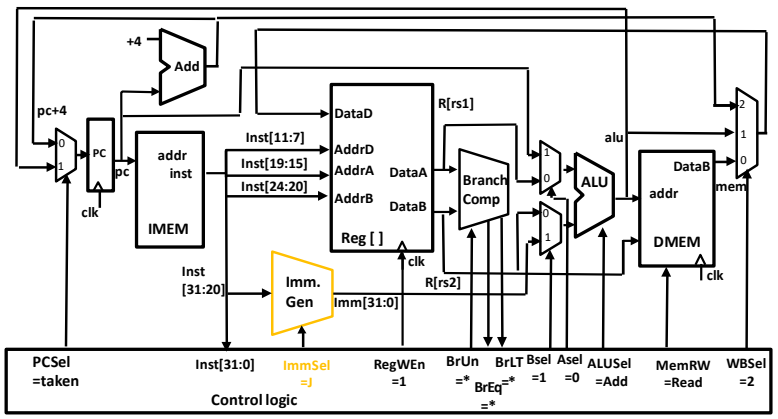
Adding JAL

J-Format for Jump Instructions

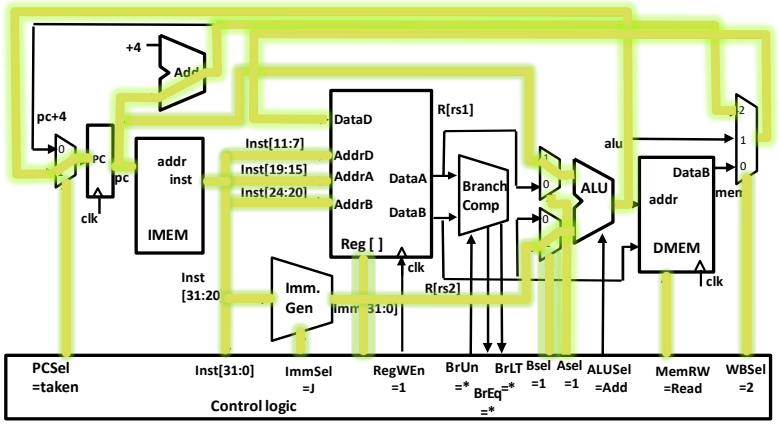


- Two changes to the state
  - **`jal`** saves PC+4 in register **`rd`** (the return address)
  - Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

Datapath with **`JAL`**

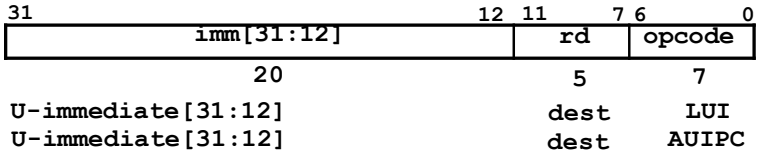


Light Up **`JAL`** Path



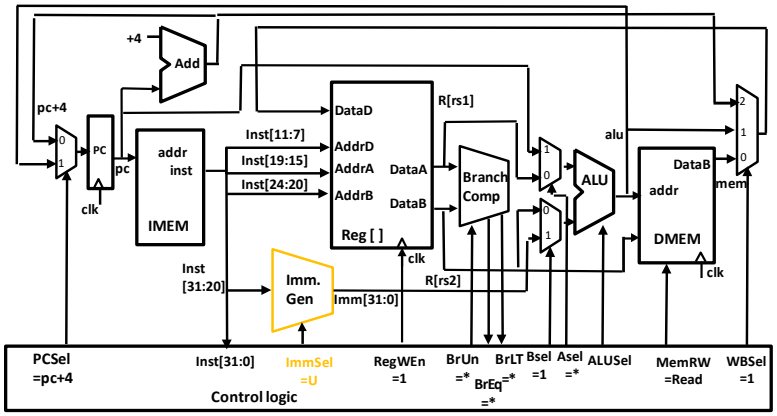
Adding U-Types

U-Format for "Upper Immediate" Instructions

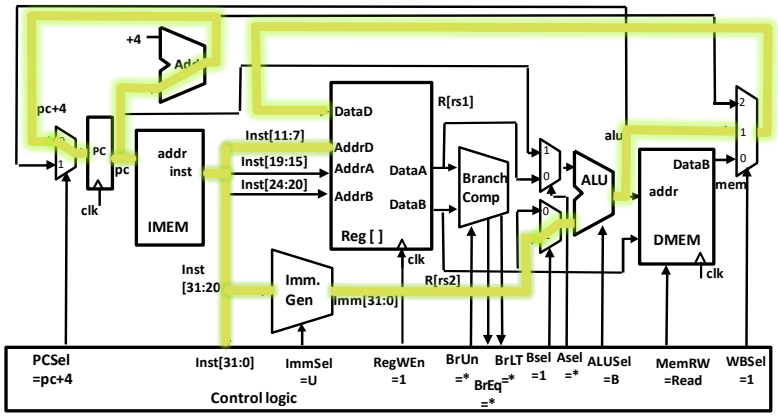


- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, **`rd`**
- Used for two instructions
  - **`lui`** – Load Upper Immediate
  - **`auipc`** – Add Upper Immediate to PC

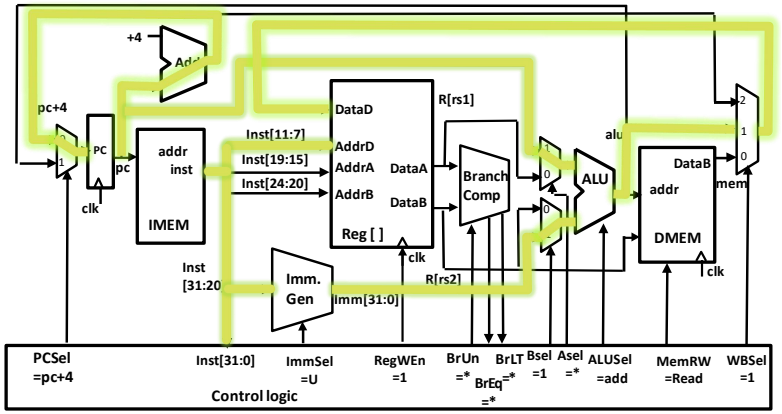
Datapath With `LUI`, `AUIPC`



Lighting Up `LUI`

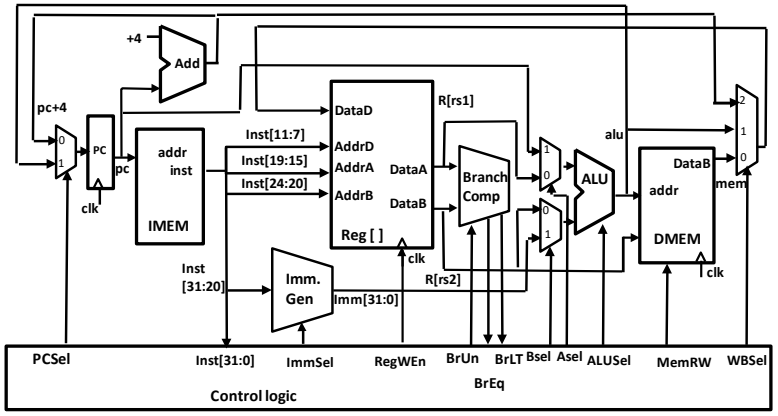


Lighting Up `AUIPC`



"And In Conclusion…"

Complete RV32I Datapath!



Complete RV32I ISA!

Open RISC-V Reference Card

Base Integer Instructions: RV32I

Category	Name	Fmt	RV32I Base	Category	Name	Fmt	RV32I Base
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2	Loads	Load Byte	I	LB rd,rs1,imm
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt		Load Halfword	I	LH rd,rs1,imm
	Shift Right Logical	R	SRL rd,rs1,rs2		Load Byte Unsigned	I	LBU rd,rs1,imm
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt		Load Half Unsigned	I	LHU rd,rs1,imm
	Shift Right Arithmetic	R	SRA rd,rs1,rs2		Load Word	I	LW rd,rs1,imm
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	Stores	Store Byte	S	SB rs1,rs2,imm
Arithmetic	ADD	R	ADD rd,rs1,rs2		Store Halfword	S	SH rs1,rs2,imm
	ADD Immediate	I	ADDI rd,rs1,imm		Store Word	S	SW rs1,rs2,imm
	SUBtract	R	SUB rd,rs1,rs2	Branches	Branch =	B	BEQ rs1,rs2,imm
	Load Upper Imm	U	LUI rd,imm		Branch ≠	B	BNE rs1,rs2,imm
	Add Upper Imm to PC	U	AUIPC rd,imm		Branch <	B	BLT rs1,rs2,imm
Logical	XOR	R	XOR rd,rs1,rs2		Branch ≥	B	BGE rs1,rs2,imm
	XOR Immediate	I	XORI rd,rs1,imm		Branch < Unsigned	B	BLTU rs1,rs2,imm
	OR	R	OR rd,rs1,rs2		Branch ≥ Unsigned	B	BGEU rs1,rs2,imm
	OR Immediate	I	ORI rd,rs1,imm	Jump & Link	JAL	J	JAL rd,imm
	AND	R	AND rd,rs1,rs2		Jump & Link Register	I	JALR rd,rs1,imm
	AND Immediate	I	ANDI rd,rs1,imm				
Compare	Set <	R	SLT rd,rs1,rs2	Synch	Synch thread	I	FENCE
	Set < Immediate	I	SLTI rd,rs1,imm				
	Set < Unsigned	R	SLTU rd,rs1,rs2	Environment	CALL	I	ECALL
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm		BREAK	I	EBREAK

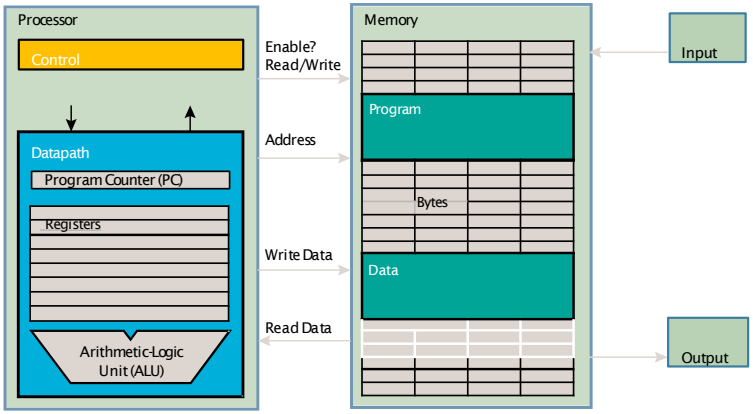
Not in 61C

Review

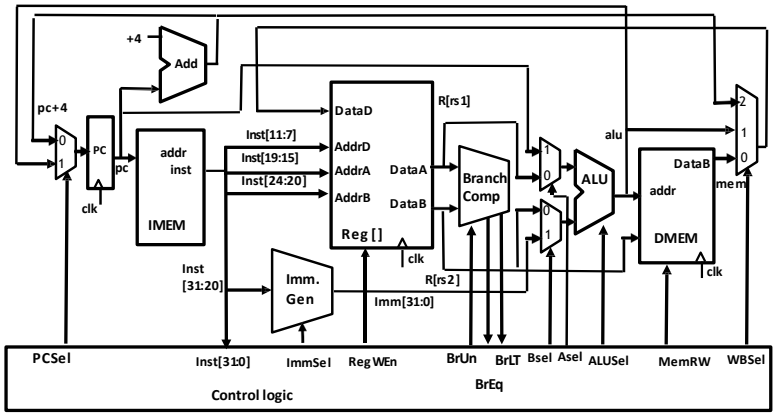
- We have designed a complete datapath
  - Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions
- 5 Phases of execution
  - IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases
- Controller specifies how to execute instructions
  - We still need to design it

Datapath Control

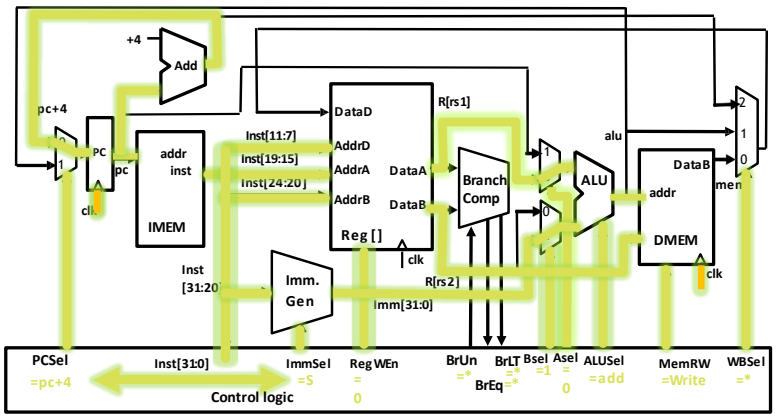
Our Single-Core Processor



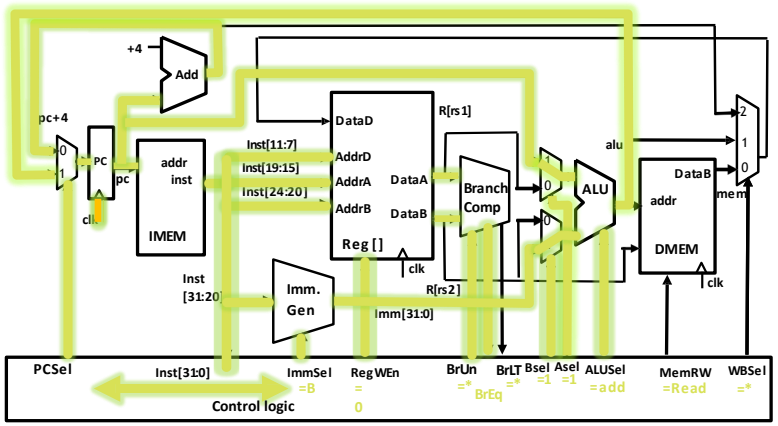
Single-Cycle RV32I Datapath and Control



Example: `sw`

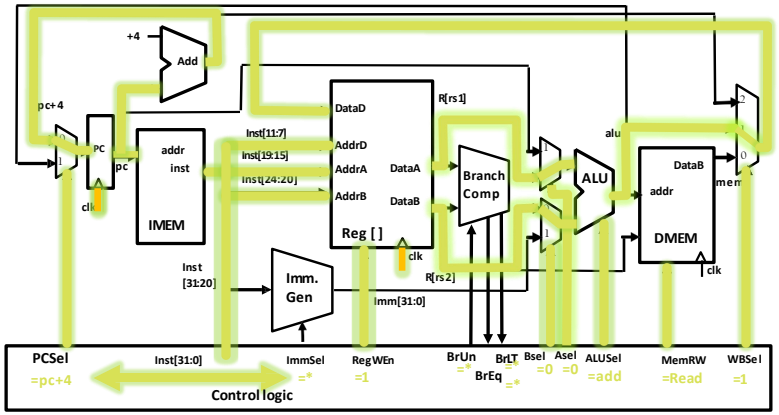


Example: **beq**

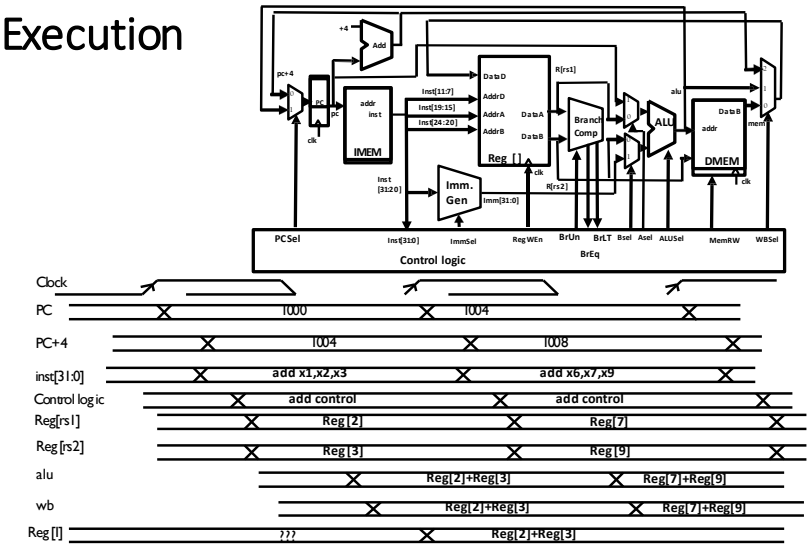


Instruction Timing

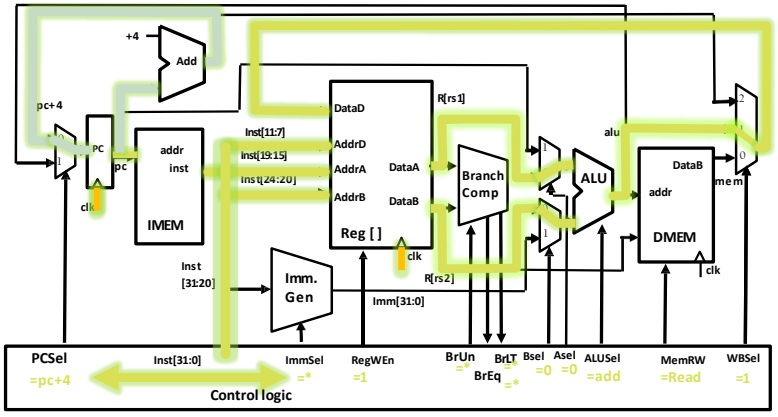
Example: **add**



Add Execution



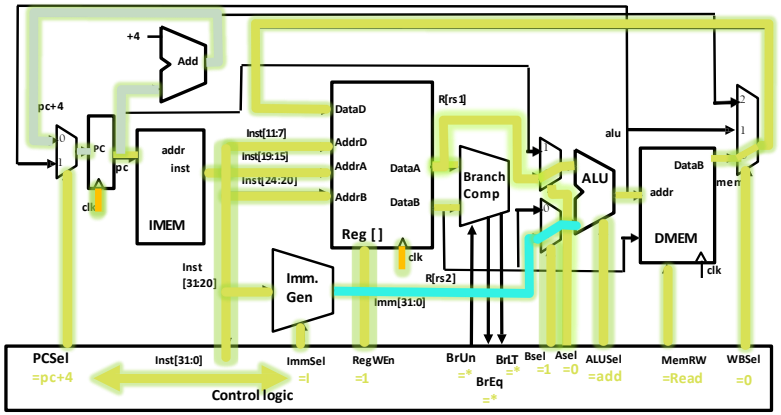
Example: **add** timing



Critical path = $t_{clk\text{-}q} + \max\{t_{Add}+t_{mux},\ t_{IMEM}+t_{Reg}+t_{mux}+t_{ALU}+t_{mux}\}+t_{setup}$

$= t_{clk\text{-}q} + t_{IMEM}+t_{Reg}+t_{mux}+t_{ALU}+t_{mux}+t_{setup}$

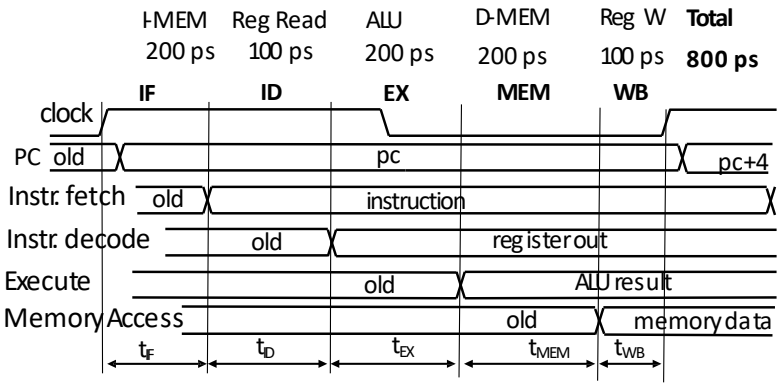
Example: **lw**



Critical path = $t_{clk\text{-}q} + \max\{t_{Add}+t_{mux},\ t_{IMEM}+t_{Imm}+t_{mux}+t_{ALU}+t_{DMEM}+t_{mux},$

$t_{IMEM}+t_{Reg}+t_{mux}+t_{ALU}+t_{DMEM}+t_{mux}\}+t_{setup}$

Instruction Timing



Instruction Timing

Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X		X	600ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

- Maximum clock frequency
  - f_{max} = 1/800ps = 1.25 GHz
- Most blocks idle most of the time
  - E.g. $f_{max,ALU}$ = 1/200ps = 5 GHz!

Control Logic Design

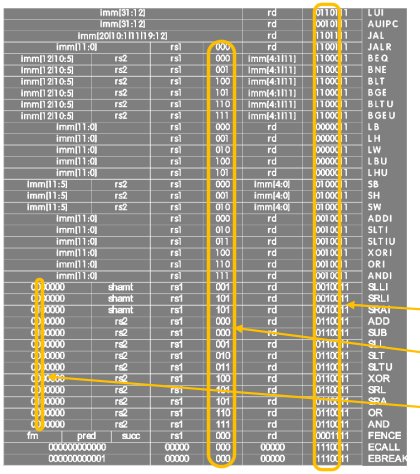
Control Logic Truth Table

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	+4	*	*	Reg	Reg	Add	Read	1	ALU
sub	*	*	+4	*	*	Reg	Reg	Sub	Read	1	ALU
(R-R Op)	*	*	+4	*	*	Reg	Reg	(Op)	Read	1	ALU
addi	*	*	+4	I	*	Reg	Imm	Add	Read	1	ALU
lw	*	*	+4	I	*	Reg	Imm	Add	Read	1	Mem
sw	*	*	+4	S	*	Reg	Imm	Add	Write	0	*
beq	0	*	+4	B	*	PC	Imm	Add	Read	0	*
beq	1	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	0	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	1	*	+4	B	*	PC	Imm	Add	Read	0	*
blt	*	1	ALU	B	0	PC	Imm	Add	Read	0	*
bltu	*	1	ALU	B	1	PC	Imm	Add	Read	0	*
jalr	*	*	ALU	I	*	Reg	Imm	Add	Read	1	PC+4
jal	*	*	ALU	J	*	PC	Imm	Add	Read	1	PC+4
auipc	*	*	+4	U	*	PC	Imm	Add	Read	1	ALU

Control Realization Options

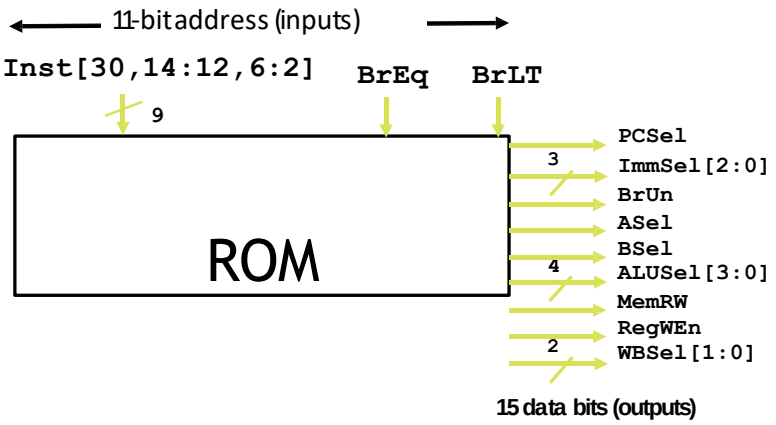
- ROM
  - "Read-Only Memory"
  - Regular structure
  - Can be easily reprogrammed
    - fix errors
    - add instructions
  - Popular when designing control logic manually
- Combinatorial Logic
  - Today, chip designers use logic synthesis tools to convert truth tables to networks of gates

RV32I, A Nine-Bit ISA!

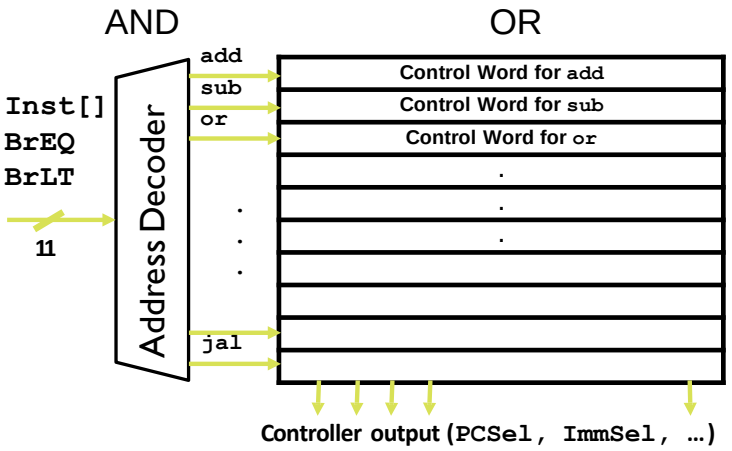


- Instruction type encoded using only 9 bits:
- `inst[30]`, `inst[14:12]`, `inst[6:2]`

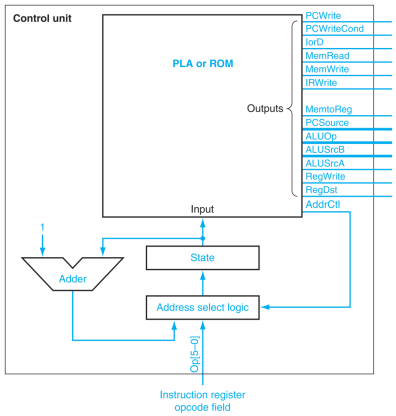
ROM-based Control

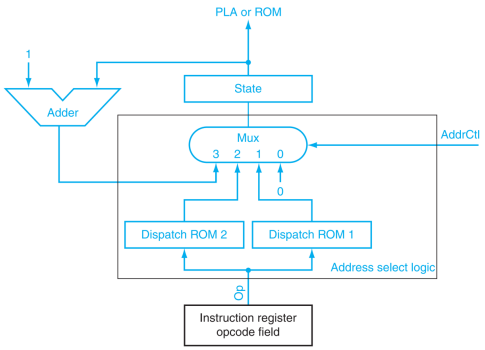


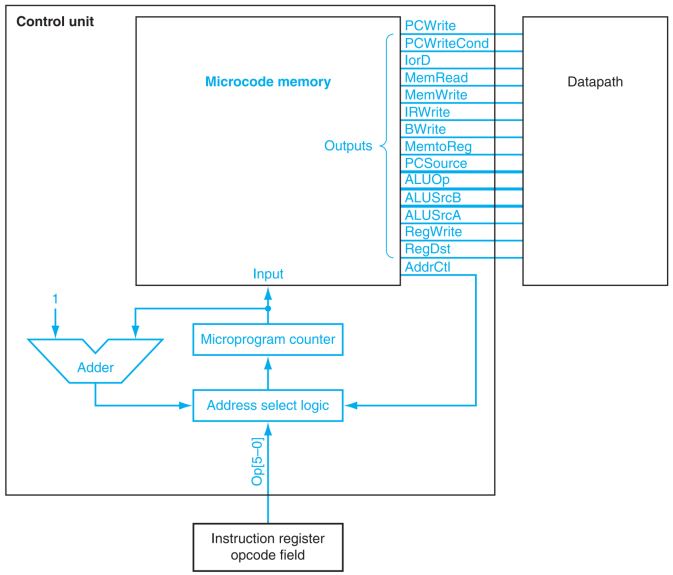
ROM Controller Implementation



The control unit using an explicit counter to compute the next state



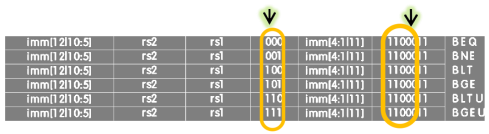




Combinational Logic Control

- Simplest example: **`BrUn`**

inst[14:12] inst[6:2]



imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

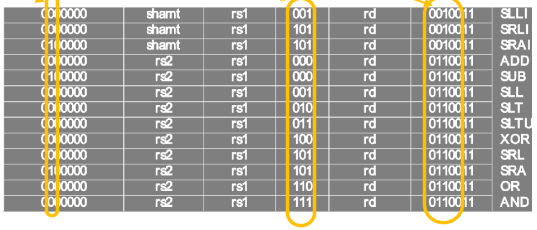
- How to decode whether BrUn is 1?

`BrUn = Inst [13] • Branch`

Control Logic to Decode add

`add = i[30]•i[14]•i[13]•i[12]•R-type`

inst[30] inst[14:12] inst[6:2]



0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

`R-type = i[6]•i[5]•i[4]•i[3]•i[2]•RV32I`

`RV32I = i[1]•i[0]`

"And In Conclusion…"

Call home, we've made HW/SW contact!

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

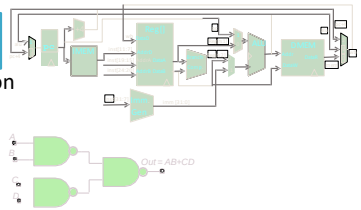
```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

Assembler

Machine Language Program (RISC-V)

```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

"And In conclusion…"

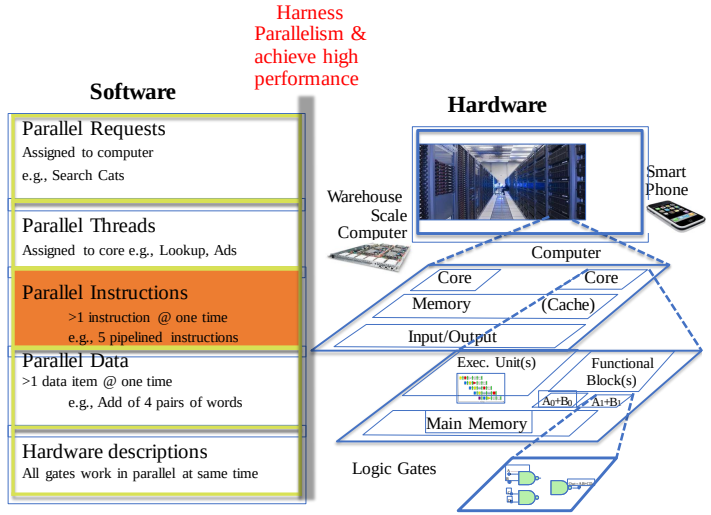
- We have built a processor!
  - Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions
  - Critical path changes
- 5 Phases of execution
  - IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases
- Controller specifies how to execute instructions
  - Implemented as ROM or logic

Computer Architecture 1

Computer Organization and Design
THE HARDWARE/SOFTWARE INTERFACE

[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolić, © 2020, UC Berkeley]

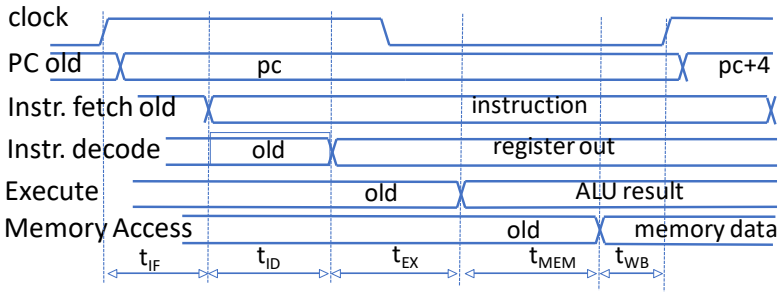
New-School Machine Structures



6 Great Ideas in Computer Architecture

1. Abstraction (Layers of Representation/Interpretation)
2. Moore's Law
3. Principle of Locality/Memory Hierarchy
4. **Parallelism**
5. Performance Measurement & Improvement
6. Dependability via Redundancy

Instruction Timing



IF	ID	EX	MEM	WB	Total
I-MEM	Reg Read	ALU	D-MEM	Reg W	
200 ps	100 ps	200 ps	200 ps	100 ps	**800 ps**

Instruction Timing

Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X		X	600ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

- Maximum clock frequency
  - f_{max} = 1/800ps = 1.25 GHz

Performance Measures

- "Our" Single-cycle RISC-V CPU executes instructions at 1.25 GHz
  - 1 instruction every 800 ps
- Can we improve its performance?
  - What do we mean with this statement?
  - Not so obvious:
    - Quicker response time, so one job finishes faster?
    - More jobs per unit time (e.g. web server returning pages, spoken words recognized)?
    - Longer battery life?

Transportation Analogy

	Sports Car	Bus
Passenger Capacity	2	50
Travel Speed	200 mph	50 mph
Gas Mileage	5 mpg	2 mpg

50 Mile trip (assume they return instantaneously)

	Sports Car	Bus
Travel Time	15 min	60 min
Time for 100 passengers	750 min (50 2-person trips)	120 min (two 50-person trips)
Gallons per passenger	5 gallons	0.5 gallons

Computer Analogy

Transportation	Computer
Trip Time	Program execution time: e.g. time to update display
Time for 100 passengers	Throughput: e.g. number of server requests handled per hour
Gallons per passenger	Energy per task*: e.g. how many movies you can watch per battery charge or energy bill for datacenter

* Note: Power is not a good measure, since low-power CPU might run for a long time to complete one task consuming more energy than faster computer running at higher power for a shorter time

Processor Performance Iron Law

"Iron Law" of Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

CPI = Cycles Per Instruction

Defining (Speed) Performance

- Normally interested in reducing
  - Response time (aka execution time) – the time between the start and the completion of a task
    - Important to individual users
  - Thus, to maximize performance, need to minimize execution time

$$\text{performance}_X = 1 / \text{execution\_time}_X$$

If X is n times faster than Y, then

$$\frac{\text{performance}_X}{\text{performance}_Y} = \frac{\text{execution\_time}_Y}{\text{execution\_time}_X} = n$$

- Throughput – the total amount of work done in a given time
  - Important to data center managers
- Decreasing response time almost always improves throughput

Performance Factors

- Want to distinguish elapsed time and the time spent on our task
- CPU execution time (CPU time) – time the CPU spends working on a task
  - Does not include time waiting for I/O or running other programs

$$\text{CPU execution time for a program} = \text{\# CPU clock cycles for a program} \times \text{clock cycle time}$$

or

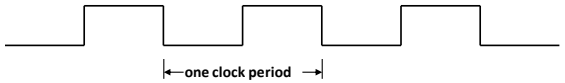
$$\text{CPU execution time for a program} = \frac{\text{\# CPU clock cycles for a program}}{\text{clock rate}}$$

- Can improve performance by reducing either the length of the clock cycle or the number of clock cycles required for a program

Review: Machine Clock Rate

- Clock rate (MHz, GHz) is inverse of clock cycle time (clock period)

$$CC = 1 / CR$$



10 nsec clock cycle => 100 MHz clock rate
5 nsec clock cycle => 200 MHz clock rate
2 nsec clock cycle => 500 MHz clock rate
1 nsec clock cycle => 1 GHz clock rate
500 psec clock cycle => 2 GHz clock rate
250 psec clock cycle => 4 GHz clock rate
200 psec clock cycle => 5 GHz clock rate

Clock Cycles per Instruction

- Not all instructions take the same amount of time to execute
  - One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction

$$\text{\# CPU clock cycles for a program} = \text{\# Instructions for a program} \times \text{Average clock cycles per instruction}$$

- Clock cycles per instruction (CPI) – the average number of clock cycles each instruction takes to execute
  - A way to compare two different implementations of the same ISA

	CPI for this instruction class		
	A	B	C
CPI	1	2	3

Effective CPI

- Computing the overall effective CPI is done by looking at the different types of instructions and their individual cycle counts and averaging

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (CPI_i \times IC_i)$$

  - Where IC_i is the count (percentage) of the number of instructions of class i executed
  - CPI_i is the (average) number of clock cycles per instruction for that instruction class
  - n is the number of instruction classes

- The overall effective CPI varies by instruction mix – a measure of the dynamic frequency of instructions across one or many programs

THE Performance Equation

- Our basic performance equation is then

$$\text{CPU time} = \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle}$$

or

$$\text{CPU time} = \frac{\text{Instruction\_count} \times \text{CPI}}{\text{clock\_rate}}$$

- These equations separate the three key factors that affect performance
  - Can measure the CPU execution time by running the program
  - The clock rate is usually given
  - Can measure overall instruction count by using profilers/ simulators without knowing all of the implementation details
  - CPI varies by instruction type and ISA implementation for which we must know the implementation details

A Simple Example

Op	Freq	CPI_i	Freq x CPI_i			
ALU	50%	1	.5	.5	.5	.25
Load	20%	5	1.0	.4	1.0	1.0
Store	10%	3	.3	.3	.3	.3
Branch	20%	2	.4	.4	.2	.4
			$\sum$ = 2.2	1.6	2.0	1.95

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

  CPU time new = 1.6 x IC x CC so 2.2/1.6 means 37.5% faster
- How does this compare with using branch prediction to shave a cycle off the branch time?

  CPU time new = 2.0 x IC x CC so 2.2/2.0 means 10% faster
- What if two ALU instructions could be executed at once?

  CPU time new = 1.95 x IC x CC so 2.2/1.95 means 12.8% faster

Instructions per Program

Determined by

- Task
- Algorithm, e.g. $O(N^2)$ vs $O(N)$
- Programming language
- Compiler
- Instruction Set Architecture (ISA)

(Average) Clock Cycles per Instruction (CPI)

Determined by

- ISA
- Processor implementation (or *microarchitecture*)
- E.g. for "our" single-cycle RISC-V design, CPI = 1
- Complex instructions (e.g. **`strcpy`**), CPI >> 1
- Superscalar processors, CPI < 1 (next lectures)

Time per Cycle (1/Frequency)

Determined by

- Processor microarchitecture (determines critical path through logic gates)
- Technology (e.g. 5nm versus 28nm)
- Power budget (lower voltages reduce transistor speed)

Speed Tradeoff Example

- For some task (e.g. image compression) …

	Processor A	Processor B
# Instructions	1 Million	1.5 Million
Average CPI	2.5	1
Clock rate f	2.5 GHz	2 GHz
Execution time	1 ms	0.75 ms

Processor B is faster for this task, despite executing more instructions and having a slower clock rate!

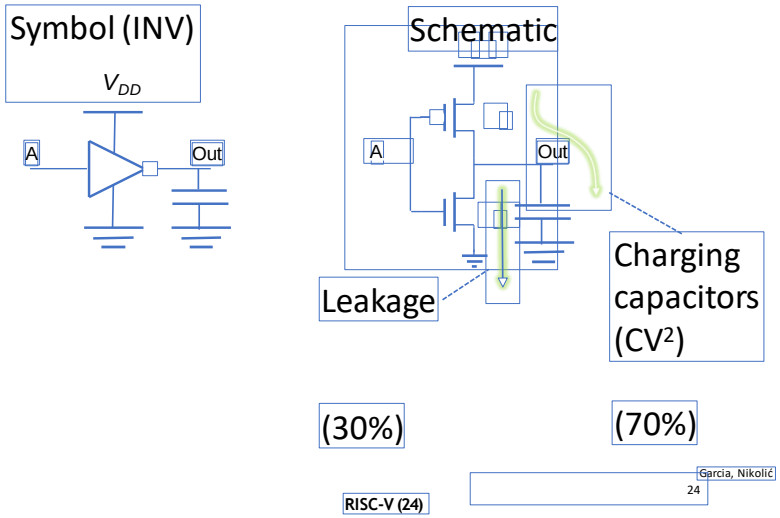
Determinates of CPU Performance

CPU time = Instruction_count x CPI x clock_cycle

	Instruction_ count	CPI	clock_cycle
Algorithm	x	x	
Programming language	x	x	
Compiler	x	x	
ISA	x	x	x
Processor organization		x	x
Technology			x

Energy Efficiency

Where Does Energy Go in CMOS?



Energy per Task

Energy / Program = Instructions / Program * Energy / Instruction

Energy / Program α Instructions / Program * C V^2

"Capacitance" depends on technology, processor features e.g. # of cores

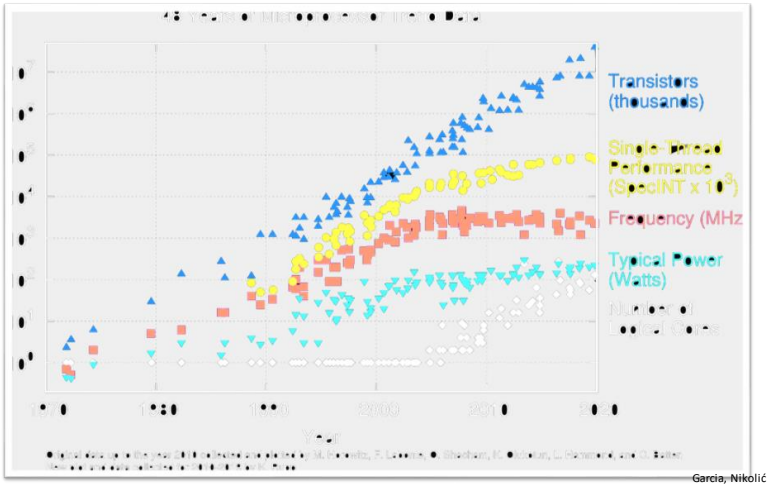
Supply voltage, e.g. 1V

Want to reduce capacitance and voltage to reduce energy/task

Energy Tradeoff Example

- "Next-generation" processor
  - C (Moore's Law): -15 %
  - Supply voltage, V_{sup}: -15 %
  - Energy consumption: 0 - (1-0.85^3) = -39 %
- Significantly improved energy efficiency thanks to
  - Moore's Law AND
  - Reduced supply voltage

Performance/Power Trends



End of Scaling

- In recent years, industry has not been able to reduce supply voltage much, as reducing it further would mean increasing "leakage power" where transistor switches don't fully turn off (more like dimmer switch than on-off switch)
- Also, size of transistors and hence capacitance, not shrinking as much as before between transistor generations
  - Need to go to 3D
- Power becomes a growing concern – the "power wall"

Energy "Iron Law"

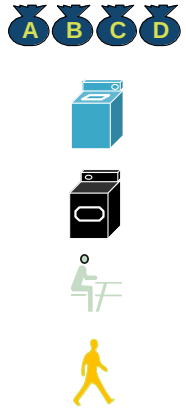
Performance = Power * Energy Efficiency
(Tasks/Second) (Joules/Second) (Tasks/Joule)

- Energy efficiency (e.g., instructions/Joule) is key metric in all computing devices
- For power-constrained systems (e.g., 20MW datacenter), need better energy efficiency to get more performance at same power
- For energy-constrained systems (e.g., 1W phone), need better energy efficiency to prolong battery life

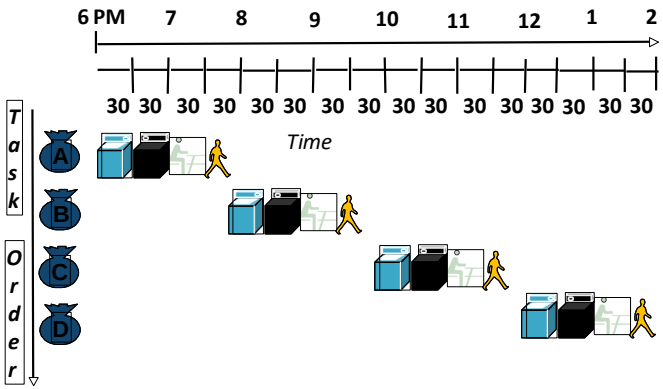
Introduction to Pipelining

Gotta Do Laundry

- Avi, Bora, Caroline, Dan each have one load of clothes to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - "Folder" takes 30 minutes
  - "Stasher" takes 30 minutes to put clothes into drawers

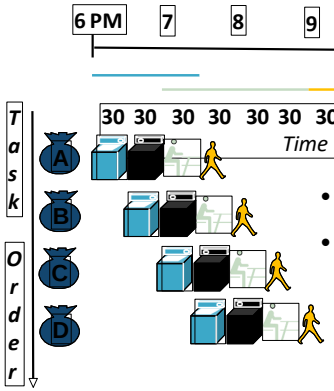


Sequential Laundry



Sequential laundry takes 8 hours for 4 loads!

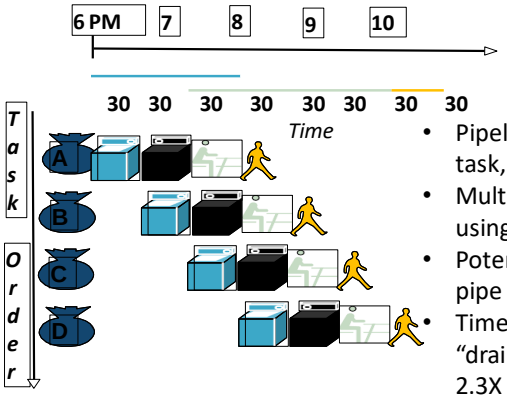
Pipelined Laundry



- What happens *sequentially*?
- What happens *simultaneously*?

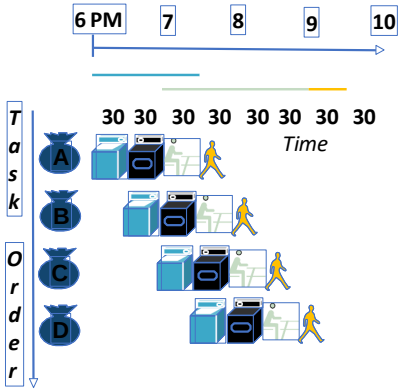
Pipelined laundry takes 3.5 hours for 4 loads!

Sequential Laundry



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number of pipe stages
- Time to "fill" pipeline and time to "drain" it reduces speedup: 2.3X v. 4X in this example

Sequential Laundry



Suppose:
- new Washer takes 20 minutes
- new Stasher takes 20 minutes.

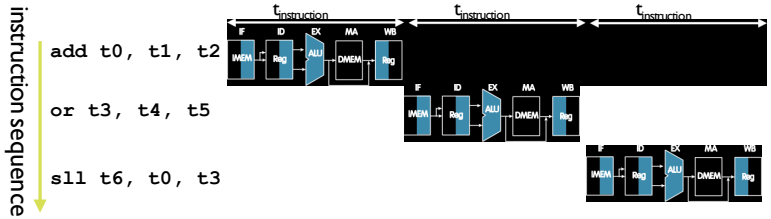
How much faster is pipeline?

Pipeline rate limited by slowest pipeline stage

Unbalanced lengths of pipe stages reduce speedup

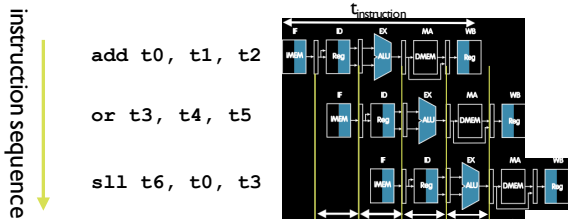
‘Sequential’ RISC-V Datapath

Phase	Pictogram	t_{step} Serial
Instruction Fetch		200 ps
Reg Read		100 ps
ALU		200 ps
Memory		200 ps
Register Write		100 ps
$t_{instruction}$		800 ps

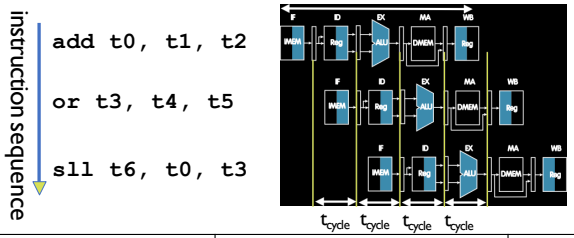


Pipelined RISC-V Datapath

Phase	Pictogram	t_{step} Serial	t_{cycle} Pipelined
Instruction Fetch		200 ps	200 ps
Reg Read		100 ps	200 ps
ALU		200 ps	200 ps
Memory		200 ps	200 ps
Register Write		100 ps	200 ps
$t_{instruction}$		800 ps	1000 ps



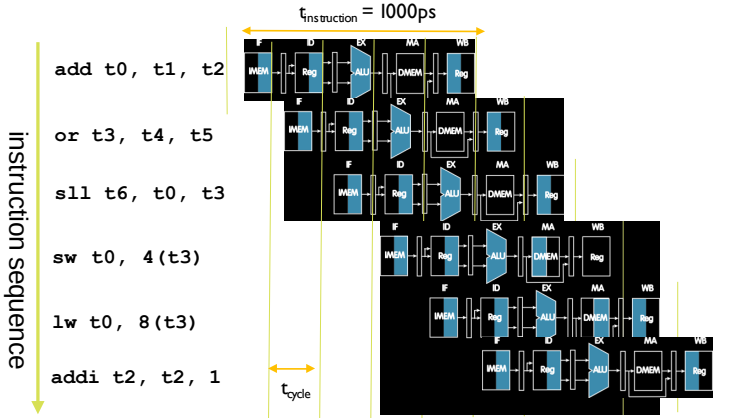
Pipelined RISC-V Datapath



	Single Cycle	Pipelined
Timing	t_{step} = 100 … 200 ps	t_{cycle} = 200 ps
	Register access only 100 ps	All cycles same length
Instruction time, $t_{instruction}$	= t_{cycle} = 800 ps	1000 ps
CPI (Cycles Per Instruction)	~1 (ideal)	~1 (ideal), <1 (actual)
Clock rate, f_s	1/800 ps = 1.25 GHz	1/200 ps = 5 GHz
Relative speed	1 x	4 x

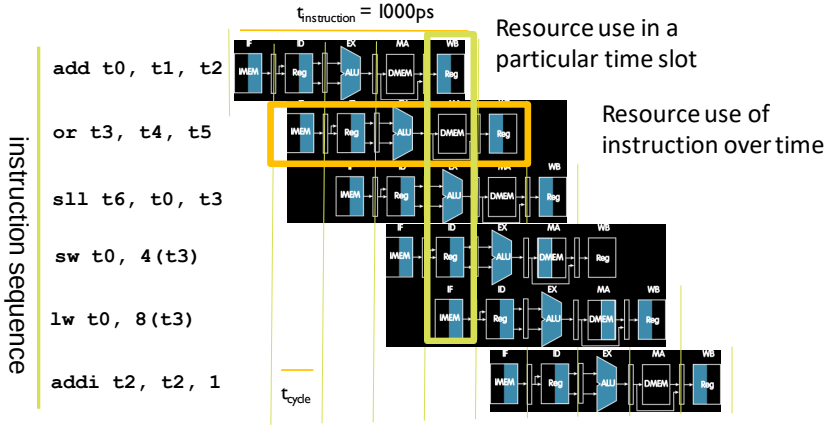
Sequential vs. Simultaneous

- What happens sequentially and what simultaneously?



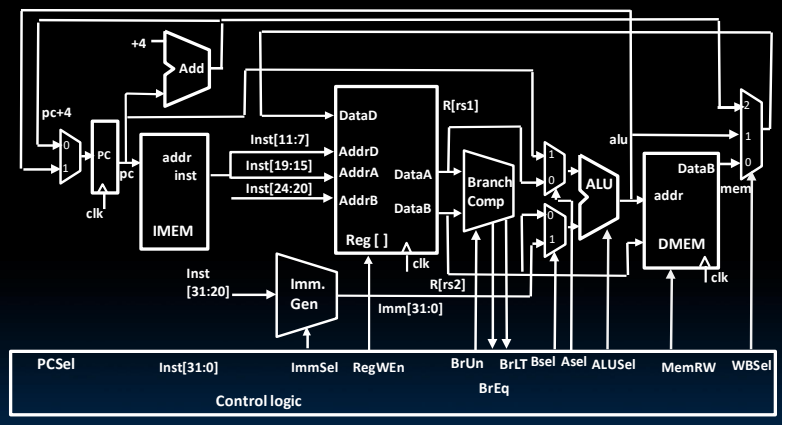
Sequential vs. Simultaneous

- What happens sequentially and what simultaneously?

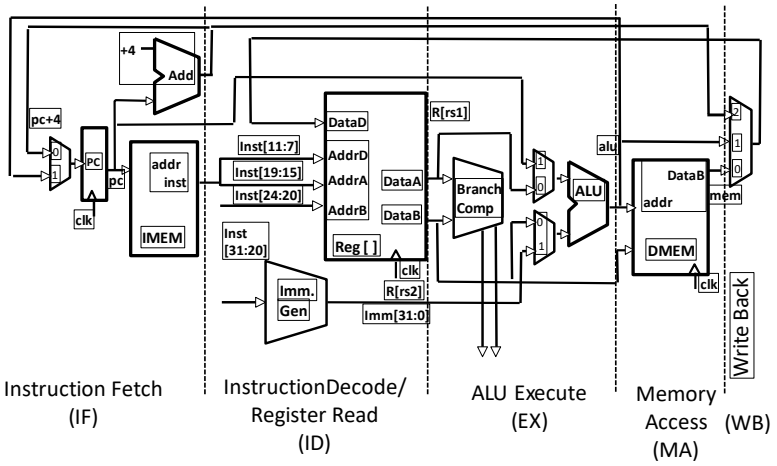


Pipelining Datapath

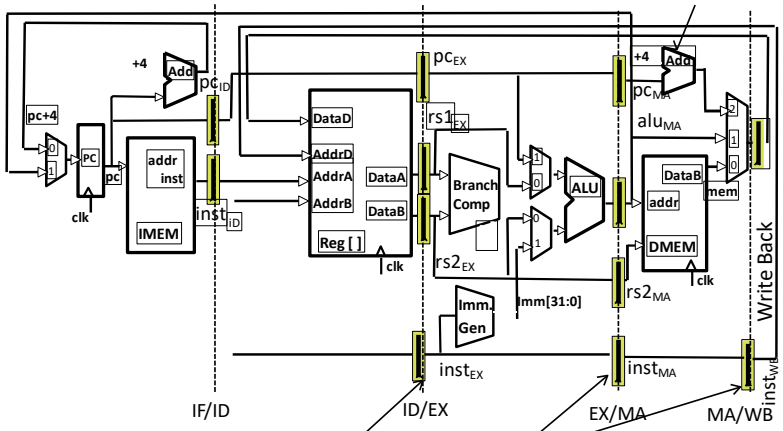
Single-Cycle

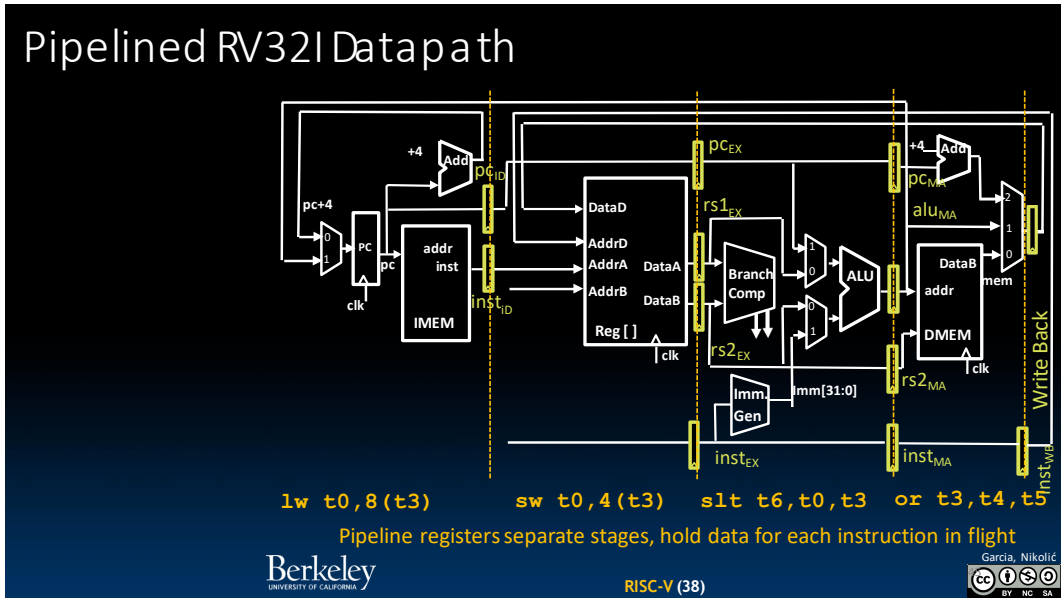


Single-Cycle RV32I Datapath



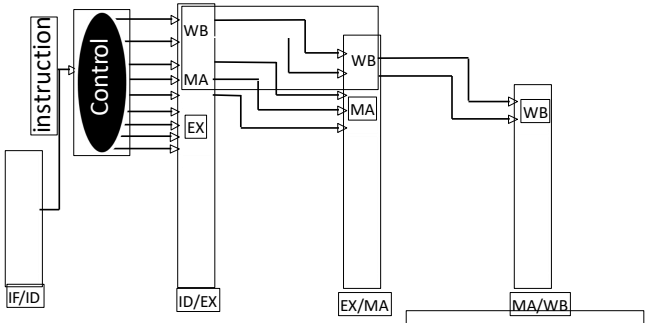
Pipelined RV32I Datapath





Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
  - Information is stored in pipeline registers for use by later stages



Pipeline Hazards

Hazards Ahead!

Pipelining Hazards

A *hazard* is a situation that prevents starting the next instruction in the next clock cycle

1) *Structural hazard*
   - A required resource is busy (e.g. needed in multiple stages)
2) *Data hazard*
   - Data dependency between instructions
   - Need to wait for previous instruction to complete its data read/write
3) *Control hazard*
   - Flow of execution depends on previous instruction

Structural Hazard

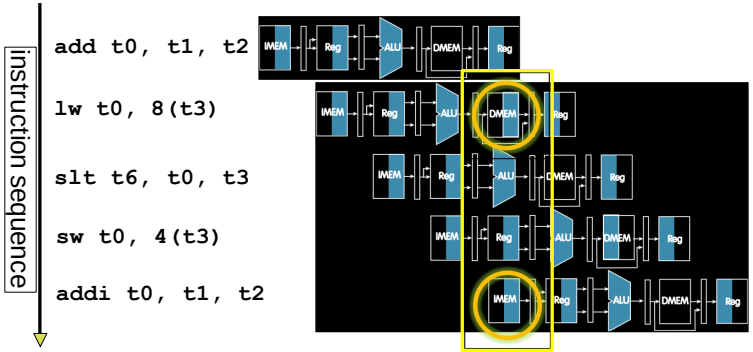
- **Problem:** Two or more instructions in the pipeline compete for access to a single physical resource
- **Solution 1:** Instructions take it in turns to use resource, some instructions have to stall
- **Solution 2:** Add more hardware to machine
- Can always solve a structural hazard by adding more hardware

Regfile Structural Hazards

- Each instruction:
  - Can read up to two operands in decode stage
  - Can write one value in writeback stage
- Avoid structural hazard by having separate "ports"
  - Two independent read ports and one independent write port
- Three accesses per cycle can happen simultaneously

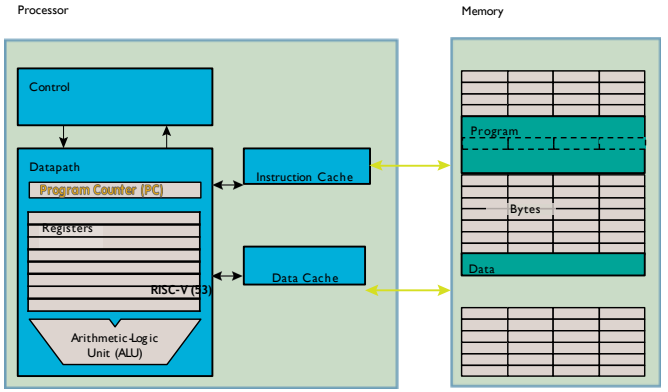
Structural Hazard: Memory Access

- Instruction and data memory used simultaneously
  - ✓ Use two separate memories



Instruction and Data Caches

- Fast, on-chip memory, separate for instructions and data



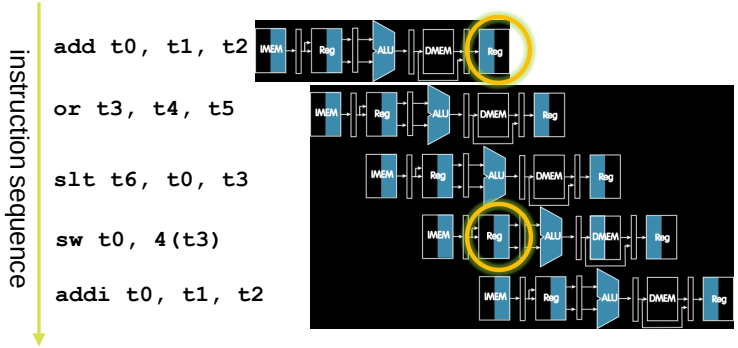
Structural Hazards – Summary

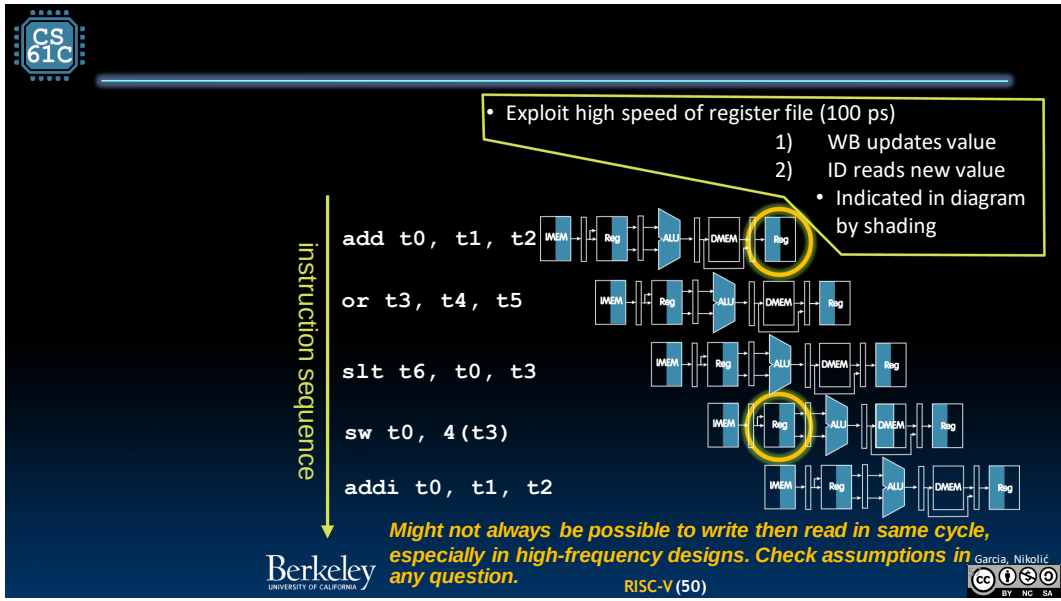
- Conflict for use of a resource
- In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Without separate memories, instruction fetch would have to stall for that cycle
    - All other operations in pipeline would have to wait
- Pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
- RISC ISAs (including RISC-V) designed to avoid structural hazards
  - e.g. at most one memory access/instruction

Data Hazards

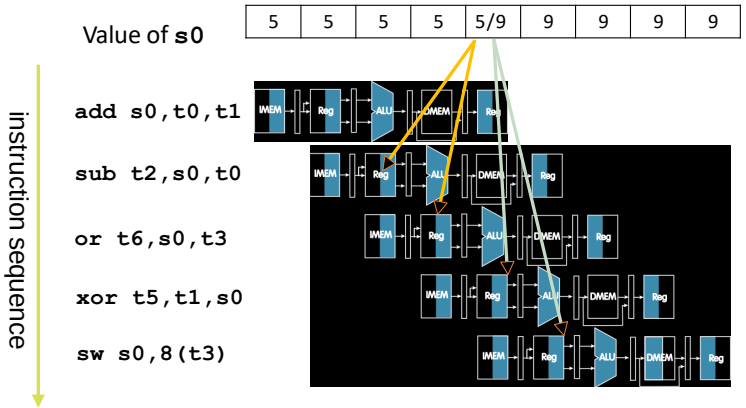
Data Hazard: Register Access

- Separate ports, but what if write to same register as read?
- Does **`sw`** in the example fetch the old or new value?





Data Hazard: ALU Result

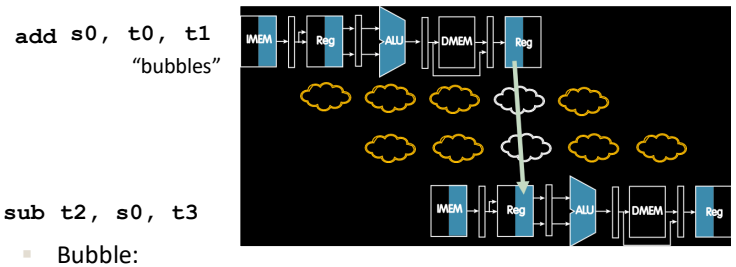


Without some fix, **`sub`** and **`or`** will calculate wrong result!

Solution 1: Stalling

- Problem: Instruction depends on result from previous instruction

```
add s0, t0, t1
sub t2, s0, t3
```

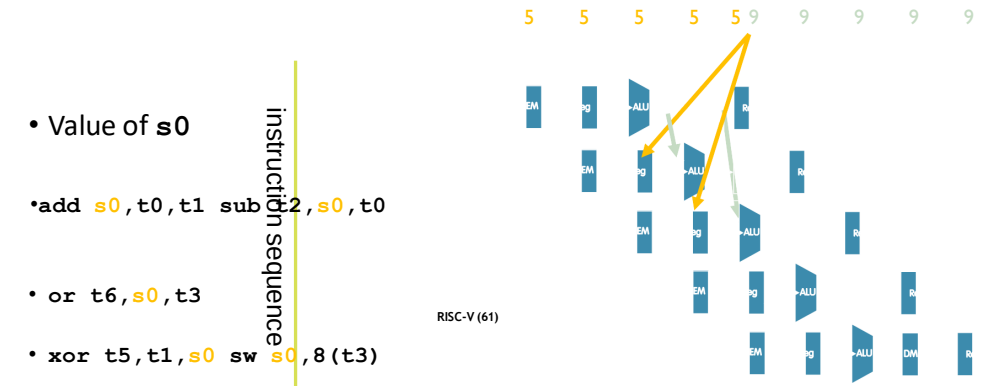


- Bubble:
  - Effectively **`nop`**: Affected pipeline stages do "nothing"

Stalls and Performance

- Stalls reduce performance
  - But stalls are required to get correct results
- Compiler can arrange code or insert **nop**s (**addi x0, x0, 0**) to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Solution 2: Forwarding

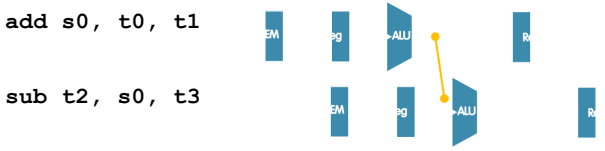


RISC-V (61)

• Forwarding: grab operand from pipeline stage, rather than register file

Forwarding (aka Bypassing)

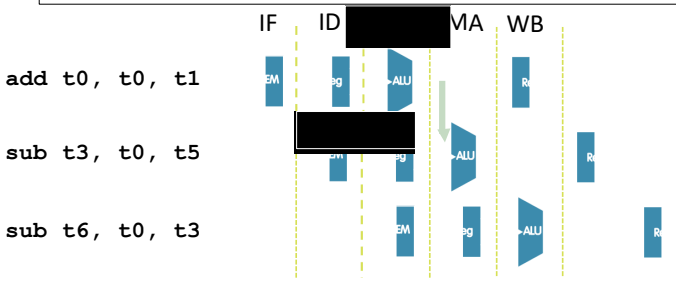
- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



RISC-V (62)

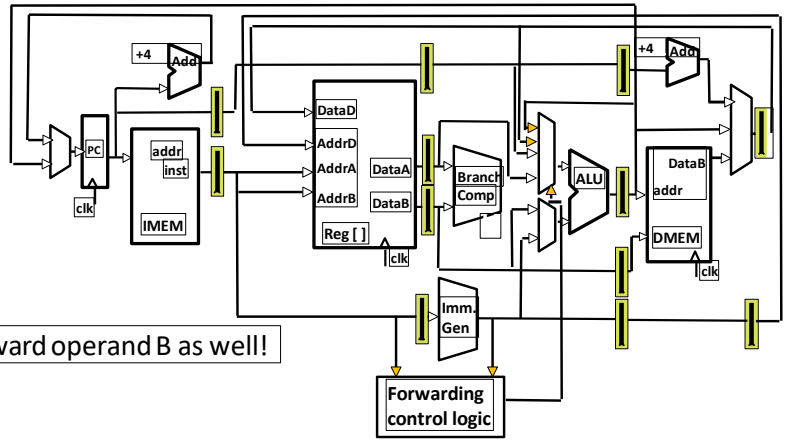
Data Needed for Forwarding (Example)

- Compare destination of older instructions in pipeline with sources of new instruction in decode stage.
- Must ignore writes to x0!



RISC-V (63)

Pipelined RV32I Datapath



Remember to forward operand B as well!

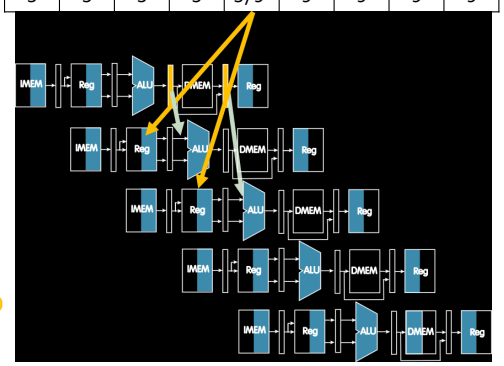
Data Hazard and Forwarding

- Value of **s0**

5	5	5	5	5/9	9	9	9	9

instruction sequence

- `add s0,t0,t1`
- `sub t2,s0,t0`
- `or t6,s0,t3`
- `xor t5,t1,s0`
- `sw s0,8(t3)`



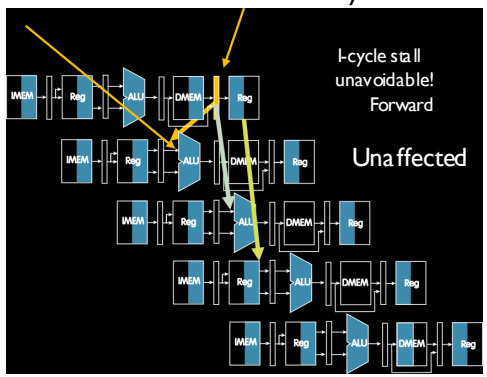
- Forwarding: grab operand from pipeline stage, rather than register file

Load Data Hazard

Needs to be here! Data from memory is here

instruction sequence

```
lw s2,20(s1)
and s4,s2,s5
or s8,s2,s6
add s9,s4,s2
slt s1,s6,s7
```

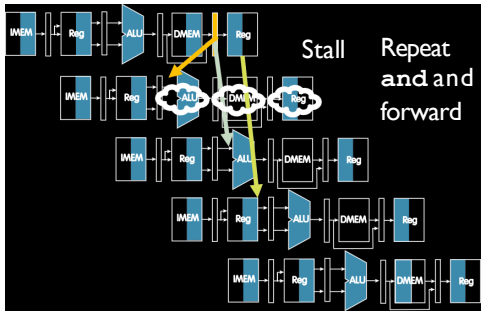


Load requires one-cycle pipeline stall

Stall Pipeline

instruction sequence

```
lw s2,20(s1)
and s4, s2, s5
nop
or s8,s2,s6
add s9,s4,s2
slt s1,s6,s7
```



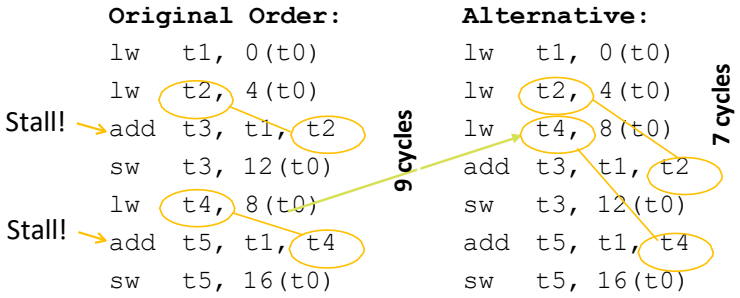
Load requires one-cycle pipeline stall

`lw` Data Hazard

- Slot after a load is called a *load delay slot*
  - If that instruction uses the result of the load, then the hardware will stall for one cycle
  - Equivalent to inserting an explicit `nop` in the slot
    - except the latter uses more code space
  - Performance loss
- Idea:
  - Put unrelated instruction into load delay slot
  - No performance loss!

Code Scheduling to Avoid Stalls

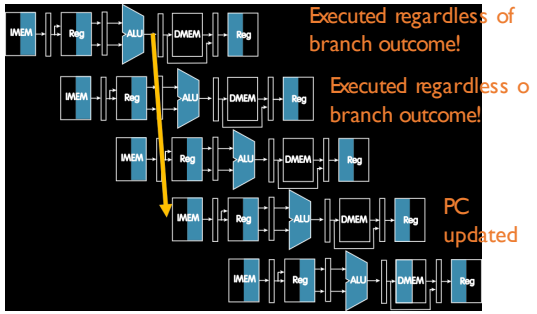
- Reorder code to avoid use of load result in the next instr!
- RISC-V code for `A[3]=A[0]+A[1]; A[4]=A[0]+A[2]`



Control Hazards

Control Hazards

```
beq t0,t1,Label
sub t2,s0,t0
or t6,s0,t3
xor t5,t1,s0
sw s0,8(t3)
```



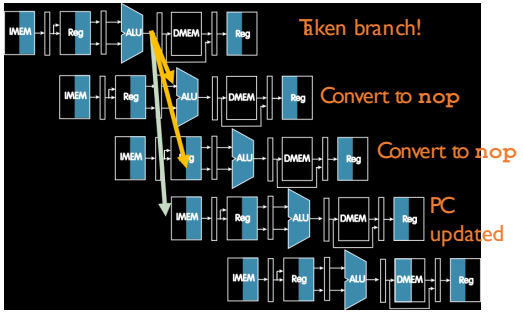
Two stall cycles after a branch!

Observation

- If branch not taken, then instructions fetched sequentially after branch are correct
- If branch or jump taken, then need to flush incorrect instructions from pipeline by converting to NOPs

Kill Instructions after Branch if Taken

```
beq t0,t1,Label
sub t2,s0,t0
or t6,s0,t3
Label: xor t5,t1
```

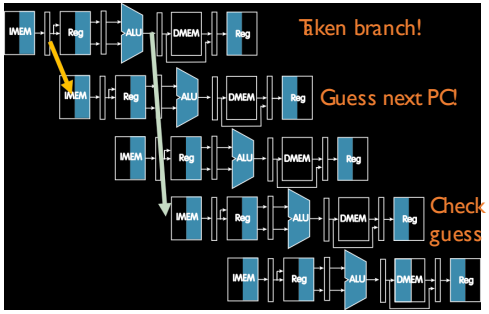


Reducing Branch Penalties

- Every taken branch in simple pipeline costs 2 dead cycles
- To improve performance, use “branch prediction” to guess which way branch will go earlier in pipeline
- Only flush pipeline if branch prediction was incorrect

Branch Prediction

```
beq t0,t1,Label
Label :…
```



Superscalar Processors

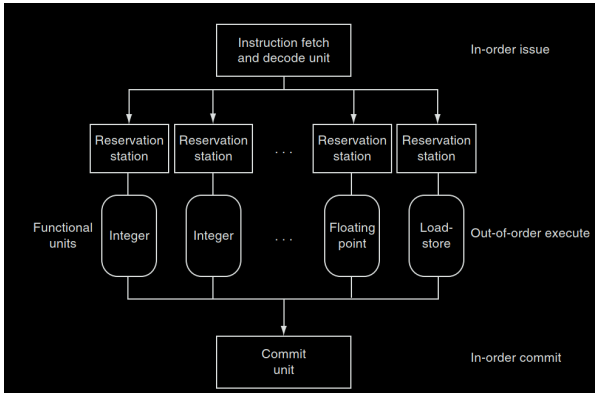
Increasing Processor Performance

1. Clock rate
   - Limited by technology and power dissipation
2. Pipelining
   - "Overlap" instruction execution
   - Deeper pipeline: 5 => 10 => 15 stages
     - Less work per stage → shorter clock cycle
     - But more potential for hazards (CPI > 1)
3. Multi-issue "superscalar" processor

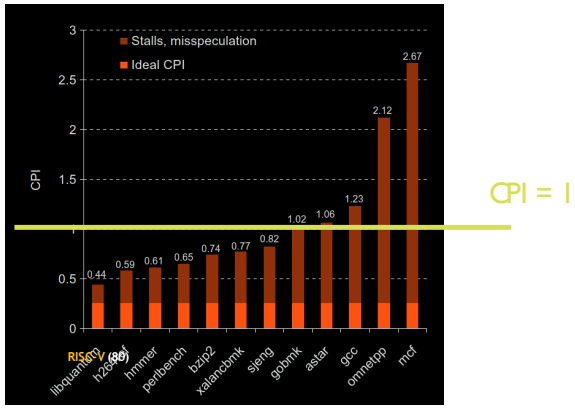
Superscalar Processor

- Multiple issue "superscalar"
  - Replicate pipeline stages ⇒ multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, so use Instructions Per Cycle (IPC)
  - E.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
  - Dependencies reduce this in practice
- "Out-of-Order" execution
  - Reorder instructions dynamically in hardware to reduce impact of hazards
- *CS152 discusses these techniques!*

Superscalar Processor



Benchmark: CPI of i7



CPI = 1

"Iron Law" of Processor Performance

CPI = Cycles Per Instruction

Can time | Can count | Can look up

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

$$\text{CPI} = \frac{\text{Cycles}}{\text{Instruction}} = \frac{\text{Time}}{\text{Program}} \div \left(\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Time}}{\text{Cycle}}\right)$$

Berkeley UNIVERSITY OF CALIFORNIA

Pipelining and ISA Design

- RISC-V ISA designed for pipelining
  - All instructions are 32-bits
    - Easy to fetch and decode in one cycle
    - Versus x86: 1- to 15-byte instructions
  - Few and regular instruction formats
    - Decode and read registers in one step
  - Load/store addressing
    - Calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

"And In conclusion…"

- We have built a processor!
  - Capable of executing all RISC-V instructions in one cycle each
- 5 Phases of execution
  - IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases
- Controller specifies how to execute instructions
  - Implemented as ROM or logic
- Pipelining improves performance
  - But we must resolve hazards

Computer Architecture 1

Computer Organization and Design
THE HARDWARE/SOFTWARE INTERFACE

[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolíc, © 2020, UC Berkeley]

Binary Prefix

Kilo, Mega, Giga, Tera, Peta, Exa, Zetta, Yotta

- Common use prefixes (all SI, except K [= k in SI])
- Confusing! Common usage of "kilobyte" means 1024 bytes, but the "correct" SI value is 1000 bytes
- Hard Disk manufacturers & Telecommunications are the only computing groups that use SI factors
  - What is advertised as a 1 TB drive actually holds about 90% of what you expect
  - A 1 Mbit/s connection transfers 10^6 bps.

Name	Abbr	Factor	SI size
Kilo	K	2^{10} = 1,024	10^3 = 1,000
Mega	M	2^{20} = 1,048,576	10^6 = 1,000,000
Giga	G	2^{30} = 1,073,741,824	10^9 = 1,000,000,000
Tera	T	2^{40} = 1,099,511,627,776	10^{12} = 1,000,000,000,000
Peta	P	2^{50} = 1,125,899,906,842,624	10^{15} = 1,000,000,000,000,000
Exa	E	2^{60} = 1,152,921,504,606,846,976	10^{18} = 1,000,000,000,000,000,000
Zetta	Z	2^{70} = 1,180,591,620,717,411,303,424	10^{21} = 1,000,000,000,000,000,000,000
Yotta	Y	2^{80} = 1,208,925,819,614,629,174,706,176	10^{24} = 1,000,000,000,000,000,000,000,000

kibi, mebi, gibi, tebi, pebi, exbi, zebi, yobi

- IEC Standard Prefixes [only to exbi officially]

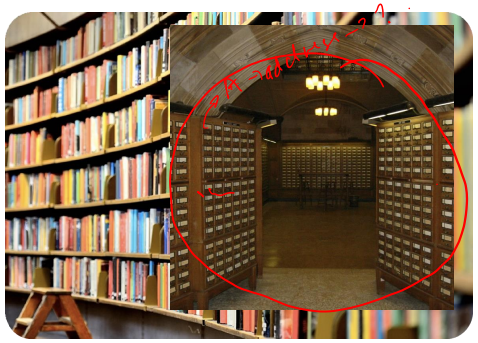
Name	Abbr	Factor
kibi	Ki	2^{10} = 1,024
mebi	Mi	2^{20} = 1,048,576
gibi	Gi	2^{30} = 1,073,741,824
tebi	Ti	2^{40} = 1,099,511,627,776
pebi	Pi	2^{50} = 1,125,899,906,842,624
exbi	Ei	2^{60} = 1,152,921,504,606,846,976
zebi	Zi	2^{70} = 1,180,591,620,717,411,303,424
yobi	Yi	2^{80} = 1,208,925,819,614,629,174,706,176

- International Electrotechnical Commission (IEC) in 1999
- Names come from shortened versions of the original SI prefixes (same pronunciation) and bi is short for "binary", but pronounced "bee" :-(
- Now SI prefixes only have their base-10 meaning and never have a base-2 meaning

Library Analogy

Why are Large Memories Slow? Library Analogy

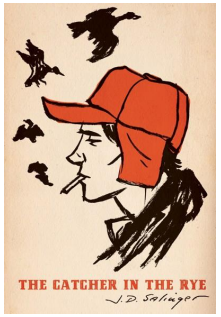
- Time to find a book in a large library
  - Search a large card catalog – (mapping title/author to index number)
  - Round-trip time to walk to the stacks and retrieve the desired book
- Larger libraries worsen both delays
- Electronic memories have same issue, plus the technologies used to store a bit slow down as density increases (e.g., SRAM vs. DRAM vs. Disk)



What we want is a large, yet fast memory!

What To Do: Library Analogy

- Write a report using library books
  - E.g., works of J.D. Salinger
- Go to library, look up books, fetch from stacks, and place on desk in library. If need more, check out, keep on desk
  - But don't return earlier books since might need them
- You hope this collection of ~10 books on desk enough to write report, despite 10 being only 0.00001% of books in UC Berkeley libraries



New-School Machine Structures

Software

Parallel Requests
Assigned to computer
e.g., Search "Cats"

Parallel Threads
Assigned to core e.g., Lookup, Ads

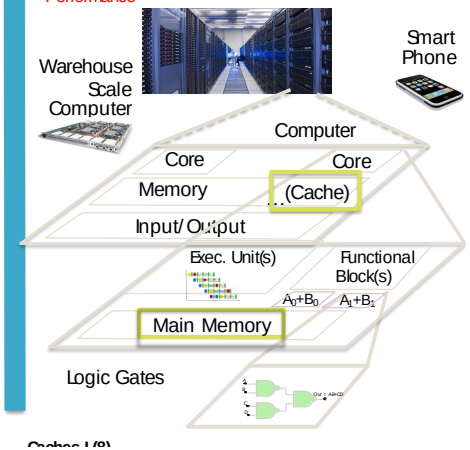
Parallel Instructions
>1 instruction @ one time
e.g., 5 pipelined instructions

Parallel Data
>1 data item @ one time
e.g., Add of 4 pairs of words

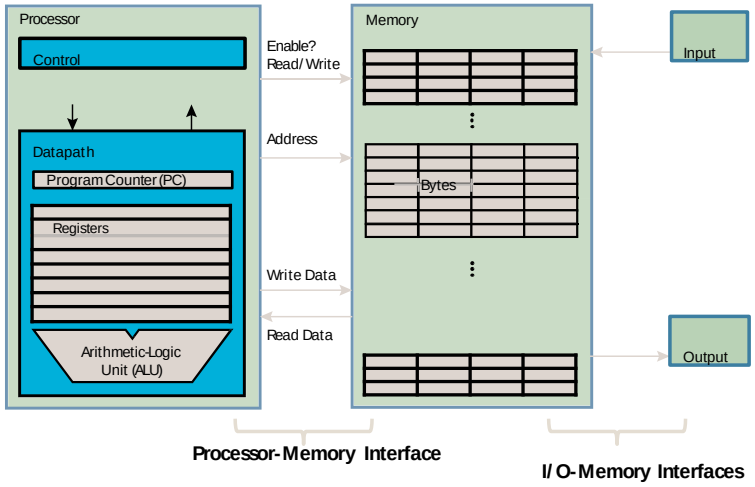
Hardware descriptions
All gates work in parallel at same time

Harness Parallelism & Achieve High Performance

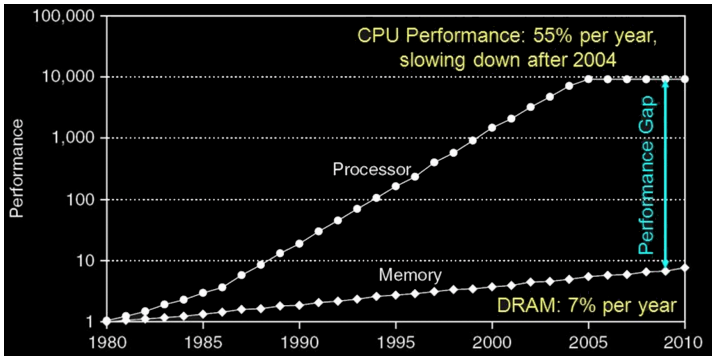
Hardware



Components of a Computer



Processor-DRAM Gap (Latency)



1980 microprocessor executes ~one instruction in same time as DRAM access
2020 microprocessor executes ~1000 instructions in same time as DRAM access

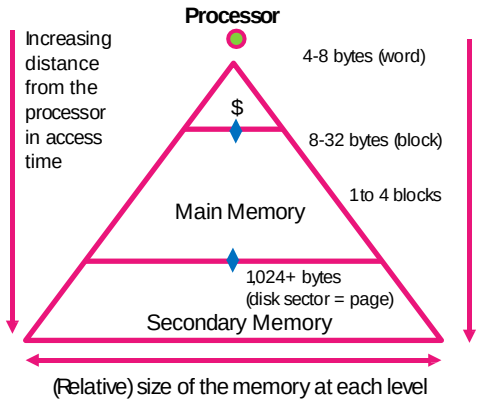
Slow DRAM access has disastrous impact on CPU performance!

Memory Hierarchy

Memory Caching

- Mismatch between processor and memory speeds leads us to add a new level…
  - Introducing a "memory cache"
- Implemented with same IC processing technology as the CPU (usually integrated on same chip)
  - faster but more expensive than DRAM memory.
- Cache is a copy of a subset of main memory
- Most processors have separate caches for instructions and data.

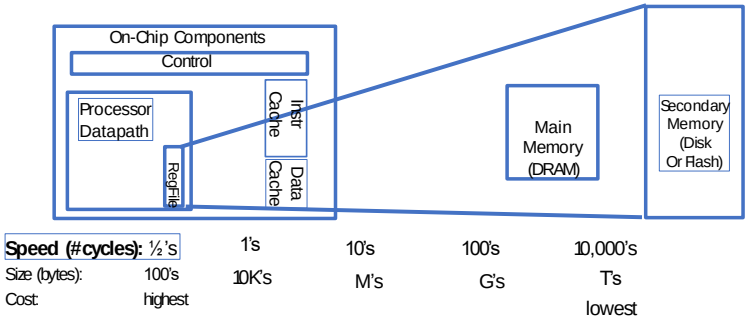
Characteristics of the Memory Hierarchy



Inclusive– what is in L1$ is a subset what is in MM that is a subset of is in SM

Typical Memory Hierarchy

- The Trick: present processor with as much memory as is available in the cheapest technology at the speed offered by the fastest technology



Memory Hierarchy

- If level closer to Processor, it is:
  - Smaller
  - Faster
  - More expensive
  - subset of lower levels (contains most recently used data)
- Lowest Level (usually disk=HDD/ SSD) contains all available data (does it go beyond the disk?)
- Memory Hierarchy presents the processor with the illusion of a very large & fast memory

Locality, Design, Management

Memory Hierarchy Basis

- Cache contains copies of data in memory that are being used.
- Memory contains copies of data on disk that are being used.
- Caches work on the principles of temporal and spatial locality.
  - Temporal locality (locality in time): If we use it now, chances are we'll want to use it again soon.
  - Spatial locality (locality in space): If we use a piece of memory, chances are we'll use the neighboring pieces soon.

What to Do About Locality

- Temporal Locality
  - If a memory location is referenced then it will tend to be referenced again soon
- Keep most recently accessed data items closer to the processor
- Spatial Locality
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon
- Move blocks consisting of contiguous words closer to the processor

Cache Design

- **How do we organize cache?**
- **Where does each memory address map to?**
  - (Remember that cache is subset of memory, so multiple memory addresses map to the same cache location.)
- **How do we know which elements are in cache?**
- **How do we quickly locate them?**

How is the Hierarchy Managed?

- **registers ↔ memory**
  - By compiler (or assembly level programmer)
- **cache ↔ main memory**
  - By the cache controller hardware
- **main memory ↔ disks (secondary storage)**
  - By the operating system (virtual memory)
  - Virtual to physical address mapping assisted by the hardware ('translation lookaside buffer' or TLB)
  - By the programmer (files)

“And in Conclusion…”

- **Caches provide an illusion to the processor that the memory is infinitely large and infinitely fast**

Example: caching instructions

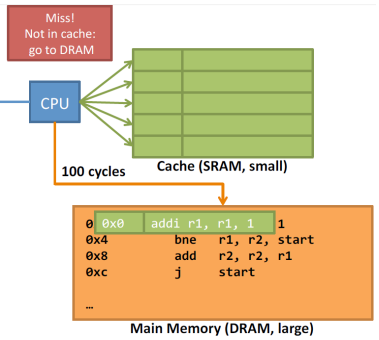
```
0x0 start:  addi  r1, r1, 1
0x4         bne   r1, r2, start
0x8         add   r2, r2, r1
0xc         j     start

…
```

Main Memory (DRAM, large)

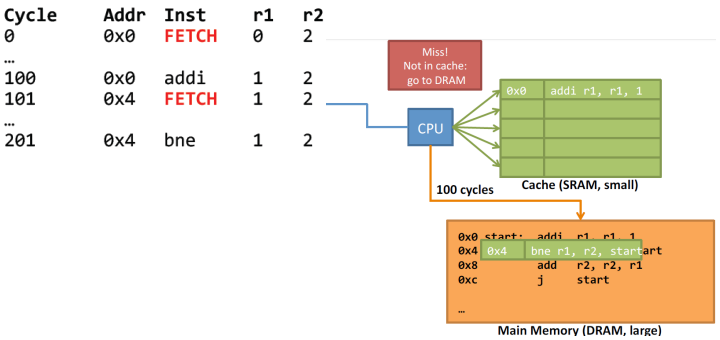
Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2



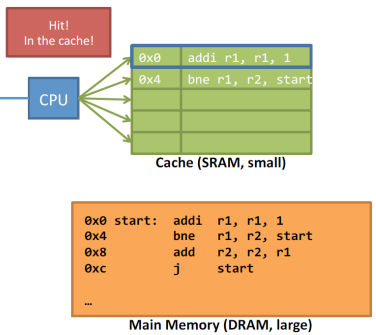
Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2
101	0x4	FETCH	1	2
…				
201	0x4	bne	1	2



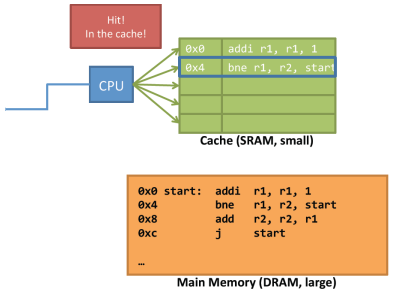
Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2
101	0x4	FETCH	1	2
…				
201	0x4	bne	1	2
202	0x0	addi	2	2



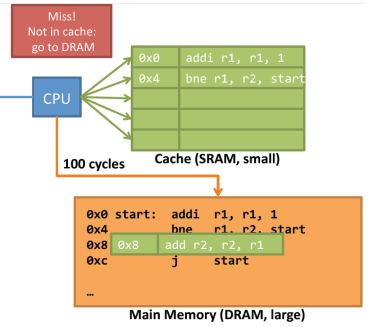
Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2
101	0x4	FETCH	1	2
…				
201	0x4	bne	1	2
202	0x0	addi	2	2
203	0x4	bne	2	2



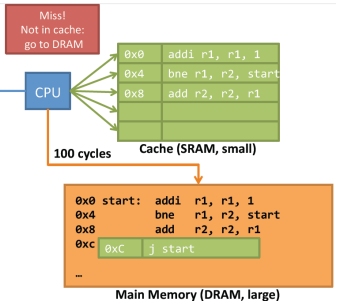
Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2
101	0x4	FETCH	1	2
…				
201	0x4	bne	1	2
202	0x0	addi	2	2
203	0x4	bne	2	2
204	0x8	FETCH	2	2
…				
304	0x8	add	2	4



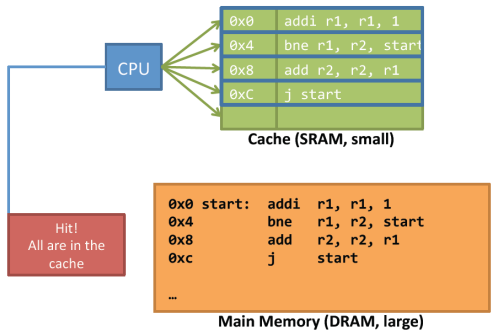
Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2
101	0x4	FETCH	1	2
…				
201	0x4	bne	1	2
202	0x0	addi	2	2
203	0x4	bne	2	2
204	0x8	FETCH	2	2
…				
304	0x8	add	2	4
305	0xC	FETCH	2	4
…				
405	0xC	j	2	4



Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2
101	0x4	FETCH	1	2
…				
201	0x4	bne	1	2
202	0x0	addi	2	2
203	0x4	bne	2	2
204	0x8	FETCH	2	2
…				
304	0x8	add	2	4
305	0xC	FETCH	2	4
…				
405	0xC	j	2	4
406	0x0	addi	3	4
407	0x4	bne	3	4
408	0x0	addi	4	4
409	0x4	bne	4	4
410	0x8	add	4	8
411	0xC	j	4	8
412	0x0	addi	5	8

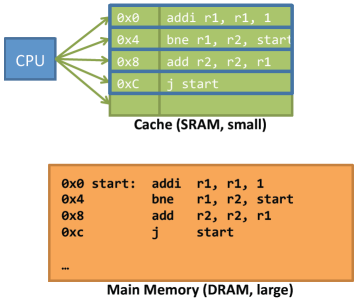


Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2
101	0x4	FETCH	1	2
…				
201	0x4	bne	1	2
202	0x0	addi	2	2
203	0x4	bne	2	2
204	0x8	FETCH	2	2
…				
304	0x8	add	2	4
305	0xC	FETCH	2	4
…				
405	0xC	j	2	4
406	0x0	addi	3	4
407	0x4	bne	3	4
408	0x0	addi	4	4
409	0x4	bne	4	4
410	0x8	add	4	8
411	0xC	j	4	8
412	0x0	addi	5	8

100 cycles per instruction: **in DRAM**

1 cycle per instruction: **all in cache**



Computer Architecture 1

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

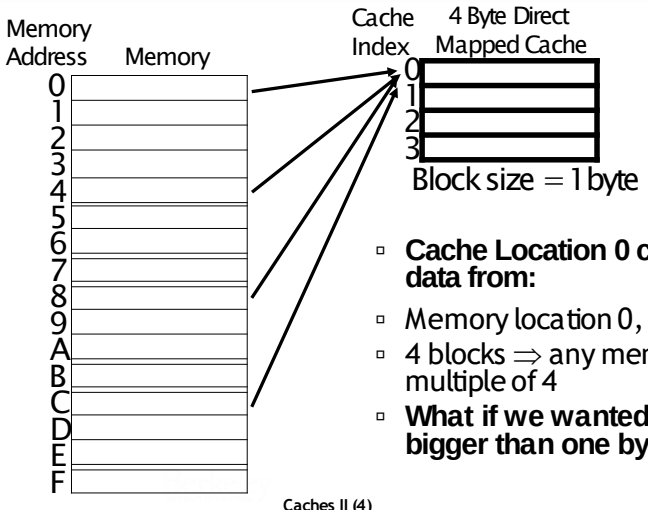
[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolíc, © 2020, UC Berkeley]

Direct Mapped Caches

Direct-Mapped Cache (1/4)

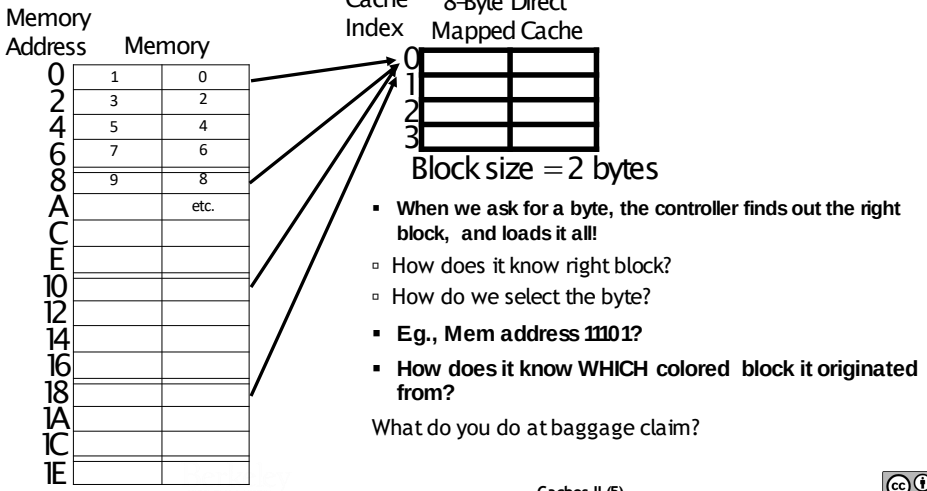
- **In a direct-mapped cache, each memory address is associated with one possible block within the cache**
  - Therefore, we only need to look in a single location in the cache for the data if it exists in the cache
  - Block is the unit of transfer between cache and memory

Direct-Mapped Cache (2/4)



- **Cache Location 0 can be occupied by data from:**
  - Memory location 0, 4, 8, ...
  - 4 blocks ⇒ any memory location that is multiple of 4
  - **What if we wanted a block to be bigger than one byte?**

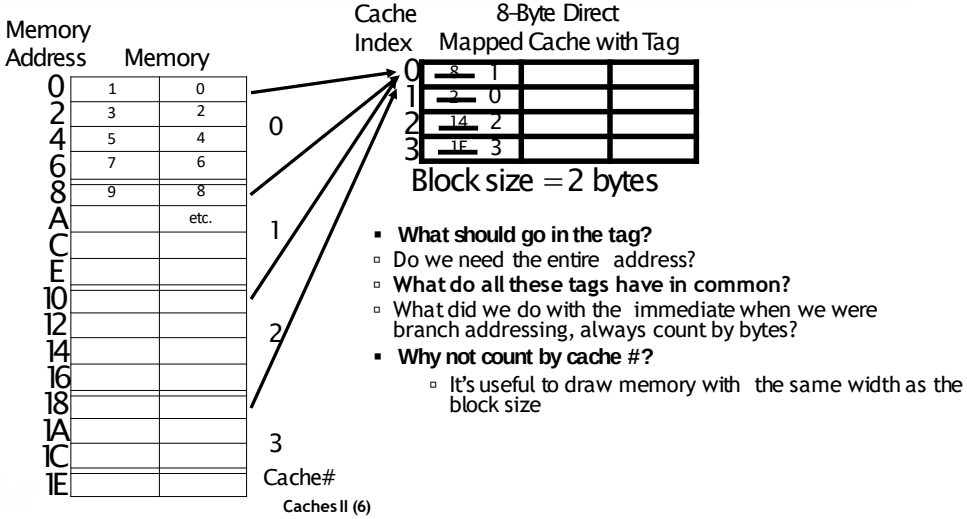
Direct-Mapped Cache (3/4)



- **When we ask for a byte, the controller finds out the right block, and loads it all!**
  - How does it know right block?
  - How do we select the byte?
- **Eg., Mem address 11101?**
- **How does it know WHICH colored block it originated from?**

What do you do at baggage claim?

Direct-Mapped Cache (4/4)



- **What should go in the tag?**
  - Do we need the entire address?
  - **What do all these tags have in common?**
  - What did we do with the immediate when we were branch addressing, always count by bytes?
- **Why not count by cache #?**
  - It's useful to draw memory with the same width as the block size

Issues with Direct-Mapped

- Since multiple memory addresses map to same cache index, how do we tell which one is in there?
- What if we have a block size > 1 byte?
- Answer: divide memory address into three fields

tttttttttttttttt	iiiiiiiiii	oooo
Tag to check if have correct block	**Index to select block**	**Byte offset within block**

Direct-Mapped Cache Terminology

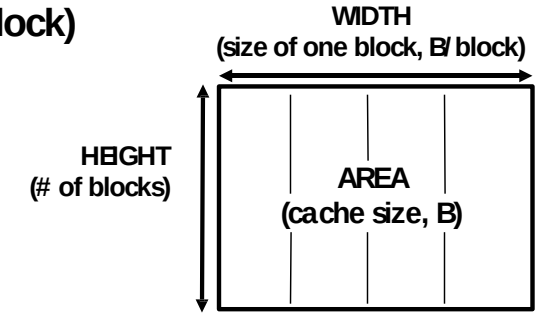
- **All fields are read as unsigned integers.**
- **Index**
  - specifies the cache index (which "row"/block of the cache we should look in)
- **Offset**
  - once we've found correct block, specifies which byte within the block we want
- **Tag**
  - the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location

TIO Cache

Tag	Index	Offset

AREA (cache size, B)
= HEIGHT (# of blocks) * WIDTH (size of one block, B/ block)

$$2^{(H+W)} = 2^H * 2^W$$



Direct Mapped Example

DM Cache Example (1/5)

- Cache parameters:
  - Direct-mapped, address space of 64B, block size of $_{4B}$, cache size of $_{16B}$, write-through
- TIO Breakdown:
  - O = $\log_2(4) = 2$
  - Cache size / block size = $16/4 = 4$, so I = $\log_2(4) = 2$
  - A = $\log_2(64) = 6$ bits, so T = 6 – 2 – 2 = 2
- Bits in cache = $2^2 \times (8 \times 2^2 + 2 + 1) = 140$ bits

Memory Addresses: XX XX XX
Block address

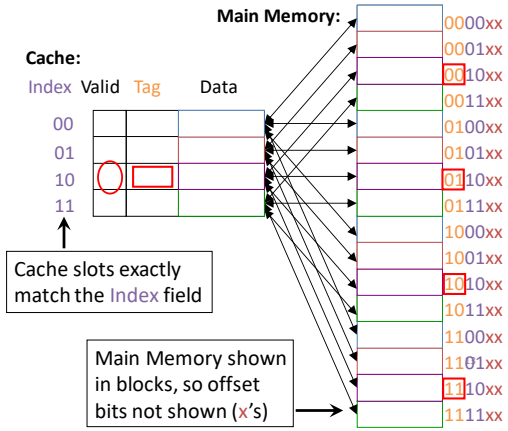
DM Cache Example (2/5)

- Cache parameters:
  - Direct-mapped, address space of 64B, block size of $_{4B}$, cache size of $_{16B}$, write-through
  - Offset – 2 bits, Index – 2 bits, Tag – 2 bits

Index	V	Tag	Offset 00	01	10	11
00	X	XX	0x??	0x??	0x??	0x??
01	X	XX	0x??	0x??	0x??	0x??
10	X	XX	0x??	0x??	0x??	0x??
11	X	XX	0x??	0x??	0x??	0x??

- 35 bits per index/slot, 140 bits to implement

DM Cache Example (3/5)



Which blocks map to each row of the cache?
(see colors)

On a memory request:
(let's say 001011_{two})

1) Take Index field (10)

2) Check if Valid bit is true in that row of cache

3) If valid, then check if Tag matches

DM Cache Example (4/5)

- Consider the sequence of memory address accesses

Starting with a cold cache: 0 2 4 8 20 16 0 2

000000
0 miss

00	1	00	M[0]	M[1]	M[2]	M[3]
01	0	00	0x??	0x??	0x??	0x??
10	0	00	0x??	0x??	0x??	0x??
11	0	00	0x??	0x??	0x??	0x??

000010
2 hit

00	1	00	M[0]	M[1]	M[2]	M[3]
01	0	00	0x??	0x??	0x??	0x??
10	0	00	0x??	0x??	0x??	0x??
11	0	00	0x??	0x??	0x??	0x??

000100
4 miss

00	1	00	M[0]	M[1]	M[2]	M[3]
01	1	00	M[4]	M[5]	M[6]	M[7]
10	0	00	0x??	0x??	0x??	0x??
11	0	00	0x??	0x??	0x??	0x??

001000
8 miss

00	1	00	M[0]	M[1]	M[2]	M[3]
01	1	00	M[4]	M[5]	M[6]	M[7]
10	1	00	M[8]	M[9]	M[10]	M[11]
11	0	00	0x??	0x??	0x??	0x??

DM Cache Example (5/5)

- Consider the sequence of memory address accesses

Starting with a cold cache: 0 2 4 8 20 16 0 2

010100
20 miss

00	1	00	M[0]	M[1]	M[2]	M[3]
01	1	00	M[4]	M[5]	M[6]	M[7]
10	1	00	M[8]	M[9]	M[10]	M[11]
11	0	00	0x??	0x??	0x??	0x??

010000
16 miss

00	1	00	M[0]	M[1]	M[2]	M[3]
01	1	01	M[20]	M[21]	M[22]	M[23]
10	1	00	M[8]	M[9]	M[10]	M[11]
11	0	00	0x??	0x??	0x??	0x??

000000
0 miss

00	1	01	M[16]	M[17]	M[18]	M[19]
01	1	01	M[20]	M[21]	M[22]	M[23]
10	1	00	M[8]	M[9]	M[10]	M[11]
11	0	00	0x??	0x??	0x??	0x??

000010
2 hit

00	1	00	M[0]	M[1]	M[2]	M[3]
01	1	01	M[20]	M[21]	M[22]	M[23]
10	1	00	M[8]	M[9]	M[10]	M[11]
11	0	00	0x??	0x??	0x??	0x??

- 8 requests, 6 misses – last slot was never used!

Worst-Case for Direct-Mapped

- Cold DM $ that holds four 1-word blocks
- Consider the memory accesses: 0, 16, 0, 16,...

000000 **0** Miss

00	M[0-3]

010000 **16** Miss

~~00~~	M[0-3]

000000 **0** Miss

~~01~~	M[16-19]

. . .

- HR of 0%
  - Ping pong effect: alternating requests that map into the same cache slot
- Does fully associative have this problem?

Direct-Mapped Cache Example (1/3)

- **Suppose we have a 8B of data in a direct-mapped cache with 2-byte blocks**
  - Sound familiar?
- **Determine the size of the tag, index and offset fields if using a 32-bit arch (RV32)**
- **Offset**
  - need to specify correct byte within a block
  - block contains 2 bytes = 2^1 bytes
  - need 1 bit to specify correct byte

Direct-Mapped Cache Example (2/3)

- **Index: (~index into an "array of blocks")**
  - need to specify correct block in cache
  - cache contains 8 B = 2^3 bytes
  - block contains 2 B = 2^1 bytes
  - # blocks/cache

$$= \frac{\text{bytes/cache}}{\text{bytes/block}}$$

$$= \frac{2^3 \text{ bytes/cache}}{2^1 \text{ bytes/block}}$$

$$= 2^2 \text{ blocks/cache}$$

  - need 2 bits to specify this many blocks

Direct-Mapped Cache Example (3/3)

- **Tag: use remaining bits as tag**
  - tag length = addr length - offset - index
    = 32 - 1 - 2 bits
    = 29 bits
  - so tag is leftmost 29 bits of memory address
  - Tag can be thought of as "cache number"
- **Why not full 32-bit address as tag?**
  - All bytes within block need same address
  - Index must be same for every address within a block, so it's redundant in tag check, thus can leave off to save memory

Memory Access without Cache

- **Load word instruction:** `lw t0, 0(t1)`
- `t1` **contains** 1022_{ten}, `Memory[1022] = 99`

1. Processor issues address 1022_{ten} to `Memory`
2. Memory reads word at address 1022_{ten} (`99`)
3. Memory sends `99` to Processor
4. Processor loads `99` into register `t0`

Memory Access with Cache

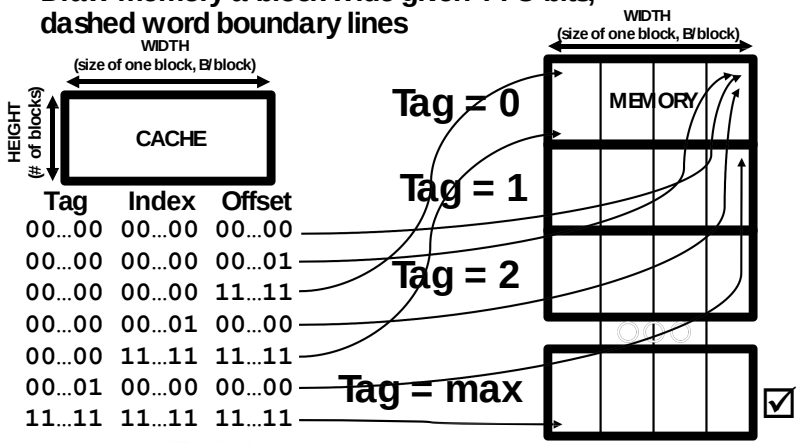
- **Load word instruction:** `lw t0, 0(t1)`
- `t1` **contains** 1022_{ten}, `Memory[1022] = 99`
- **With cache (similar to a hash)**
  1. Processor issues address 1022_{ten} to Cache
  2. Cache checks to see if has copy of data at address 1022_{ten}

     2a. If finds a match (Hit): cache reads `99`, sends to processor 2b. No match (Miss): cache sends address `1022` to Memory
     - I. Memory reads `99` at address 1022_{ten}
     - II. Memory sends `99` to Cache
     - III. Cache replaces word with new `99`
     - IV. Cache sends `99` to processor
  3. Processor loads `99` into register `t0`

Solving Cache problems

Tag	Index	Offset

- **Draw memory a block wide given T I O bits, dashed word boundary lines**



Cache Terminology

Caching Terminology

- When reading memory, 3 things can happen:
  - cache hit: cache block is valid and contains proper address, so read desired word
  - cache miss: nothing in cache in appropriate block, so fetch from memory
  - cache miss, block replacement: wrong data is in cache at appropriate block, so discard it and fetch desired data from memory (cache always copy)

Cache Temperatures

- Cold
  - Cache empty
- Warming
  - Cache filling with values you'll hopefully be accessing again soon
- Warm
  - Cache is doing its job, fair % of hits
- Hot
  - Cache is doing very well, high % of hits

Cache Terms

- Hit rate: fraction of access that hit in the cache
- Miss rate: 1– Hit rate
- Miss penalty: time to replace a block from lower level in memory hierarchy to cache
- Hit time: time to access cache memory (including tag comparison)
- Abbreviation: "$" = cache
  - … a Berkeley innovation!

One More Detail: Valid Bit

- When start a new program, cache does not have valid information for this program
- Need an indicator whether this tag entry is valid for this program
- Add a "valid bit" to the cache tag entry 0 → cache miss, even if by chance, address = tag 1 → cache hit, if processor address = tag

"And in Conclusion…"

- We have learned the operation of a direct-mapped cache
- Mechanism for transparent movement of data among levels of a memory hierarchy
  - set of address/value bindings
  - address → index to set of candidates
  - compare desired address with tag
  - service hit or miss
  - load new block and binding on miss

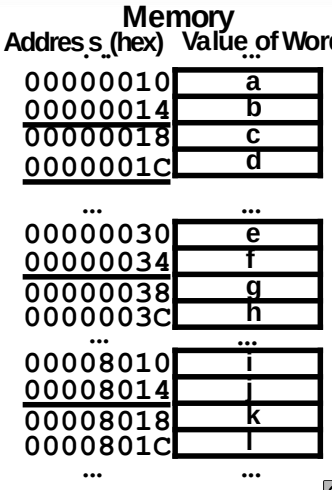
☑

Accessing data in a direct mapped cache

- Ex.: 16KB of data, direct-mapped, 4 word blocks
  - Can you work out height, width, area?
- Read 4 addresses

1. 0x00000014
2. 0x0000001C
3. 0x00000034
4. 0x00008014

- Memory values here:



Accessing data in a direct mapped cache

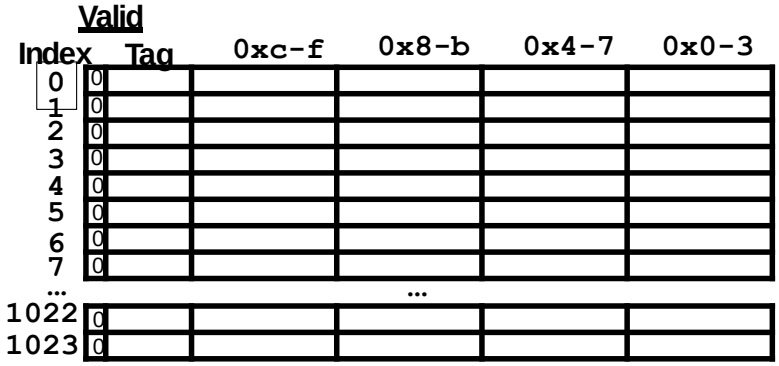
- 4 Addresses:
  - 0x00000014, 0x0000001C, 0x00000034, 0x00008014
- 4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields

```
00000000000000000 0000000001 0100
00000000000000000 0000000001 1100
00000000000000000 0000000011 0100
00000000000000010 0000000001 0100
```

Tag Index Offset

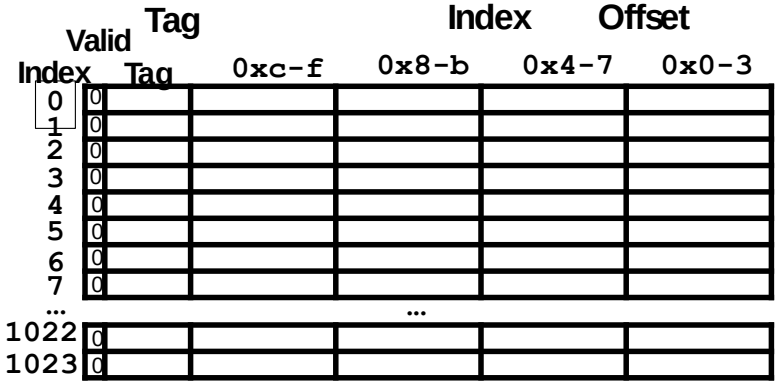
Example: 16 KB Direct-Mapped Cache, 16B blocks

- Valid bit: determines whether anything is stored in that row (when computer initially powered up, all entries invalid)



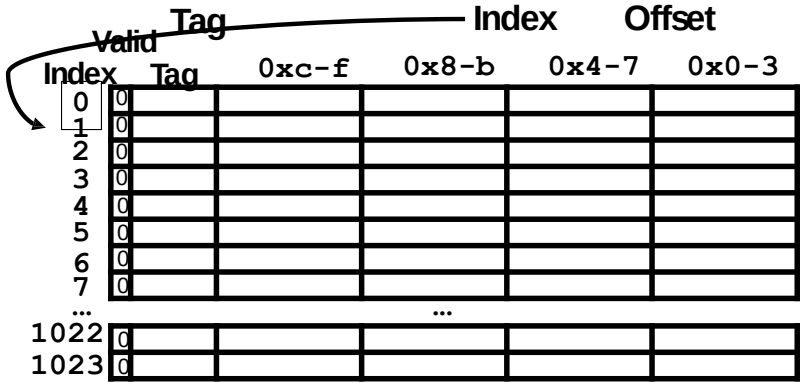
1. Read 0x00000014

- 00000000000000000 0000000001 0100



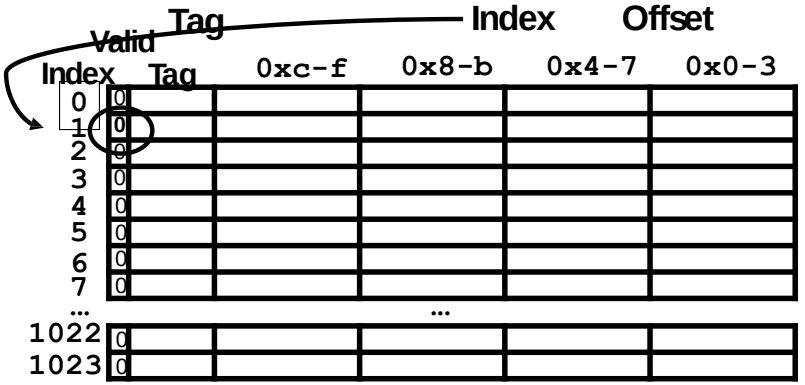
So we read block 1 (0000000001)

- 00000000000000000 0000000001 0100



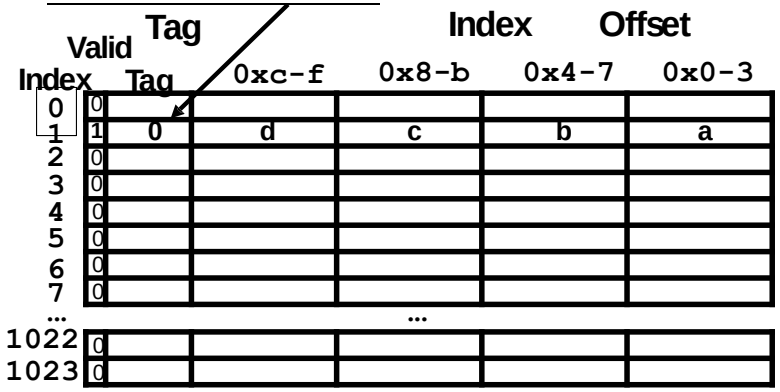
No valid data

- 00000000000000000 0000000001 0100



So load that data into cache, setting tag, valid

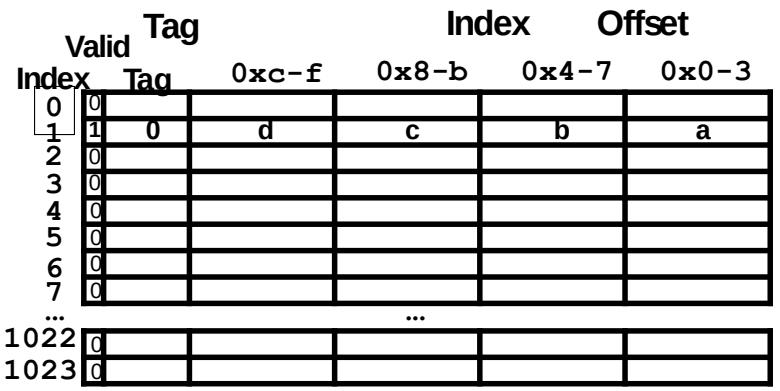
- 00000000000000000 0000000001 0100



Garcia, Nikolić

2. Read 0x0000001C = 0…00 0..0011100

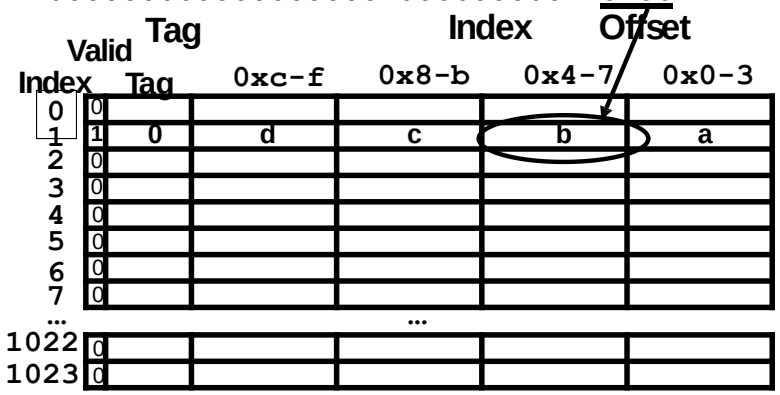
- 00000000000000000 0000000001 1100



Garcia, Nikolić

Read from cache at offset, return word b

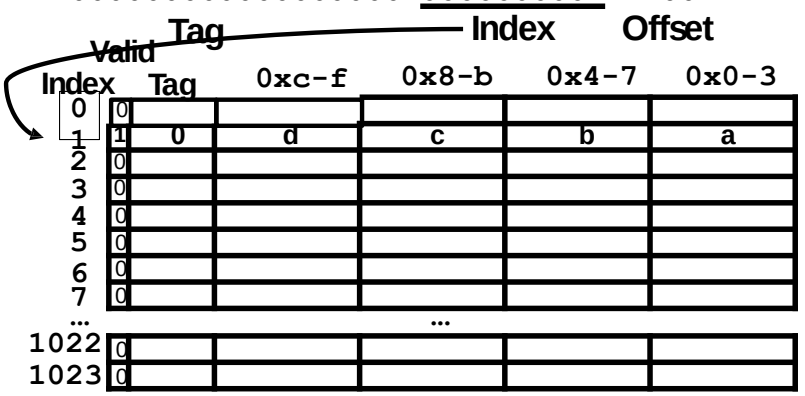
- 00000000000000000 0000000001 0100



Garcia, Nikolić

Index is Valid

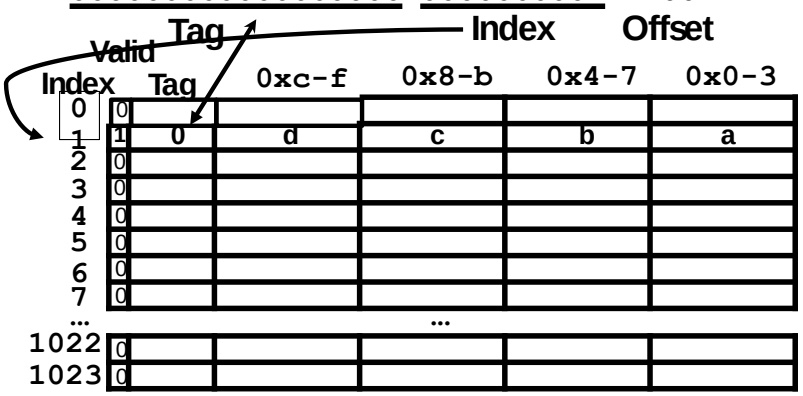
- 00000000000000000 0000000001 1100



Garcia, Nikolić

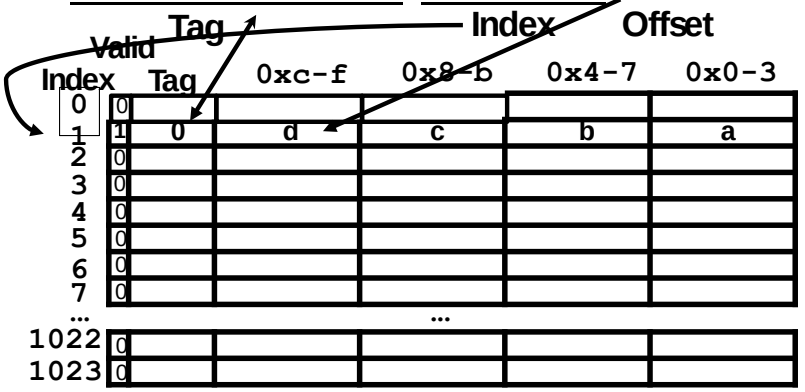
Index is Valid, Tag Matches

- 00000000000000000 0000000001 1100



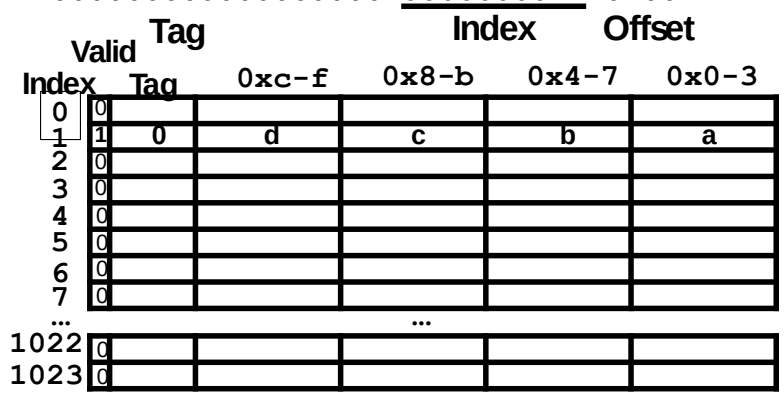
Index is Valid, Tag Matches, return d

- 00000000000000000 0000000001 1100



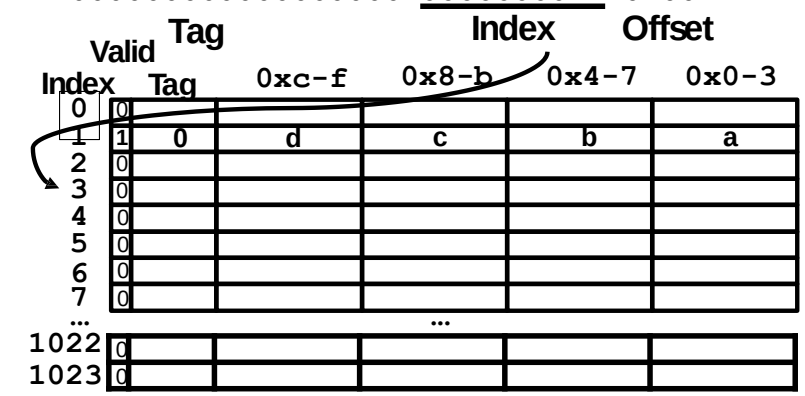
3. Read 0x00000034 = 0…00 0..0110100

- 00000000000000000 0000000011 0100



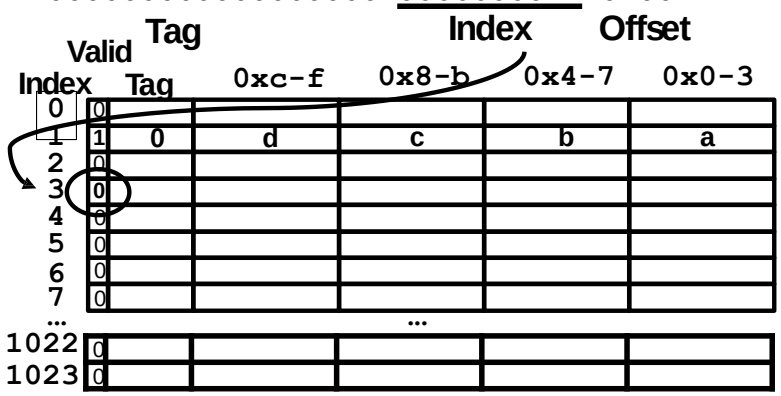
So read block 3

- 00000000000000000 0000000011 0100



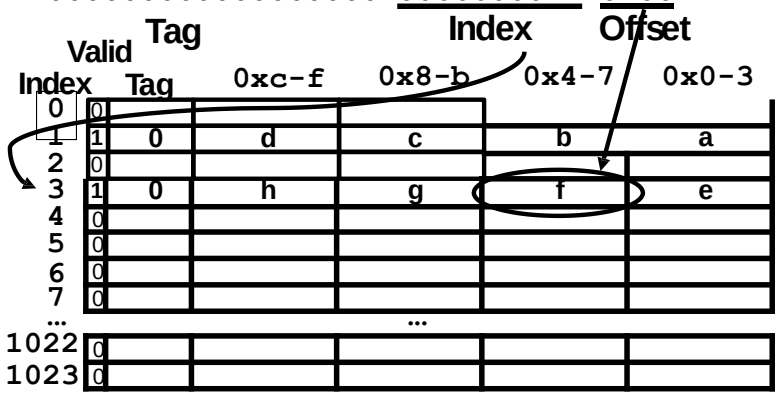
No valid data

- 00000000000000000 0000000011 0100



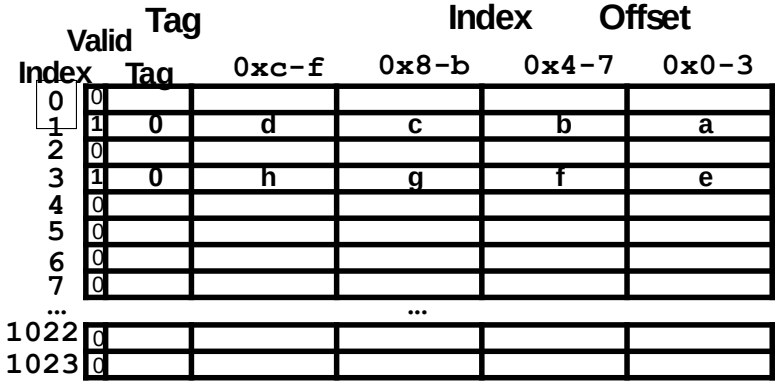
Load that cache block, return word f

- 00000000000000000 0000000011 0100



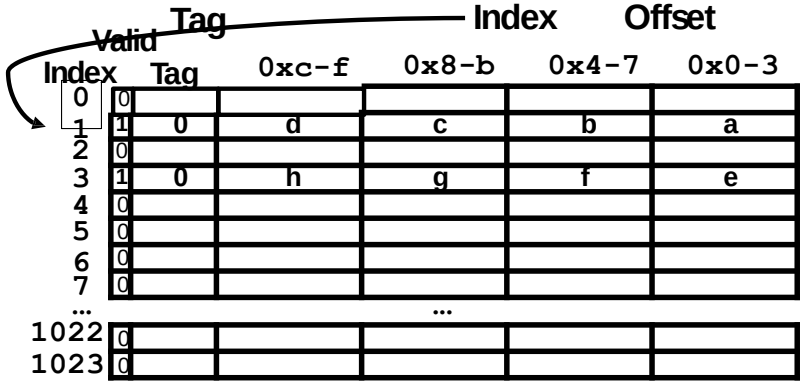
4. Read 0x00008014 = 0…10 0..0010100

- 00000000000000010 0000000001 0100



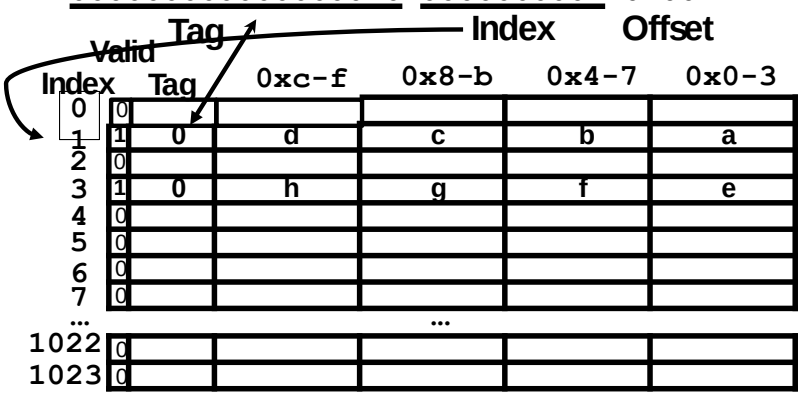
So read Cache Block 1, Data is Valid

- 00000000000000010 0000000001 0100



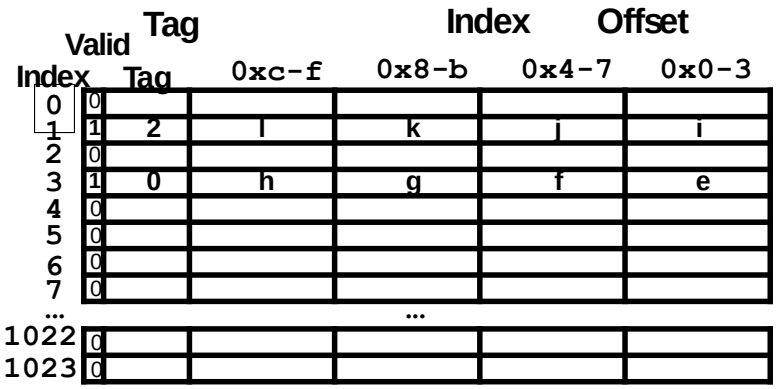
Cache Block 1 Tag does not match (0 ≠ 2)

- 00000000000000010 0000000001 0100



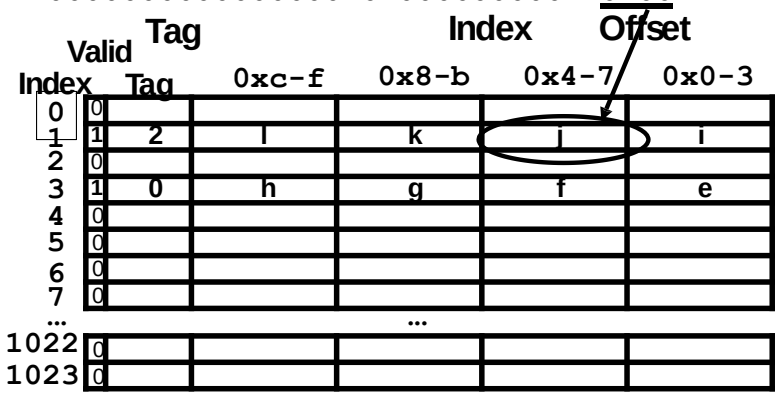
Miss, so replace block 1 with new data & tag

- 00000000000000010 0000000001 0100



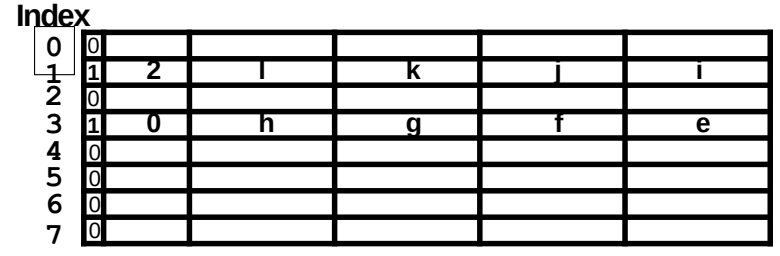
And return word J

- 00000000000000010 0000000001 0100



Do an example yourself. What happens?

- Chose from: Cache: Hit, Miss, Miss w. replace
  Values returned: a ,b, c, d, e, ..., k, l
- Read address 0x00000030 ?
  00000000000000000 0000000011 0000
- Read address 0x0000001c ?
  00000000000000000 0000000001 1100



Answers

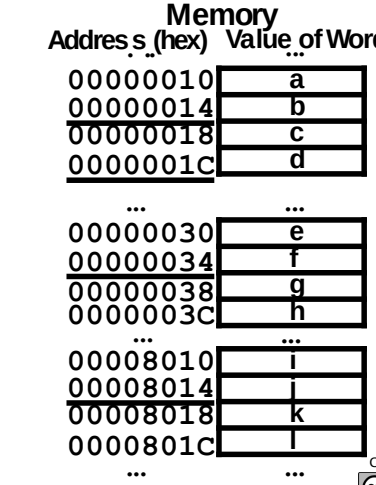
- 0x00000030 a **hit**

  Index = 3, Tag matches, Offset = 0, value = e

- 0x0000001c a **miss**

  Index = 1, Tag mismatch, so replace from memory, Offset = 0xc, value = d

- **Since reads, values must = memory values whether or not cached:**
  - 0x00000030 = e
  - 0x0000001c = d

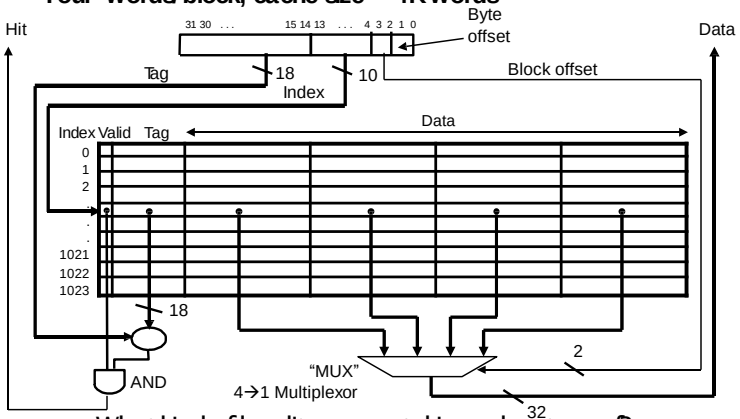


☑

Writes, Block Sizes, Misses

Multiword-Block Direct-Mapped Cache

- **Four words/block, cache size = 4K words**



What kind of locality are we taking advantage of?

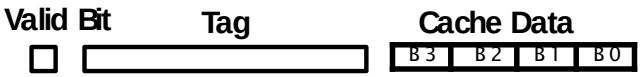
What to do on a write hit?

- **Write-through**
  - Update both cache and memory
- **Write-back**
  - update word in cache block
  - allow memory word to be "stale"
  - add 'dirty' bit to block
    - memory & Cache inconsistent
    - needs to be updated when block is replaced
  - ... OS flushes cache before I/O...
- **Performance trade-offs?**

Block Size Tradeoff

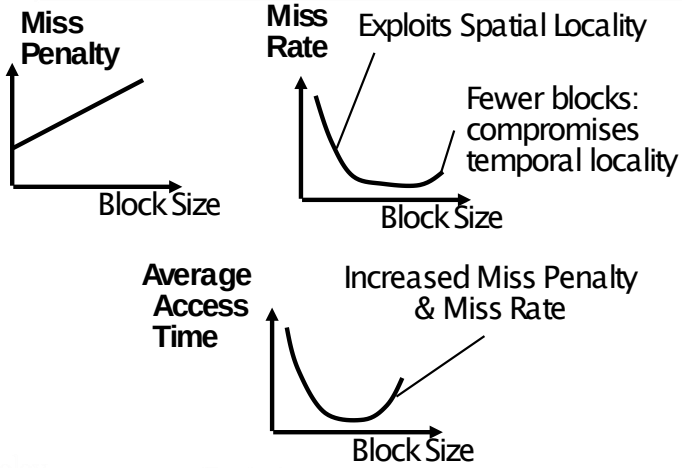
- **Benefits of Larger Block Size**
  - Spatial Locality: if we access a given word, we're likely to access other nearby words soon
  - Very applicable with Stored-Program Concept
  - Works well for sequential array accesses
- **Drawbacks of Larger Block Size**
  - Larger block size means larger miss penalty
    - on a miss, takes longer time to load a new block from next level
  - If block size is too big relative to cache size, then there are too few blocks
    - Result: miss rate goes up

Extreme Example: One Big Block



- **Cache Size = 4 bytes Block Size = 4 bytes**
  - Only ONE entry (row) in the cache!
- **If item accessed, likely accessed again soon**
  - But unlikely will be accessed again immediately!
- **The next access will likely to be a miss again**
  - Continually loading data into the cache but discard data (force out) before use it again
  - Nightmare for cache designer: Ping Pong Effect

Block Size Tradeoff Conclusions



Types of Cache Misses (1/2)

- **"Three Cs" Model of Misses**
- **1st C: Compulsory Misses**
  - occur when a program is first started
  - cache does not contain any of that program's data yet, so misses are bound to occur
  - can't be avoided easily, so won't focus on these in this course
  - Every block of memory will have one compulsory miss (NOT only every block of the cache)

Types of Cache Misses (2/2)

- **2nd C: Conflict Misses**
  - miss that occurs because two distinct memory addresses map to the same cache location
  - two blocks (which happen to map to the same location) can keep overwriting each other
  - big problem in direct-mapped caches
  - how do we lessen the effect of these?
- **Dealing with Conflict Misses**
  - Solution 1: Make the cache size bigger
    - Fails at some point
  - Solution 2: Multiple distinct blocks can fit in the same cache Index?

☑

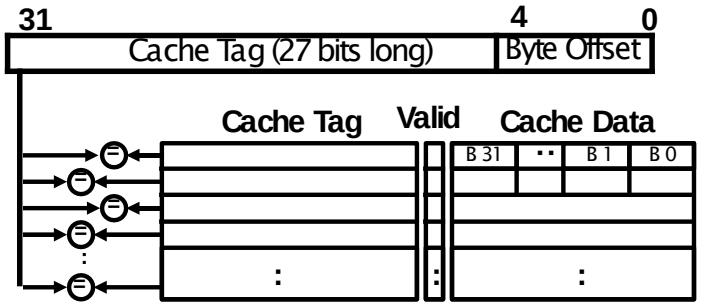
Fully Associative Caches

Fully Associative Cache (1/3)

- **Memory address fields:**
  - Tag: same as before
  - Offset: same as before
  - Index: non-existant
- **What does this mean?**
  - no "rows": any block can go anywhere in the cache
  - must compare with all tags in entire cache to see if data is there

Fully Associative Cache (2/3)

- **Fully Associative Cache (e.g., 32 B block)**
  - compare tags in parallel



Fully Associative Cache (3/3)

- **Benefit of Fully Assoc Cache**
  - No Conflict Misses (since data can go anywhere)
- **Drawbacks of Fully Assoc Cache**
  - Need hardware comparator for every single entry: if we have a 64KB of data in cache with 4B entries, we need 16K comparators: infeasible

Final Type of Cache Miss

- **3rd C: Capacity Misses**
  - miss that occurs because the cache has a limited size
  - miss that would not occur if we increase the size of the cache
  - sketchy definition, so just get the general idea
- **This is the primary type of miss for Fully Associative caches.**

How to categorize misses

- **Run an address trace against a set of caches:**
  - First, consider an infinite-size, fully-associative cache. For every miss that occurs now, consider it a compulsory miss.
  - Next, consider a finite-sized cache (of the size you want to examine) with full-associativity. Every miss that is not in #1 is a capacity miss.
  - Finally, consider a finite-size cache with finite- associativity. All of the remaining misses that are not #1 or #2 are conflict misses.
  - (Thanks to Prof. Kubiatowicz for the algorithm)

And in Conclusion…

1. **Divide into T I O bits, Go to Index = I, check valid**
   1. If 0, load block, set valid and tag (COMPULSORY MISS) and use offset to return the right chunk (1,2,4-bytes)
   2. If 1, check tag
      1. If Match (HIT), use offset to return the right chunk
      2. If not (CONFLICT MISS), load block, set valid and tag, use offset to return the right chunk