6.5930/1
Hardware Architectures for Deep Learning

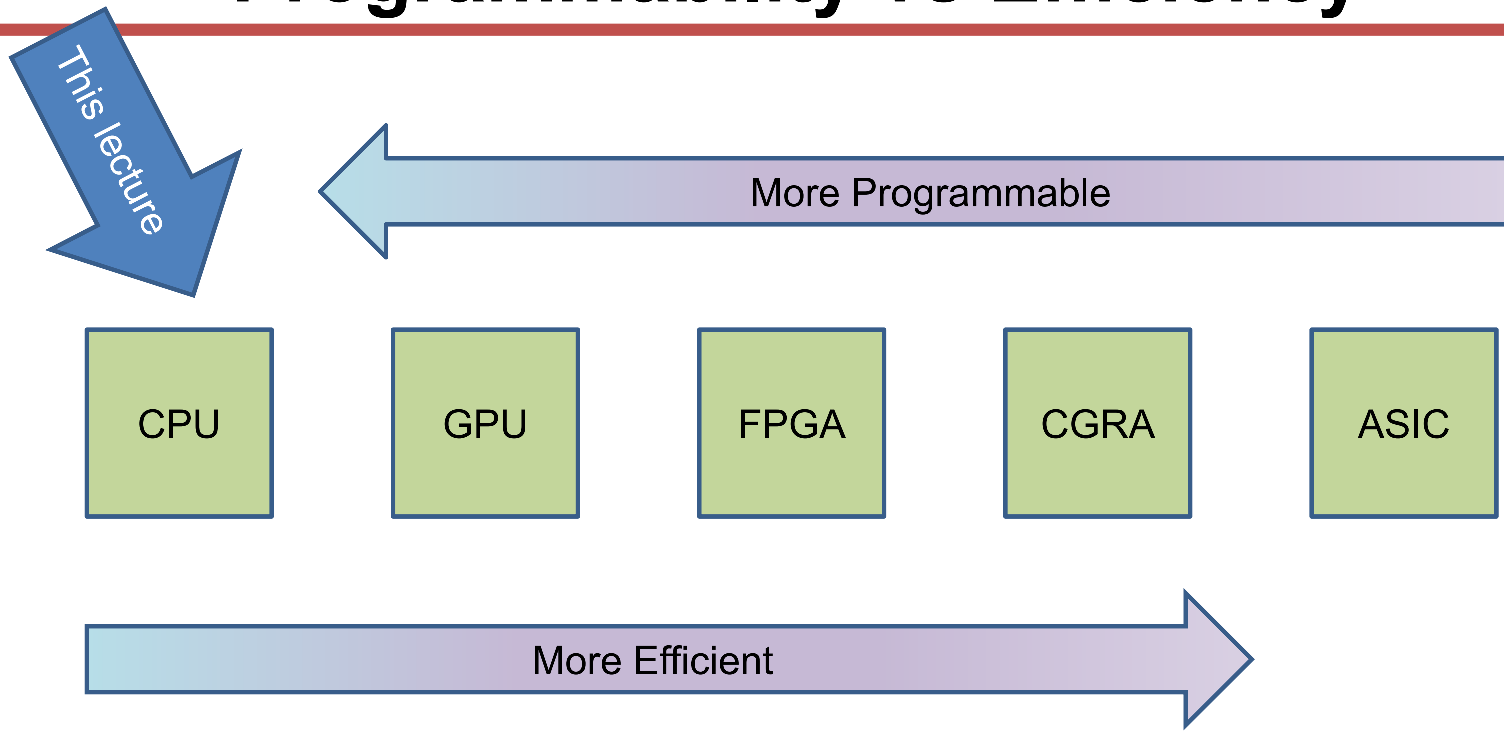
CPU Kernel Computation

February 20, 2024

Joel Emer and Vivienne Sze

Massachusetts Institute of Technology
Electrical Engineering & Computer Science

Programmability vs Efficiency



FPGA => Field programmable gate array
CGRA => Coarse-grained reconfigurable array
ASIC => Application-specific integrated circuit

Goals of Today's Lecture

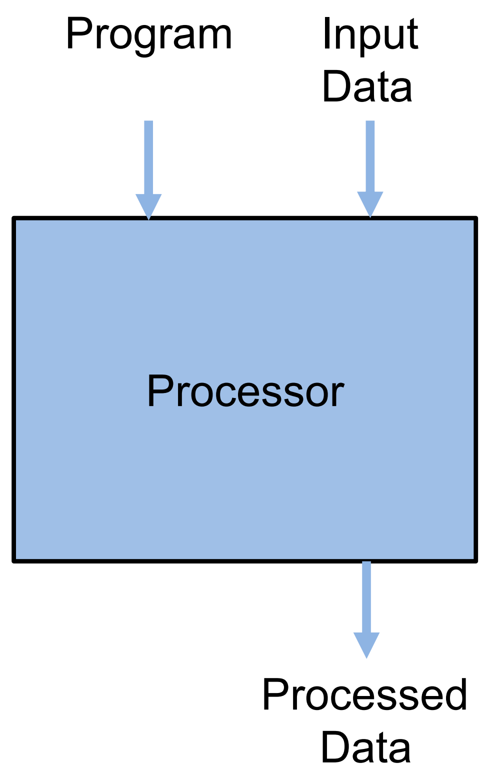
- Basic CPU architecture and computation model
- Example mapping of a deep learning algorithm on CPUs
- Program/compiler (software) optimizations
- Architectural (hardware) optimizations
- Define factors that affect performance (Iron Law)
- Provide basis for contrast with other compute approaches

Background Reading

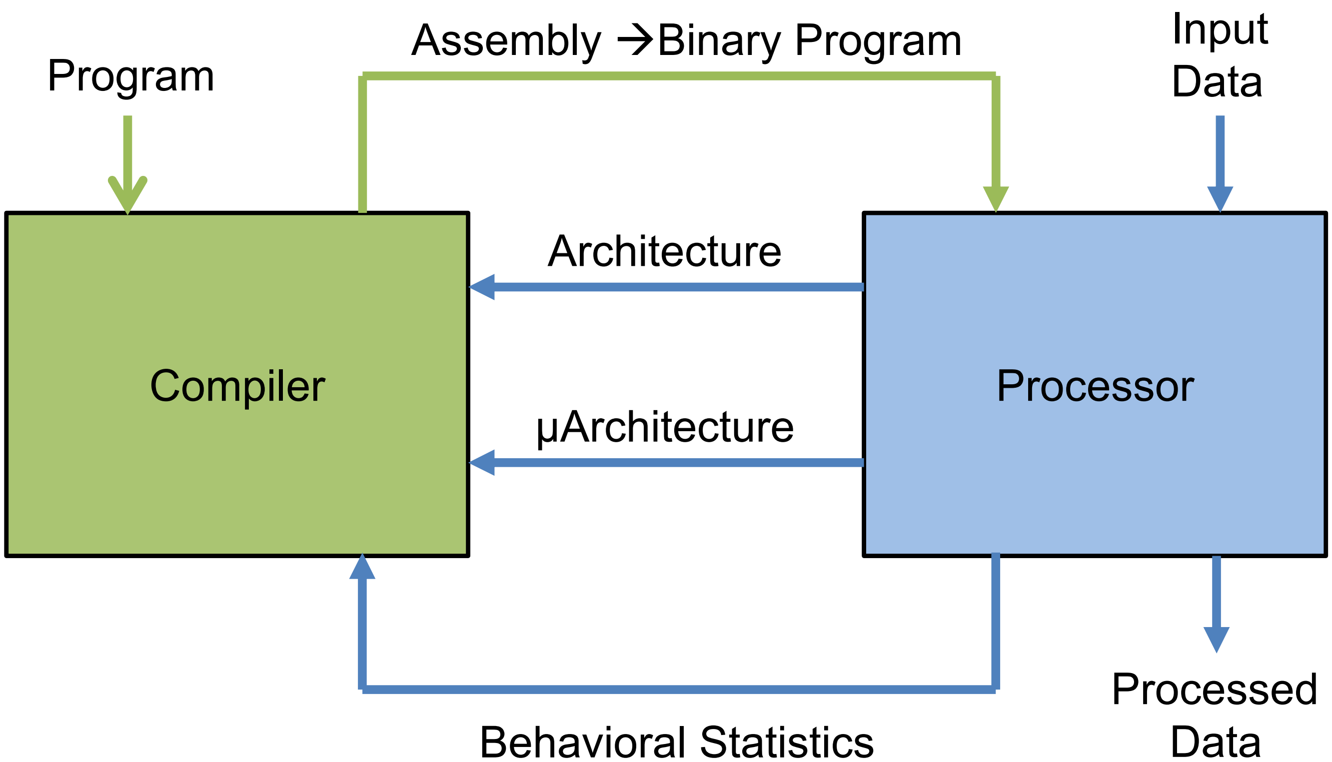
- **Instruction Pipelines and Hazards**
  - ***Computer Architecture: A Quantitative Approach*, by Hennessy and Patterson**
    - Edition 6: App C: p2-10
    - Edition 5: App C: p2-11
    - Edition 3&4: App A: p2-11
  - ***Computer Organization & Design, by Patterson and Hennessy***
    - *Chapter 6*

All these books and their online/e-book versions are available through MIT libraries.

CPU Compute Model



CPU Compute Model



CPU Architecture

- **State – User visible values**
  - **Registers – r0, r1, r2…**
  - **Memory – linearly addressable storage**
- **Instruction Set – Means of updating state**
  - **Compute**
  - **Memory access**
  - **Control**

Instruction Set

- Compute
  - add Rd, Rs, [Rt | const] - Rd = Rs + Rt
  - mul Rd, Rs, [Rt | const] - Rd = Rs * Rd
  - mv Rd, Rs - Rd = Rs

- Memory
  - ld Rd, offset(Rs) - Rd = Mem(Rs + offset)
  - st Rs, offset(Rt) - Mem(Rt + offset) = Rs

- Control
  - b<condition> Rs, Rt, dest - if (Rs <condition> Rt) goto dest

C to Assembly Language

Program (in C)

```
# Sum reduction

sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```

1000: A_0
1001: A_1
1002: A_2
1003: A_3
.
.
.

Assembly (composed of instructions)

```
       mv r1, 0            # r1 holds i
       mv r2, 0            # r2 hold sum
loop:  ld r3, a(r1)        # load a[i]
       add r2, r2, r3      # add into sum
       add r1, r1, 1       # increment i
       blt r1, 10, loop    # branch less than
       st r2, sum          # store result
```

Sum is updated in a register here

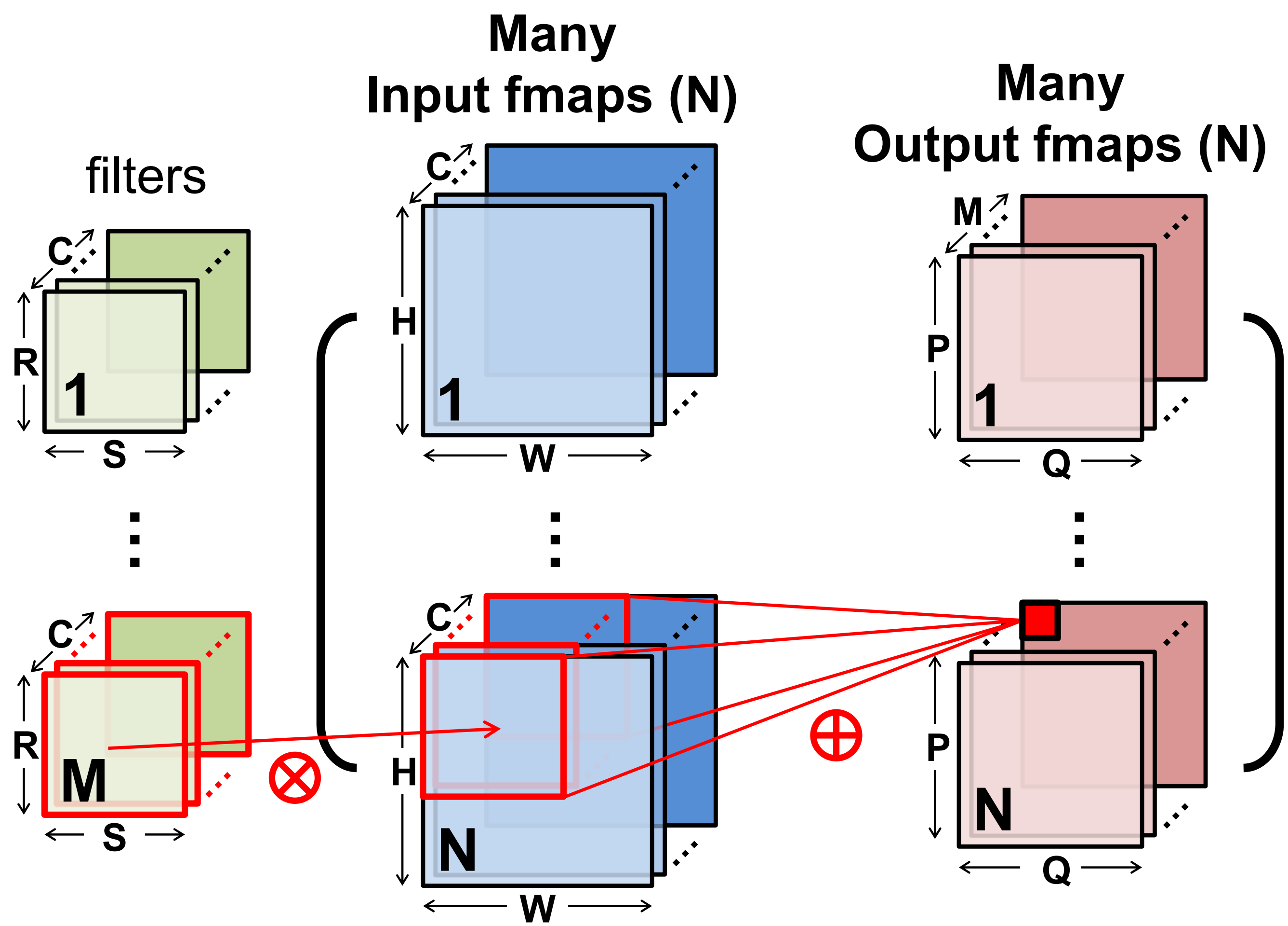
Iron Law of Performance

$$Performance = \frac{Cycles\_per\_Second}{Instructions\ * Cycles\_per\_Instruction}$$

- Instructions ~ architecture, program
- Cycles_per_instruction ~ micro-architecture
- Cycles_per_second ~ technology, circuit design

Software Optimizations

Convolution (CONV) Layer



Batch Size (N)

Einsum – Convolution

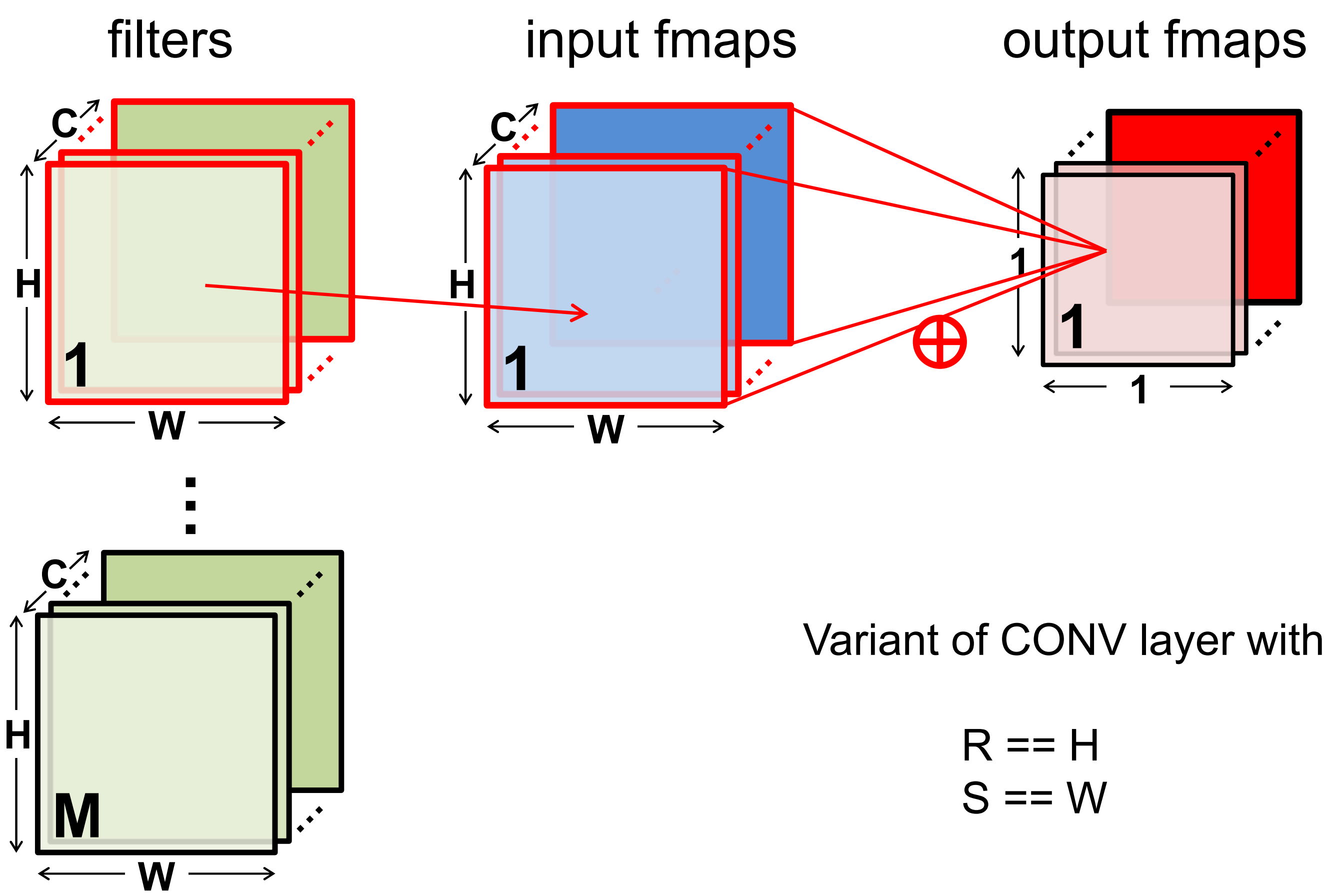
$$O_{n,m,p,q} = I_{n,c,Up+r,Uq+s} \times F_{m,c,r,s}$$

Operational Definition for Einsums (ODE):

- Traverse all points in space of all legal index values (iteration space)
- At each point in iteration space:
  - Calculate value on right hand at specified indices for each operand
  - Assign value to operand at specified indices on left hand side
  - Unless that operand is non-zero, then reduce value into it

[Relativity, Einstein, Annelen de Physik, 1916]
[Numpy/Einsum Python, ~2015]
[TACO, Kjolstad et.al., ASE 2017]
[Timeloop, Parashar et.al., ISPASS 2019]
[SAM, Hsu et.al., ASPLOS 2023]

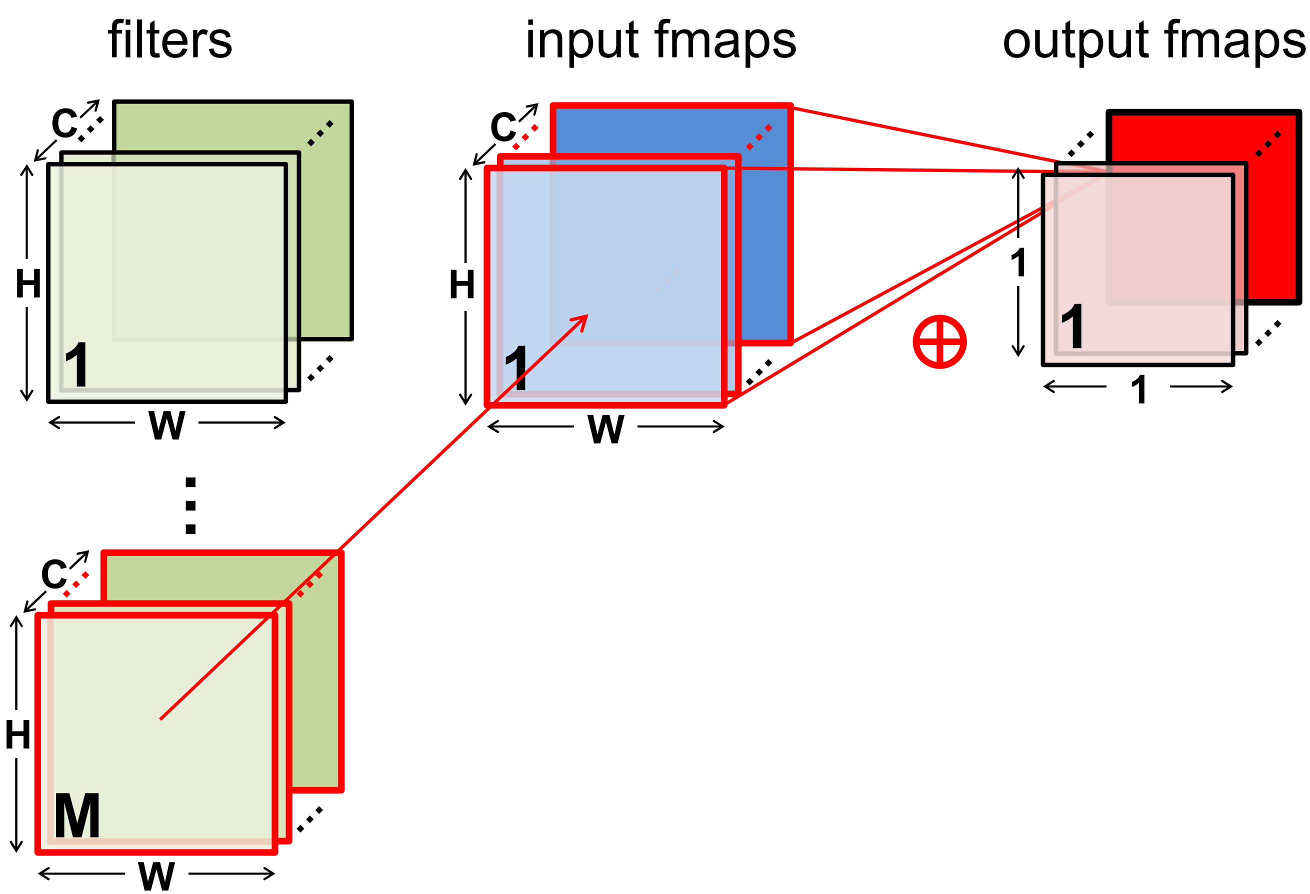
Fully Connected Computation



Variant of CONV layer with:

R == H
S == W

Fully Connected Computation



Einsum for FC

$$O_{n,m,p,q} = I_{n,c,Up+r,Uq+s} \times F_{m,c,r,s}$$

$$with\ U = 1, \mathrm{N} = 1$$

$$O_{m,p,q} = I_{c,p+r,q+s} \times F_{m,c,r,s}$$

$$with\ R = H, S = W$$

$$O_{m,p,q} = I_{c,p+h,q+w} \times F_{m,c,h,w}$$

$$note\ P = 1, Q = 1$$

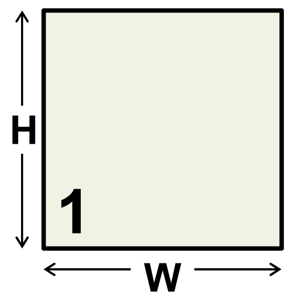
$$O_{m} = I_{c,h,w} \times F_{m,c,h,w}$$

Fully Connected Computation

```
int i[C][H][W];      # Input activations
int f[M][C][H][W];   # Filter Weights
int o[M];            # Output activations


for m in [0, M):
  o[m] = 0;
  for c in [0, C):
    for h in [0, H):
      for w in [0, W):
        o[m] += i[c][h][w]*f[m][c][h][w]
```

Should be bias, which we will ignore for simplicity

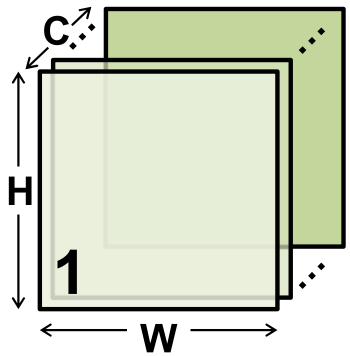
Filter Memory Layout



$f[H_0\,W_0]$ $f[H_0\,W_1]$ …
$f[H_1\,W_0]$ $f[H_1\,W_1]$ …
$f[H_2\,W_0]$ $f[H_2\,W_1]$ …
.
.
.
.

$f[H_h\,W_w]$ is at offset:
$h*W + w$

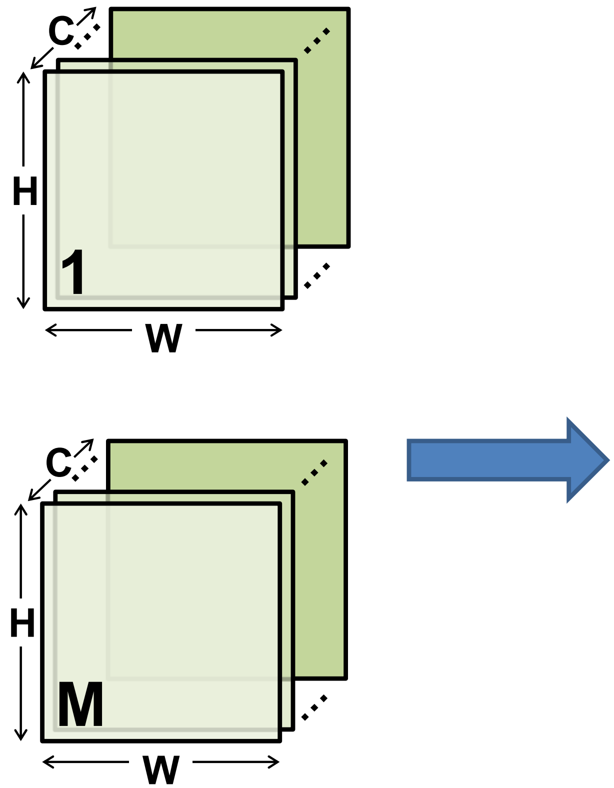
Filter Memory Layout



$f[C_0\ H_0\ W_0]$ $f[C_0\ H_0\ W_1]$ …
$f[C_0\ H_1\ W_0]$ $f[C_0\ H_1\ W_1]$ …
$f[C_0\ H_2\ W_0]$ $f[C_0\ H_2\ W_1]$ …
.
.
$f[C_1\ H_0\ W_0]$ $f[C_1\ H_0\ W_1]$ …
$f[C_1\ H_1\ W_0]$ $f[C_1\ H_1\ W_1]$ …
$f[C_1\ H_2\ W_0]$ $f[C_1\ H_2\ W_1]$ …
.
.
.

$f[C_c\ H_h\ W_w]$ is at offset:
$c*W*H + h*W + w$

Filter Memory Layout



$f[M_0\ C_0\ H_0\ W_0]$ $f[M_0\ C_0\ H_0\ W_1]$ …
$f[M_0\ C_0\ H_1\ W_0]$ $f[M_0\ C_0\ H_1\ W_1]$ …
$f[M_0\ C_0\ H_2\ W_0]$ $f[M_0\ C_0\ H_2\ W_1]$ …
.
.
$f[M_0\ C_1\ H_0\ W_0]$ $f[M_0\ C_1\ H_0\ W_1]$ …
$f[M_0\ C_1\ H_1\ W_0]$ $f[M_0\ C_1\ H_1\ W_1]$ …
$f[M_0\ C_1\ H_2\ W_0]$ $f[M_0\ C_1\ H_2\ W_1]$ …
.
.
.
$f[M_1\ C_0\ H_0\ W_0]$ $f[M_1\ C_0\ H_0\ W_1]$ …
$f[M_1\ C_0\ H_1\ W_0]$ $f[M_1\ C_0\ H_1\ W_1]$ …
$f[M_1\ C_0\ H_2\ W_0]$ $f[M_1\ C_0\ H_2\ W_1]$ …
.
.
.

$f[M_m\ C_c\ H_h\ W_w]$ is at offset:

$$m*C*H*W + c*W*H + h*W + w$$

Fully Connected Computation

```
int i[C][H][W];      # Input activations
int f[M][C][H][W];   # Filter Weights
int o[M];            # Output activations


for m in [0, M):
  o[m] = 0;
  for c in [0, C):
    for h in [0, H):
       for w in [0, W):
          o[m] += i[c][h][w]*f[m][c][h][w]
```

Flattened Filter Memory Indexing

$f[M_0\,C_0\,H_0\,W_0]$ $f[M_0\,C_0\,H_0\,W_1]$ …
$f[M_0\,C_0\,H_1\,W_0]$ $f[M_0\,C_0\,H_1\,W_1]$ …
$f[M_0\,C_0\,H_2\,W_0]$ $f[M_0\,C_0\,H_2\,W_1]$ …
.
.
$f[M_0\,C_1\,H_0\,W_0]$ $f[M_0\,C_1\,H_0\,W_1]$ …
$f[M_0\,C_1\,H_1\,W_0]$ $f[M_0\,C_1\,H_1\,W_1]$ …
$f[M_0\,C_1\,H_2\,W_0]$ $f[M_0\,C_1\,H_2\,W_1]$ …
.
.
.
$f[M_1\,C_0\,H_0\,W_0]$ $f[M_1\,C_0\,H_0\,W_1]$ …
$f[M_1\,C_0\,H_1\,W_0]$ $f[M_1\,C_0\,H_1\,W_1]$ …
$f[M_1\,C_0\,H_2\,W_0]$ $f[M_1\,C_0\,H_2\,W_1]$ …
.
.
.

f[0*C*H*W+0*W*H+0*W+0] f[0*C*H*W+0*W*H+0*W+1] …
f[0*C*H*W+0*W*H+1*W+0] f[0*C*H*W+0*W*H+1*W+1] …
f[0*C*H*W+0*W*H+2*W+0] f[0*C*H*W+0*W*H+2*W+1] …
.
.
f[0*C*H*W+1*W*H+0*W+0] f[0*C*H*W+1*W*H+0*W+1] …
f[0*C*H*W+1*W*H+1*W+0] f[0*C*H*W+1*W*H+1*W+1] …
f[0*C*H*W+1*W*H+2*W+0] f[0*C*H*W+1*W*H+2*W+1] …
.
.
.
f[1*C*H*W+0*W*H+0*W+0] f[1*C*H*W+0*W*H+0*W+1] …
f[1*C*H*W+0*W*H+1*W+0] f[1*C*H*W+0*W*H+1*W+1]] …
f[1*C*H*W+0*W*H+2*W+0] f[1*C*H*W+0*W*H+2*W+1] …
.
.
.

Fully Connected Computation

```
int i[C][H][W];      # Input activations
int f[M][C][H][W];   # Filter Weights
int o[M];            # Output activations


for m in [0, M):
  o[m] = 0;
  for c in [0, C):
    for h in [0, H):
      for w in [0, W):
        o[m] += i[c][h][w]*f[m][c][h][w]
```

Data Flattened FC Computation

```
int i[C*H*W];          # Input activations
int f[M*C*H*W];        # Filter Weights
int o[M];              # Output activations


for m in [0, M):
  o[m] = 0;
  for c in [0, C):
    for h in [0, H):
       for w in [0, W):
          o[m] += i[H*W*c + W*h + w]
                * f[C*H*W*m + H*W*c + W*h + w]
```

Iron Law of Performance

$$Performance = \frac{Cycles\_per\_Second}{Instructions * Cycles\_per\_Instruction}$$

- – Instructions ~ architecture, program
- – Cycles_per_instruction ~ micro-architecture
- – Cycles_per_second ~ technology, circuit design

Data Flattened FC Computation

```
int i[C*H*W];          # Input activations
int f[M*C*H*W];        # Filter Weights
int o[M];              # Output activations


for m in [0, M):
  o[m] = 0;
  for c in [0, C):
    for h in [0, H):
       for w in [0, W):
          o[m] += i[H*W*c + W*h + w]
                * f[C*H*W*m + H*W*c + W*h + w]
```

Lifting Loop Invariants

```
int i[C*H*W];         # Input activations
int f[M*C*H*W];       # Filter Weights
int o[M];             # Output activations

for m in [0, M):
  o[m] = 0;
  CHWm = C*H*W*m
  for c in [0, C):
    HWc = H*W*c;
    CHWm_HWc = CHWm + H*W*c
    for h in [0, H):
      HWc_Wh  = HWc + W*h
      CHWm_HWc_Wh = CHWm_HWc + W*h
       for w in [0, C):
          o[m] += i[HWc_Wh + w]
                * f[CHWm_HWc_Wh + w];
```

Index overhead is amortized over many inner loop iterations

Serial Traversal with Multiple Loops

```
for m in [0, M):
  o[m] = 0;
  CHWm = C*H*W*m
  for c in [0, C):
    HWc = H*W*c;
    CHWm_HWc = CHWm + H*W*c
    for h in [0, H):
      HWc_Wh  = HWc + W*h
      CHWm_HWc_Wh = CHWm_HWc + W*h
       for w in [0, C):
          o[m] += i[HWc_Wh + w]
                * f[CHWm_HWc_Wh + w]
```

I[C_0 H_0 W_0] I[C_0 H_0 W_1] …
I[C_0 H_1 W_0] I[C_0 H_1 W_1] …
I[C_0 H_2 W_0] I[C_0 H_2 W_1] …
.
.
I[C_1 H_0 W_0] I[C_1 H_0 W_1] …
I[C_1 H_1 W_0] I[C_1 H_1 W_1] …
I[C_1 H_2 W_0] I[C_1 H_2 W_1] …
.
.
.

F[M_0 C_0 H_0 W_0] F[M_0 C_0 H_0 W_1] …
F[M_0 C_0 H_1 W_0] F[M_0 C_0 H_1 W_1] …
F[M_0 C_0 H_2 W_0] F[M_0 C_0 H_2 W_1] …
.
.
F[M_0 C_1 H_0 W_0] F[M_0 C_1 H_0 W_1] …
F[M_0 C_1 H_1 W_0] F[M_0 C_1 H_1 W_1] …
F[M_0 C_1 H_2 W_0] F[M_0 C_1 H_2 W_1] …
.
.
.
F[M_1 C_0 H_0 W_0] F[M_1 C_0 H_0 W_1] …
F[M_1 C_0 H_1 W_0] F[M_1 C_0 H_1 W_1] …
F[M_1 C_0 H_2 W_0] F[M_1 C_0 H_2 W_1] …
.
.
.

Traversal Order

Tensor: f_MCHW[['M', 'C', 'H'], W]

Rank: W

Rank: ['M', 'C', 'H']	0	1	2	3	4	5
(0, 0, 0)	2	4	1	5	7	6
(0, 0, 1)	1	2	1	5	1	3
(0, 1, 0)	9	5	2	7	7	2
(0, 1, 1)	6	5	4	4	2	3
(1, 0, 0)	4	5	1	7	9	3
(1, 0, 1)	4	7	3	3	7	6
(1, 1, 0)	6	7	9	6	1	4
(1, 1, 1)	3	3	4	1	9	9
(2, 0, 0)	5	5	3	5	2	9
(2, 0, 1)	8	4	6	6	1	3
(2, 1, 0)	4	3	5	7	9	7
(2, 1, 1)	7	5	3	3	1	8

Tensor: i_CHW[['C', 'H'], W]

Rank: W

Rank: ['C', 'H']	0	1	2	3	4	5
(0, 0)	1	6	5	8	4	8
(0, 1)	6	6	2	6	4	5
(1, 0)	7	9	8	1	2	4
(1, 1)	6	9	3	1	9	9

Tensor: unknown[M]

Rank: M

0	1	2
0	0	0

Flattened Loops

```
int i[C*H*W];        # Input activations
int f[M*C*H*W];      # Filter Weights
int o[M];            # Output activations


for m in [0, M):
  o[m] = 0;
  CHWm = C*H*W*m;
  for chw in [0, C*H*W):
     o[m] += i[chw]
           * f[CHWm + chw];
  }
}
```

Recognizing serial access pattern dramatically reduces number of loop nests.

Optimization existed in first Fortran compiler by Backus in 1958

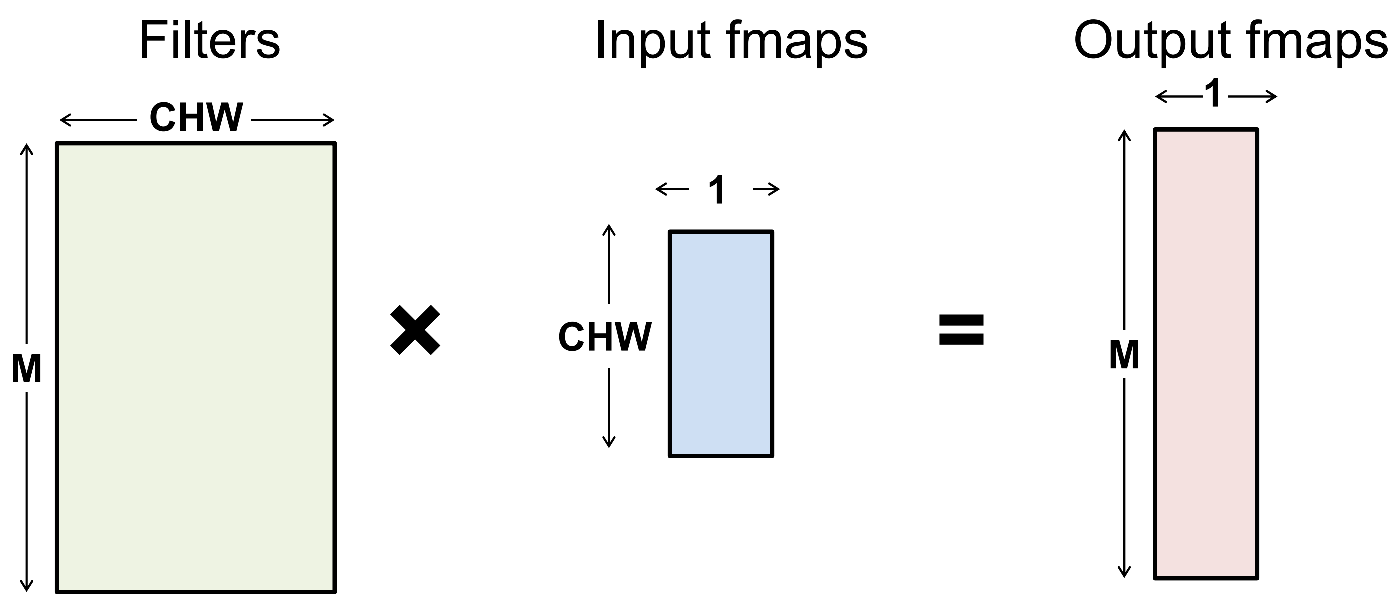
Flattened Assembly Language

```
        mv r1, 0                # r1 holds m
mloop:  mul r3, r1, C*H*W       # r3 holds m*CHW
        mv r2, 0                # r2 holds x
        mv r8, 0                # r8 holds psum (o[m])
xloop:  ld r4, i(r2)            # r4 = i[x]
        add r5, r2, r3
        ld r6, f(r5)            # r6 = w[CHWm + x]
        mul r7, r4, r6
        add r8, r7, r8          # r8 += i[x] * f[CHWm+x]
        add r2, r2, 1
        blt r2, C*W*H, xloop
        st r7, o(r1)            # store completed sum
        add r1, r1, 1
        blt r1, M, mloop
```

Partial sum held in register not memory

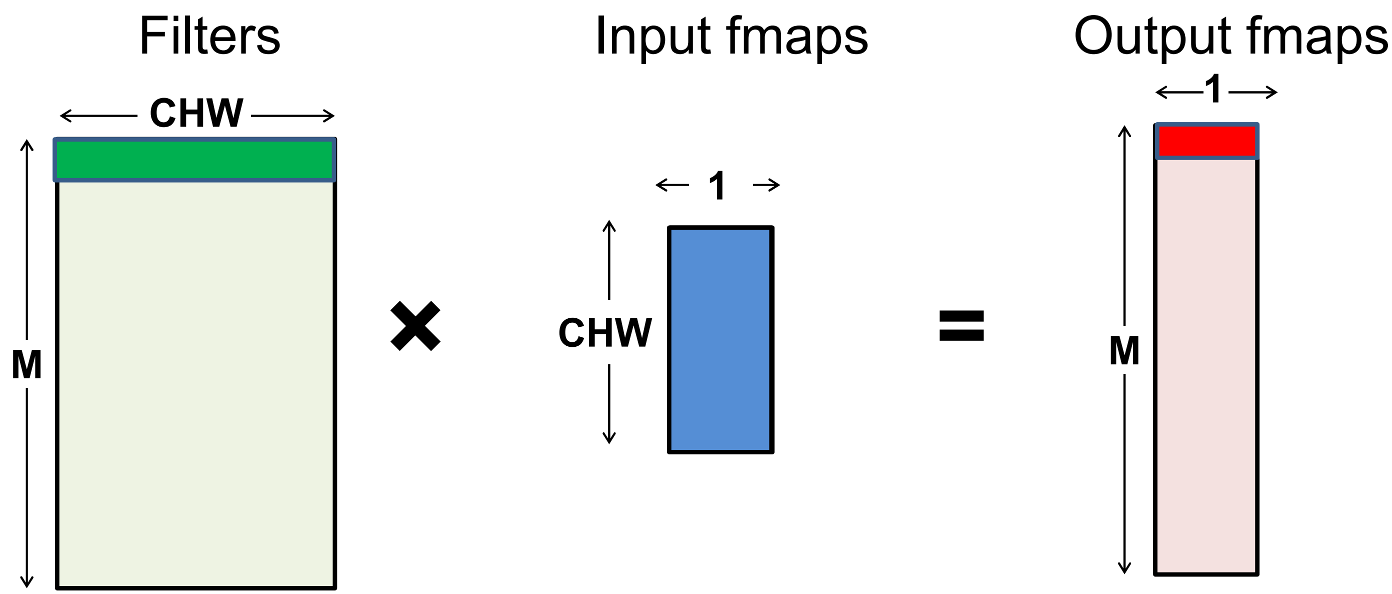
Fully-Connected (FC) Layer

- Matrix–Vector Multiply:
  - Multiply all inputs in all channels by a weight and sum



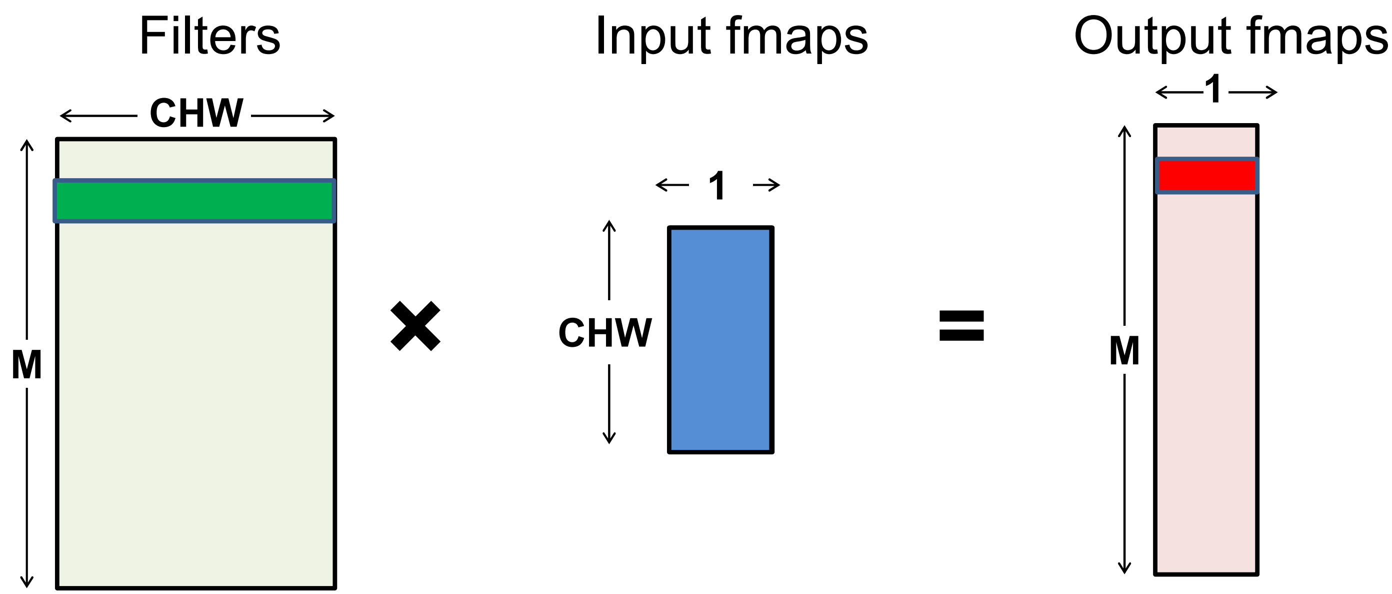
Fully-Connected (FC) Layer

- Matrix–Vector Multiply:
  - Multiply all inputs in all channels by a weight and sum



Fully-Connected (FC) Layer

- Matrix–Vector Multiply:
  - Multiply all inputs in all channels by a weight and sum



Einsum for flattened FC

$$I_{c,h,w} \rightarrow I_{H\times W\times c+W\times h+w} \rightarrow I_{chw}$$

$$F_{m,c,h,w} \rightarrow F_{m,H\times W\times c+W\times h+w} = F_{m,chw}$$

$$O_m = I_{c,h,w} \times F_{m,c,h,w}$$

$$\Downarrow$$

$$O_m = I_{chw} \times F_{m,chw}$$

Strength Reduction

```
for m in [0, M):
  o[m] = 0;
  CHWm = C*H*W*m;
  for chw in [0, C*H*W):
     o[m] += i[chw]
           * f[CHWm + chw]
```

Multiplies are expensive

```
CHWm = -C*H*W;
for m in [0, M):
  o[m] = 0;
  CHWm += C*H*W;
  for x in [0, C*H*W):
     o[m] += i[x]
           * f[CHWm + x]
```

Initialize offset

Exchange multiply for addition

Flattened Assembly Language

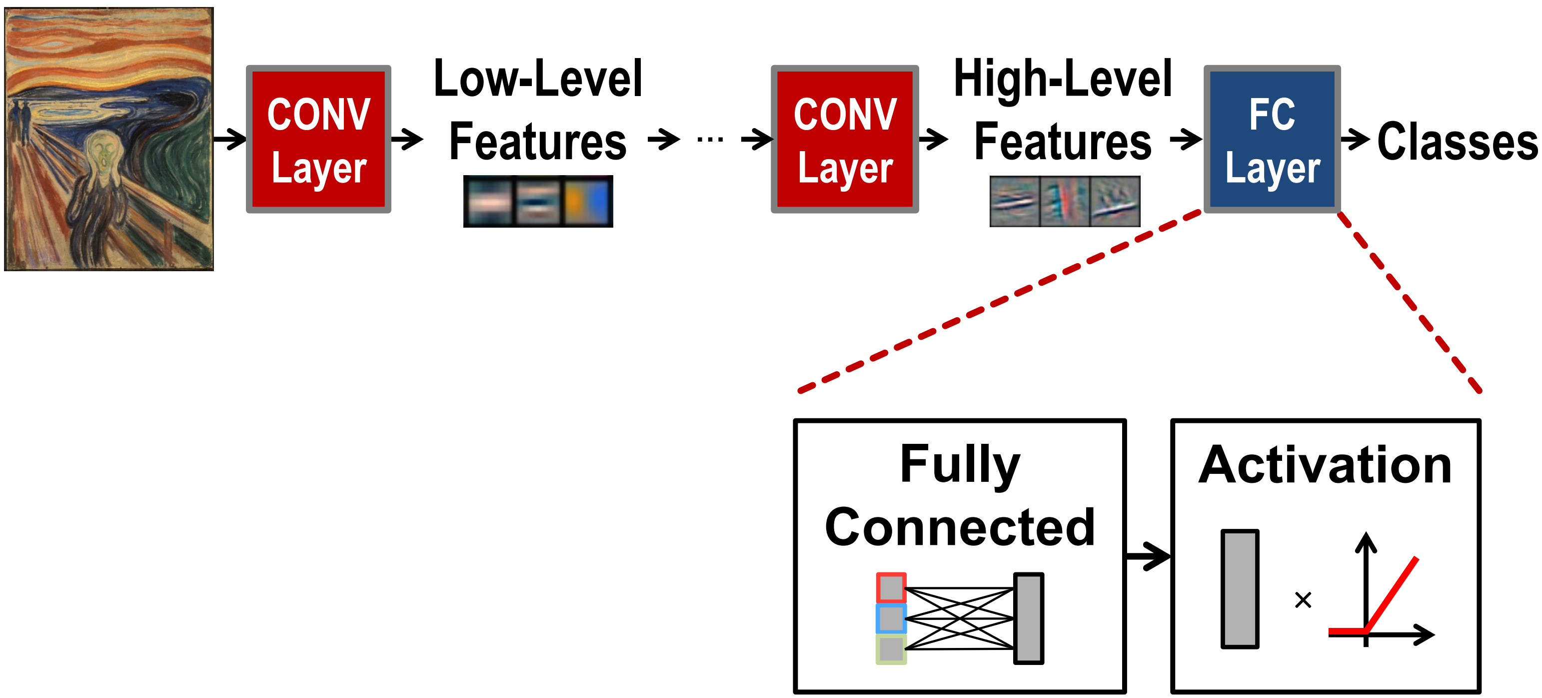
```
       mv r1, 0             # r1 holds m
mloop: mul r3, r1, C*H*W    # r3 holds m*CHW
       mv r2, 0             # r2 holds x
       mv r8, 0             # r8 holds psum (o[m])
xloop: ld r4, i(r2)         # r4 = i[x]
       add r5, r2, r3
       ld r6, f(r5)         # r6 = w[CHWm + x]
       mul r7, r4, r6
       add r8, r7, r8       # r8 += i[x] * f[CHWm+x]
       add r2, r2, 1
       blt r2, C*W*H, xloop
       st r7, o(r1)         # store completed sum
       add r1, r1, 1
       blt r1, M, mloop
```

Strength-Reduced Assembly Language

```
        mv r1, 0            # r1 holds m
        mv r3, -C*H*W       # r3 holds m*CHW
mloop:  add r3, r3, C*H*W
        mv r2, 0            # r2 holds x
        mv r8, 0            # r8 holds psum (o[m])
xloop:  ld r4, i(r2)        # r4 = i[x]
        add r5, r2, r3
        ld r6, f(r5)        # r6 = f[CHWm + x]
        mul r7, r4, r6
        add r8, r7, r8      # r8 += i[x] * f[CHWm+x]
        add r2, r2, 1
        blt r2, C*W*H, xloop
        st r7, o(r1)        # store completed sum
        add r1, r1, 1
        blt r1, M, mloop
```

Exchange multiply for addition

Convolutional Neural Networks



Separate Loops for FC and ReLU

```
int i[C*H*W];        # Input activations
int f[M*C*H*W];      # Filter Weights
int o[M];            # Output activations

CHWm = -C*H*W;
for m in [0, M):
  o[m] = 0;
  CHWm += C*H*W ;
  for x in [0, C*H*W):
     o[m] += i[x]
           * f[CHWm + x]
  }
}

for m in [0, M):
       o[m] = ReLU(o[m])
}
```

Every input to the ReLu operation was available as the final value o[m] for this loop.

Loop-Fused FC + ReLU

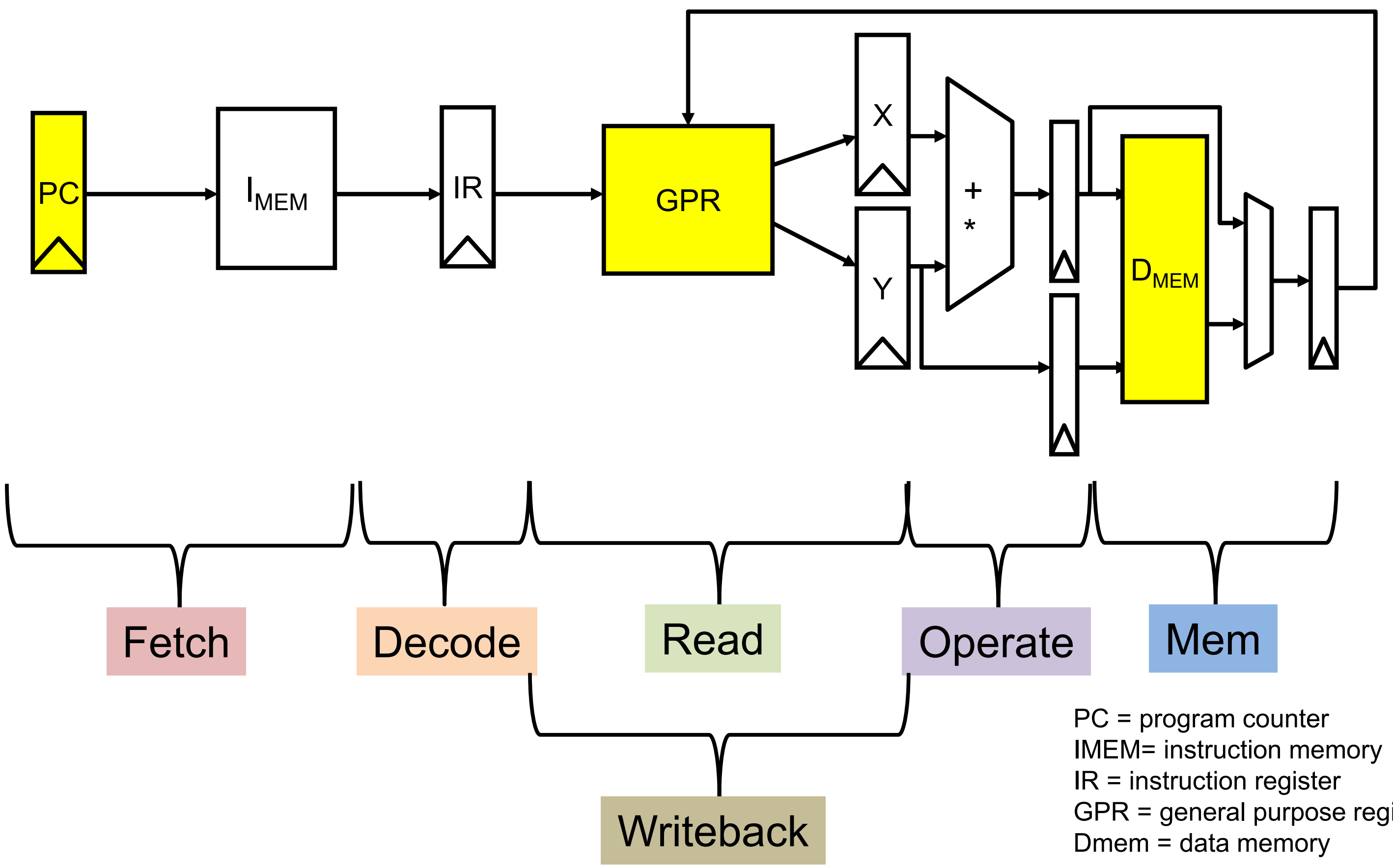
```
int i[C*H*W];          # Input activations
int f[M*C*H*W];        # Filter Weights
int o[M];              # Output activations

CHWm = -C*H*W;
for m in [0, M):
  o[m] = 0;
  CHWm += C*H*W;
  for x in [0, C*H*W):
     o[m] += i[x]
           * f[CHWm + x]

  o[m] = ReLU(o[m])
```

Hardware Optimizations

Steps of Execution

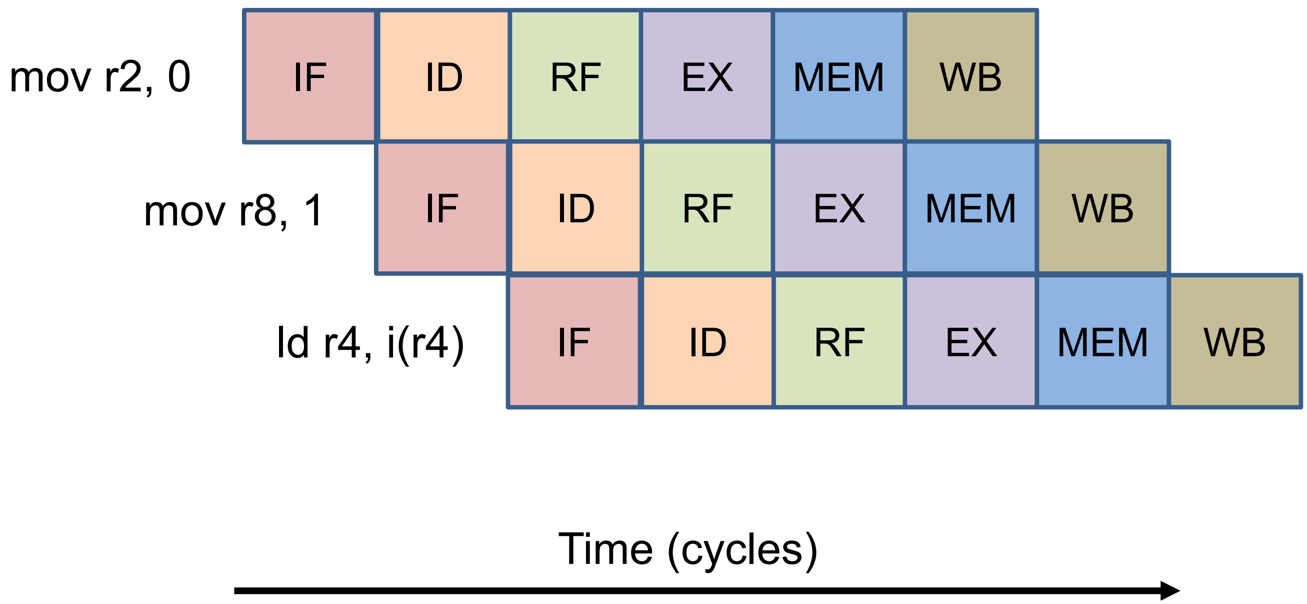
Fetch: Read in bits of instruction
Decode: Interpret bits of instruction
Register fetch: Read registers
Execute: Perform requested operation
Memory: Read or write memory (optional)
Writeback: Write result into a register

Simple Pipelined μArchitecture



PC = program counter
IMEM= instruction memory
IR = instruction register
GPR = general purpose register
Dmem = data memory

Instructions - Waterfall Diagram



Best case: Single cycle per instruction

Iron Law of Performance

$$Performance = \frac{Cycles\_per\_Second}{\color{red}{Instructions} * Cycles\_per\_Instruction}$$

- Instructions ~ architecture, program
- Cycles_per_instruction ~ micro-architecture
- Cycles_per_second ~ technology, circuit design

Multiply-Accumulate Overhead

```
        mv r1, 0                # r1 holds m
mloop:  mul r3, r1, C*H*W       # r3 holds m*CHW
        mv r2, 0                # r2 holds x
        mv r8, 0                # r8 holds psum (o[m])
xloop:  ld r4, i(r2)            # r4 = i[x]
        add r5, r2, r3
        ld r6, f(r5)            # r6 = f[CHWm + x]
        mul r7, r4, r6          # r7 = r4 * r6
        add r8, r7, r8          # r8 = r7 + r8
        add r2, r2, 1
        blt r2, C*W*H, xloop
        st r7, o(r1)            # store completed sum
        add r1, r1, 1
        blt r1, M, mloop
```

Multiply-Accumulate Overhead

```
        mv r1, 0                # r1 holds m
mloop:  mul r3, r1, C*H*W       # r3 holds m*CHW
        mv r2, 0                # r2 holds x
        mv r8, 0                # r8 holds psum (o[m])
xloop:  ld r4, i(r2)            # r4 = i[x]
        add r5, r2, r3
        ld r6, f(r5)            # r6 = f[CHWm + x]
        mac r8, r4,r6           # r8 += r4 * r6
        add r8, r7, r8
        add r2, r2, 1
        blt r2, C*W*H, xloop
        st r7, o(r1)            # store completed sum
        add r1, r1, 1
        blt r1, M, mloop
```

Issue-Rate Optimizations

- **Specialized instructions**
  - **E.g., multiply-accumulate (MAC)**
- **Superscalar**
  - **Fetch and execute multiple instructions at once**

Addresses Cycles per Instruction

Iron Law of Performance

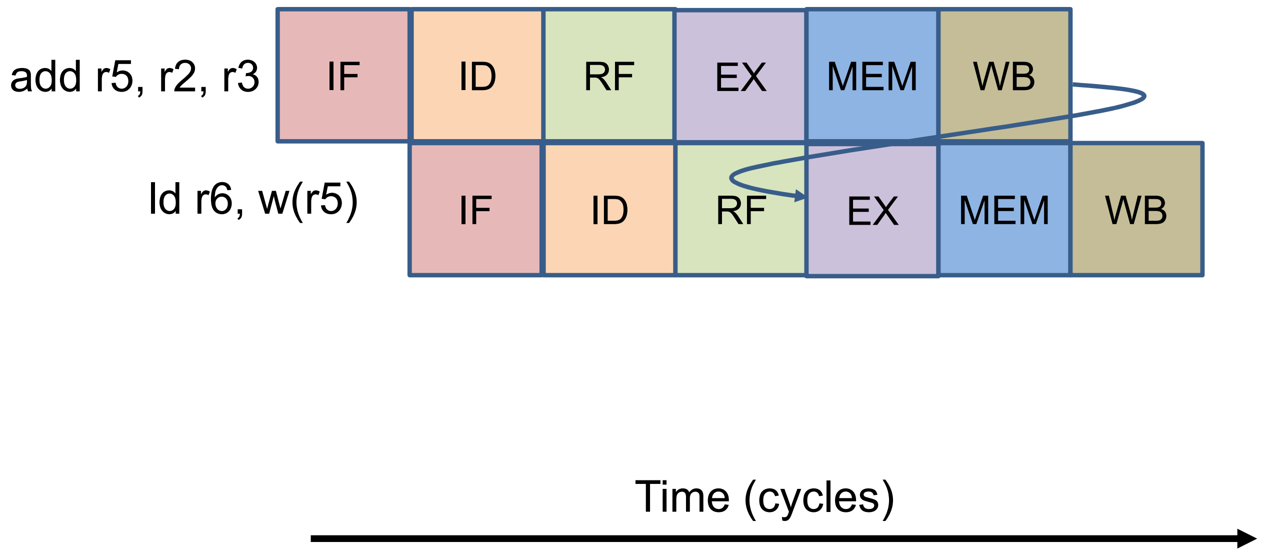
$$Performance = \frac{Cycles\_per\_Second}{Instructions * Cycles\_per\_Instruction}$$

- Instructions ~ architecture, program
- Cycles_per_instruction ~ micro-architecture
- Cycles_per_second ~ technology, circuit design

Load-Use Dependency

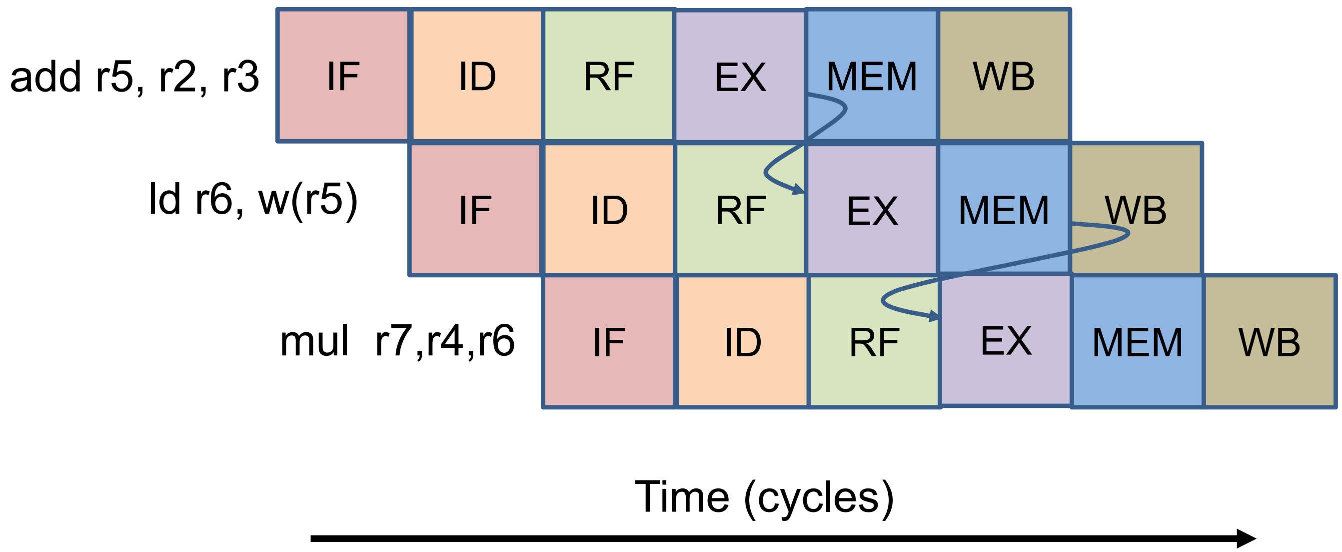
```
       mv r1, 0              # r1 holds m
mloop: mul r3, r1, C*H*W     # r3 holds m*CHW
       mv r2, 0              # r2 holds x
       mv r8, 0              # r8 holds psum (o[m])
xloop: ld r4, i(r2)          # r4 = i[x]
       add r5, r2, r3
       ld r6, f(r5)          # r6 = f[CHWm + x]
       mul r7, r4, r6
       add r8, r7, r8        # r8 += i[x] * w[CHWm+x]
       add r2, r2, 1
       blt r2, C*W*H, xloop
       st r7, o(r1)          # store completed sum
       add r1, r1, 1
       blt r1, M, mloop
```

Pipeline Dependencies



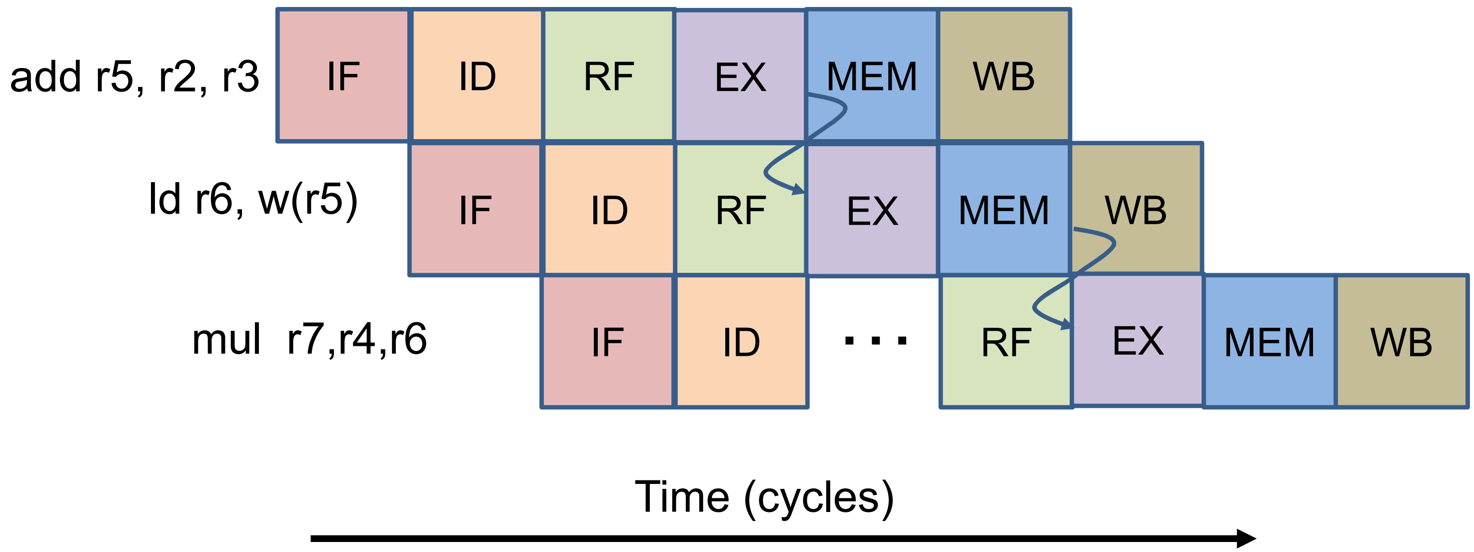
- Need r5 before it is written back

Pipeline Dependencies



- Need r5 before it is written back
- => “Bypass” it after it is available
- Need r6 before it is generated

Pipeline Dependencies



- Need r5 before it is written back
- => “Bypass” it after it is available
- Need r6 before it is generated
- => “Stall” and “bypass” when available

Dependencies - Optimizations

- Reorder code
  - Optimization by compiler or programmer

- Out-of-order execution
  - Complicated - take 6.823 for all the gory details

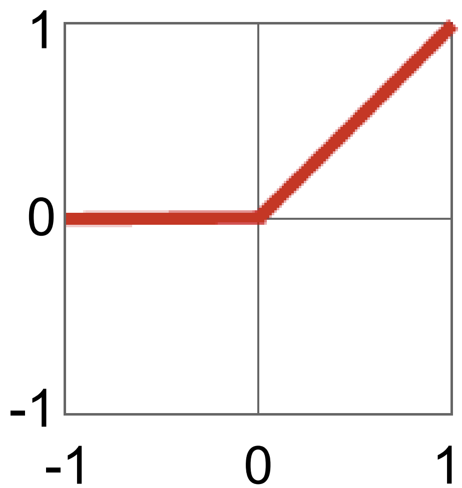
Iron Law of Performance

$$Performance = \frac{Cycles\_per\_Second}{Instructions * Cycles\_per\_Instruction}$$

- Instructions ~ architecture, program
- Cycles_per_instruction ~ micro-architecture
- Cycles_per_second ~ technology, circuit design

Implementing ReLU

Rectified Linear Unit (ReLU)

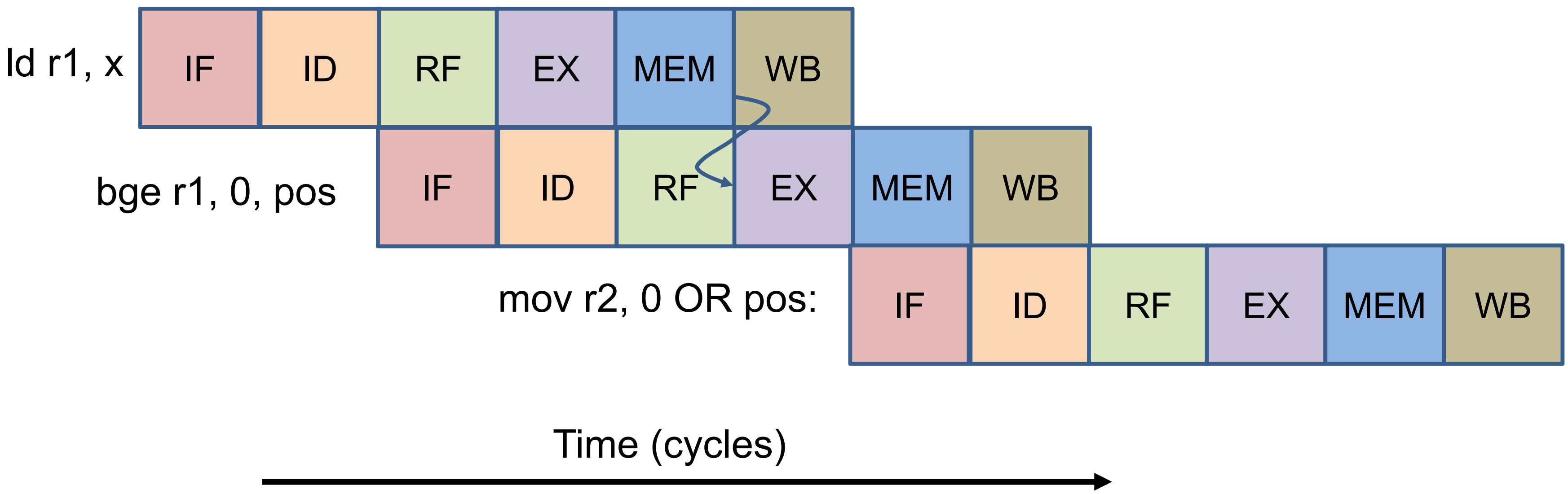


$y=max(0,x)$

```
y = (x<0)?0:x;
```

```
        ld r1, x
        bge r1, 0, pos
        mov r1, 0
pos:    ...
```

Waterfall Diagram – w/branch



Branches result in a long dependency, and branch prediction will be of little use

Conditional Move

- **Conditional move (CMOV)**
  - **cmov<condition> Rd, Rs, Rt** **- if (<condition> Rs)**
    Rd = Rt

```
        ld r1, x
        bge r1, 0, pos
        mov r1, 0
pos:
```

Target of branch

```
ld r1, x
cmovlt r1, r1, 0
```

Turns data dependent control into datapath calculation!

Loop-Fused FC + ReLU

```
        mv r1, 0              # r1 holds m
        mv r3, -C*H*W         # r3 holds m*CHW
mloop:  add r3, r3, C*H*W
        mv r2, 0              # r2 holds x
        mv r8, 0              # r8 holds psum (o[m])
xloop:  ld r4, i(r2)          # r4 = i[x]
        add r5, r2, r3
        ld r6, w(r5)          # r6 = w[CHWm + x]
        mul r7, r4, r6
        add r8, r7, r8        # r8 += i[x] * w[CHWm+x]
        add r2, r2, 1
        blt r2, C*W*H, xloop
        cmovlt r7, r7, 0      # ReLU on r7
        st r7, o(r1)          # store completed sum
        add r1, r1, 1
        blt r1, M, mloop
```

Summary

- **CPU Programmer/Compiler Optimizations [Software]**
  - **Lifting loop invariants**
  - **Flattening**
  - **Strength reduction**
  - **Loop fusing**

- **CPU Architecture Optimizations [Hardware]**
  - **Pipelining**
  - **Bypassing (and more sophisticated things c.f. 6.823)**
  - **Fused arithmetic (MAC)**
  - **Branch conversion (CMOV)**

- **Evaluation (via Iron Law)**

Next Lecture: Memory

Thank you!