AOC 2023-2024

Introduction / rappel des concepts nécessaires

Présenté par Emmanuel OSERET

Plan

- Mesure
- Optimisation
- Maîtrise environnement (Linux)
- Compilateurs : installation Intel oneAPI / AMD AOCC
- MAQAO
- OpenMP

Mesure et optimisation

- Optimiser une région de code nécessite :
  - de mesurer la métrique à optimiser
  - avec précision et fiabilité (évaluation stabilité)

- Ici : optimiser = réduire temps d'exécution

Principaux obstacles

- Précision du timer
  - RDTSC : précis à quelques cycles près mais biais de 20-30 cycles (amorti à partir de ~500 cycles)
  - Compteurs matériels : plusieurs secondes nécessaires
- Etat initial de la machine
  - Régime transitoire : C-states/DVFS, caches et prefetchers “froids”, pipeline non rempli
- Stabilité (variation d'une mesure à l'autre)
  - Pollution par système et autres applis

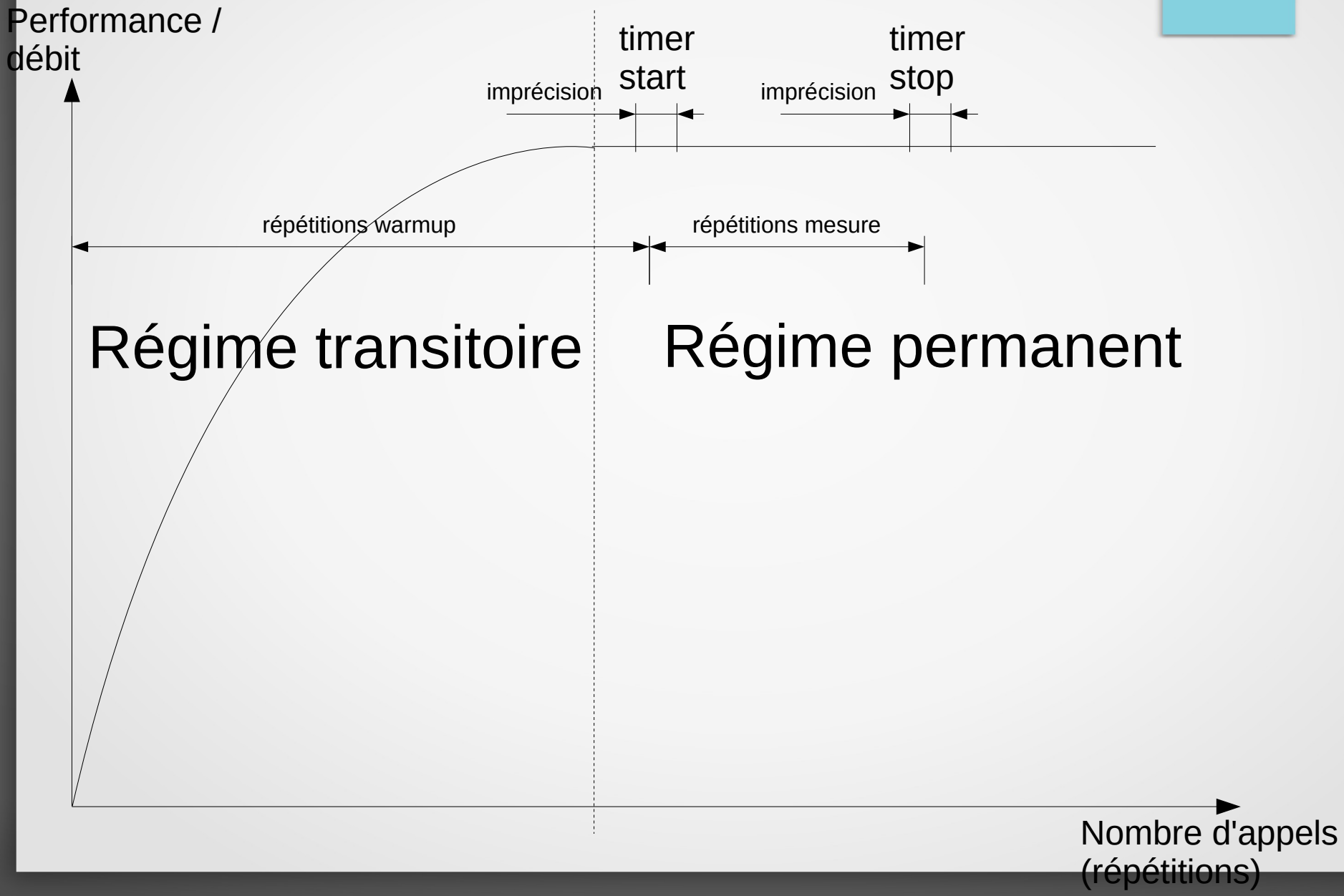
Solutions
Répétitions

- **Répétitions du noyau**

- **Avant** la mesure :
  - Pour exclure le régime transitoire
  - Appelé « warmup »
  - Surtout utile machine froide

- **Pendant** la mesure :
  - Pour amortir l'erreur du timer

Solutions aux obstacles à la précision
Répétitions



Speedup or not speedup ?

On a deux codes, lequel est optimal ?

Code 1 : 8.56 secondes
Code 2 : 8.94 secondes

Speed-up : (8.94-8.56) / 8.94 = 4.2 %

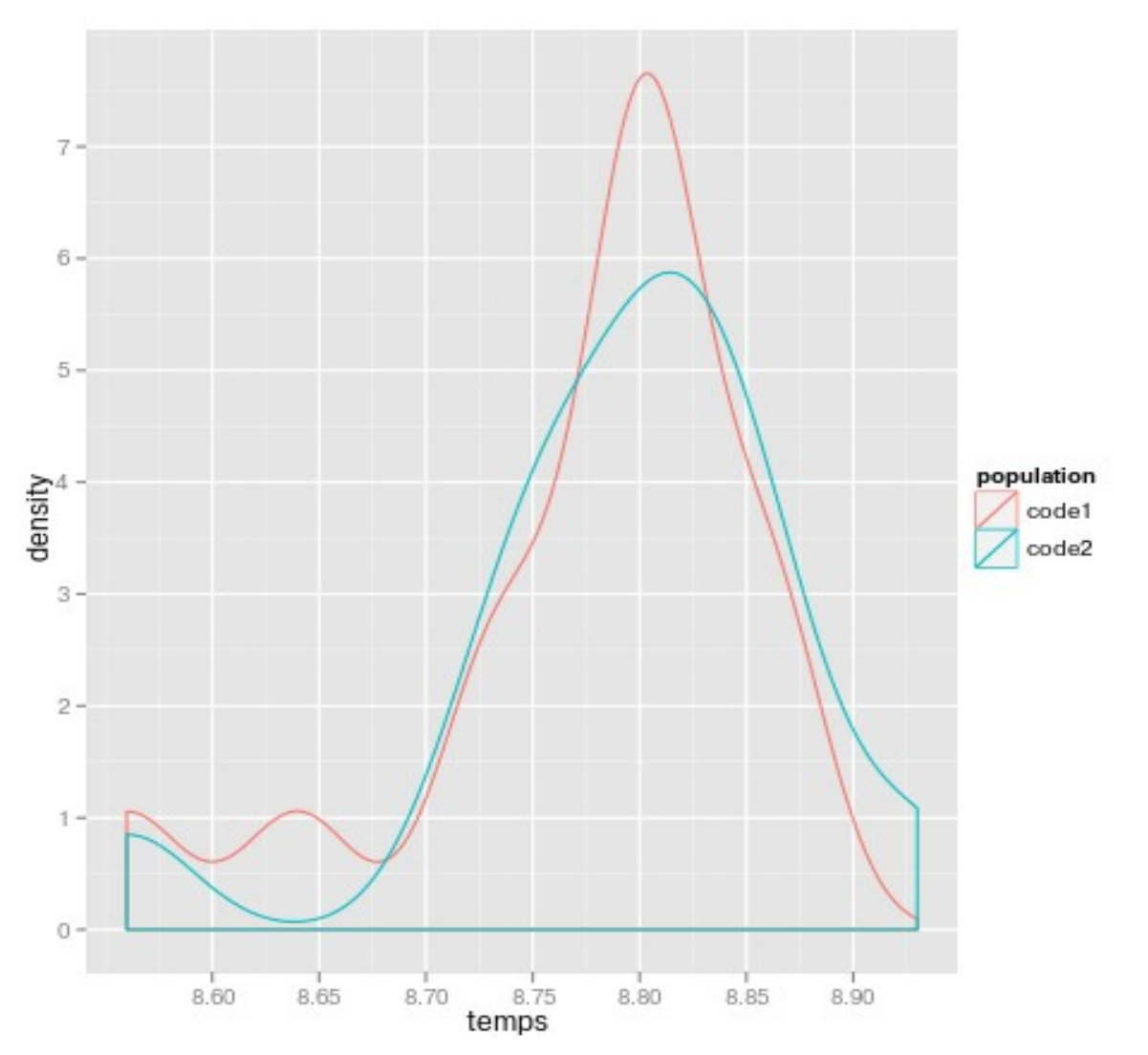
Vrai ?

Speedup or not speedup ?

Faux !

Avec 30 mesures pour chaque code, on voit que la différence n'est pas significative.

→ META-RÉPÉT. (~ 30)



Solutions
Méta-répétitions et environnement d'exé.

- Méta-répétitions pour mesurer la stabilité
  - Répétitions du corps du driver
  - Via un script ou une boucle dans le driver
  - En toute rigueur, 31 méta-répétitions nécessaires
  - Souhaité : (médiane - minimum) / minimum < 5 %

- Environnement d'exécution le plus léger possible
  - Le plus simple et efficace : redémarrer en mode *failsafe, rescue...* (console uniquement, pas de réseau, mono-utilisateur)

Solutions
Solutions avancées

- Fixer processus sur 1 coeur (process pinning)
  - `taskset -c`
  - Appel POSIX
- Exécution native (et non via machine virtuelle)
- Désactiver « turbo » (voire fixer la fréquence)

Initialisation tableaux

- Politique first touch
  - Données résidentes en mem. qu'à la 1ère écriture
  - Lectures précédentes « dans le vide » (page de 0)
  - `malloc` n'initialise pas et `calloc` pointe sur page de 0
- Valeurs
  - Validation résultats
    - valeurs aléatoires mais reproductibles (**`srand (42)`**)
  - Eviter exceptions flottantes
    - nombres dénormalisés => valeurs de préférence entre 0 et 1
    - ne pas diviser par 0

Concrètement (avec script) !

```
int main (int argc, char *argv[])
{
  int r;

  /* Récupération arguments */
  int size = atoi (argv[1]);
  int repw = atoi (argv[2]);
  int repm = atoi (argv[3]);

  /* Allocation */
  float *a = malloc (...);
  float *b = malloc (...);

  /* Init. tableaux lus */
  init_array (size, b);

  /* Warmup */
  for (r=0; r<repw; r++)
    kernel (size, a, b);

  /* Répétitions */
  uint64_t t1 = rdtsc();
  for (r=0; r<repm; r++)
    kernel (size, a, b);
  uint64_t t2 = rdtsc();

  /* Libération */
  free (a);
  free (b);

  /* Affichage performance */
  printf ("Cycles per iter.: %.2f\n",
(float) (t2 - t1) / (size * repm));

  return 0;
}
```

```
#!/bin/bash

# Meta-repetitions
for i in $(seq 1 31)
do
  ./run_kernel 1000 100 10
done
```

Concrètement (sans script) !

```
int main (int argc, char *argv[])
{
  int m, r;

  /* Récupération arguments */
  int nmet = atoi (argv[1]);
  int size = atoi (argv[2]);
  int repw = atoi (argv[3]);
  int repm = atoi (argv[4]);

  for (m=0; m<nmet; m++) {
    /* Allocation */
    float *a = malloc (...);
    float *b = malloc (...);

    /* Init. tableaux lus */
    init_array (size, b);

    /* Warmup */
    for (r=0; r<repw; r++)
        kernel (size, a, b);
```

```
    /* Répétitions */
    uint64_t t1 = rdtsc();
    for (r=0; r<repm; r++)
        kernel (size, a, b);
    uint64_t t2 = rdtsc();

    /* Libération */
    free (a);
    free (b);

    /* Affichage performance */
    printf (“Cycles per iter.: %.2f\
n”, (float) (t2 - t1) / (size *
repm));
  } // for m

  return 0;
}
```

Optimisation

- Exploitation parallélisme à tous les niveaux
- Exploitation localité spatiale et temporelle

Exploitation du parallélisme

- ILP
  - Exécution simultanée de plusieurs itérations consécutives d'une boucle
  - Optimisation typique : déroulage d'une réduction avec sommes partielles
- SIMD
  - Vectorisation (compilateur)
- Multi-coeurs
  - OpenMP (multi-threads)
  - MPI (multi-process)

Exploitation de la localité spatiale

- Localité spatiale : accès suivants à des adresses proches (groupement des données)
- Matériel
  - Eléments accédés par blocs (lignes de cache)
  - Chargement anticipé de ces blocs (prefetch)
- Optimisation
  - Pas unitaire (ce qui facilite aussi la vectorisation)
  - Permutation de boucles, SoA-AoS etc.

Exploitation de la localité temporelle

- Localité temporelle : accès répétés dans un futur proche (réutilisation des données)
- Matériel : caches
  - Mémoires plus petites mais plus rapides
  - Plusieurs niveaux
  - Pour instructions et données
- Optimisation
  - Fusion de boucle
  - Déroulage de boucle
  - Loop blocking

Boucle idéale

- Pas de dépendances inter-itérations (ILP)
- Pas de structures conditionnelles
- Pas unitaire (vectorisation, localité spatiale)
- Tiens dans le cache de données (localité tempo)
- Beaucoup plus de calculs que d'accès mémoire
  - Elimination bottleneck mémoire
- Ex typique : SGEMM / DGEMM

Transformation de boucles

- Transformations souvent effectuées automatiquement par le compilateur.
- Dans certains cas, le compilateur n'arrive pas à trouver la transformation.

- On ne donne que quelques exemples.
- Bibliographie : Survey of Loop Transformations Techniques, Eric LaForest 2010

Loop-invariant code motion

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    a[i][j] = c[i] + b[i][j] ;
```

Comment optimiser ce code ?

Loop-invariant code motion

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      a[i][j] = c[i] + b[i][j] ;
```

```
for (i=0; i<n; i++) {
   K = c[i];
   for (j=0; j<n; j++)
      a[i][j] = K + b[i][j] ;
}
```

If Hoisting

```
for (i=1; i<n; i++) {
  a[i] = a[i] + 1 ;
  if (i<3)
    b[i] = a[i] * c[i] ;
  else
    b[i] = a[i-1] * b[i-1] ;
}
```

Comment optimiser ce code ?

If Hoisting

```
for (i=1; i<n; i++) {
   a[i] = a[i] + 1 ;
   if (i<3)
      b[i] = a[i] * c[i] ;
   else
      b[i] = a[i-1] * b[i-1] ;
}
```

```
for (i=1; i<3; i++) {
   a[i] = a[i] + 1 ;
   b[i] = a[i] * c[i] ;
}
for (i=3; i<n; i++) {
   a[i] = a[i] + 1 ;
   b[i] = a[i-1] * b[i-1] ;
}
```

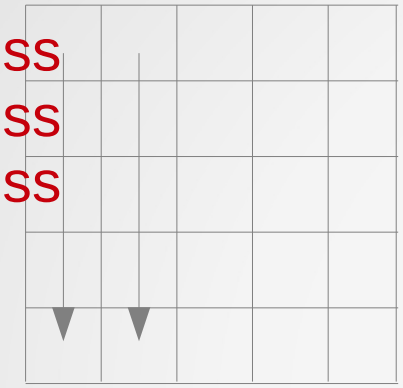
Loop interchange

```
for (i=1; i<n; i++)
  for (j=1; j<n; j++)
    a[j][i] = a[j][i] * a[j][i] ;
```

Comment optimiser ce code ?

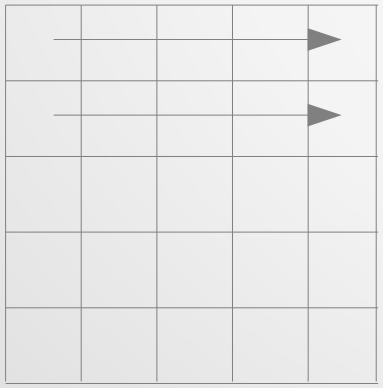
Loop interchange

Miss
Miss
Miss
...



```
for (i=1; i<n; i++)
  for (j=1; j<n; j++)
    a[j][i] = a[j][i] * a[j][i] ;
```

Miss Hit Hit ...



```
for (j=1; j<n; j++)
  for (i=1; i<n; i++)
    a[j][i] = a[j][i] * a[j][i] ;
```

Loop fusion / fission

Avantages :
Localité des accès a[i]

Moins de contrôle.

```
for (i=0; i<n; i++) {
   a[i] = a[i] + c ;
   x[i+1] = x[i]*7 + x[i+1] +
a[i] ;
}
```

Avantages :
Diminue la pression sur les registres.

Peut éviter des saturations des ports.

```
for (i=0; i<n; i++)
   a[i] = a[i] + c ;

for (i=0; i<n; i++)
   x[i+1] = x[i]*7 + x[i+1] +
a[i] ;
```


Comment choisir : Il faut mesurer, ça dépend de l'architecture !

Loop peeling

```
for (i=0; i<n; i++)
  a[i] = a[i] + 2 ;

for (i=2; i<n; i++)
  a[i] = a[i] * 2 ;
```

Comment optimiser ce code ?

Loop peeling

```
for (i=0; i<n; i++)
   a[i] = a[i] + 2 ;

for (i=2; i<n; i++)
   a[i] = a[i] * 2 ;
```

Peeling

```
a[0] = a[0] + 2 ;
a[1] = a[1] + 2 ;
for (i=2; i<n; i++)
   a[i] = a[i] + 2 ;

for (i=2; i<n; i++)
   a[i] = a[i] * 2 ;
```

Fusion

```
a[0] = a[0] + 2 ;
a[1] = a[1] + 2 ;
for (i=2; i<n; i++)
   a[i] = (a[i] + 2) * 2 ;
```

Loop unrolling

```
for (i=0; i<n; i++)
   c[i] += b[i] + a[i] ;
```

Comment optimiser ce code ?

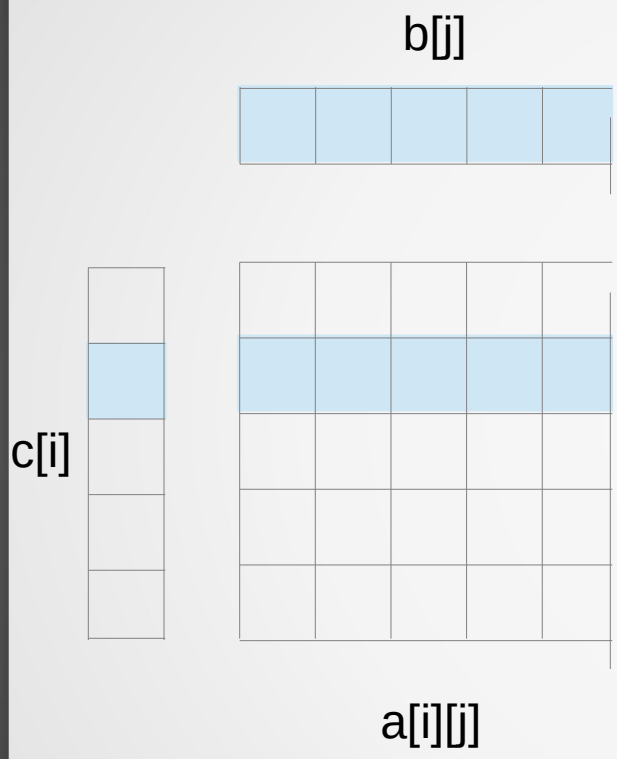
Loop unrolling

```
for (i=0; i<n; i++)
  c[i] += b[i] + a[i] ;
```

```
for (i=0; i<n-1; i+=2) {
  c[i+0] += b[i+0] + a[i+0];
  c[i+1] += b[i+1] + a[i+1];
}
if (n%2 == 1)
  c[n-1] += b[n-1] + a[n-1];
```

Réduit le surcoût de contrôle.
Pour la vectorisation automatique il vaut mieux ne pas dérouler
Parfois, fait apparaître des simplifications.

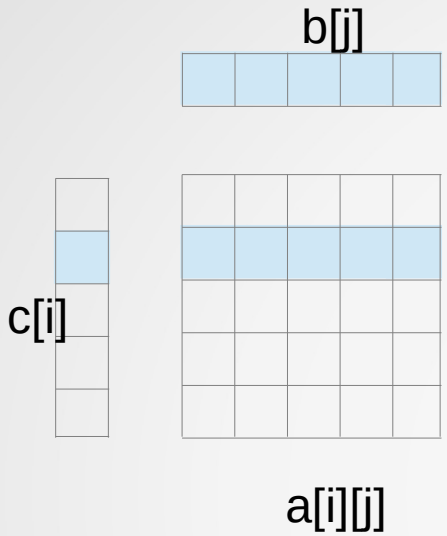
Loop tiling



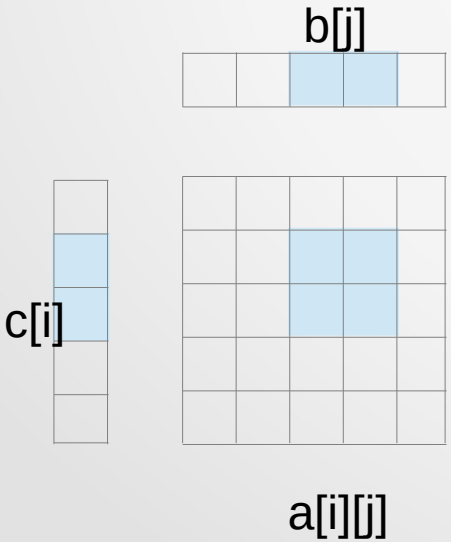

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    c[i] += a[i][j] * b[j];
```

Comment optimiser ce code lorsque n très grand ?

Loop tiling




```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    c[i] += a[i][j] * b[j];
```




```
#define TILE 2
assert(n%TILE == 0)
for (ii=0; ii<n; ii+=TILE)
for (jj=0; jj<n; jj+=TILE)
  for(i=ii; i<ii+TILE; i++)
    for(j=jj; j<jj+TILE; j++)
      c[i] += a[i][j] * b[j];
```

Compilateurs : Intel oneAPI

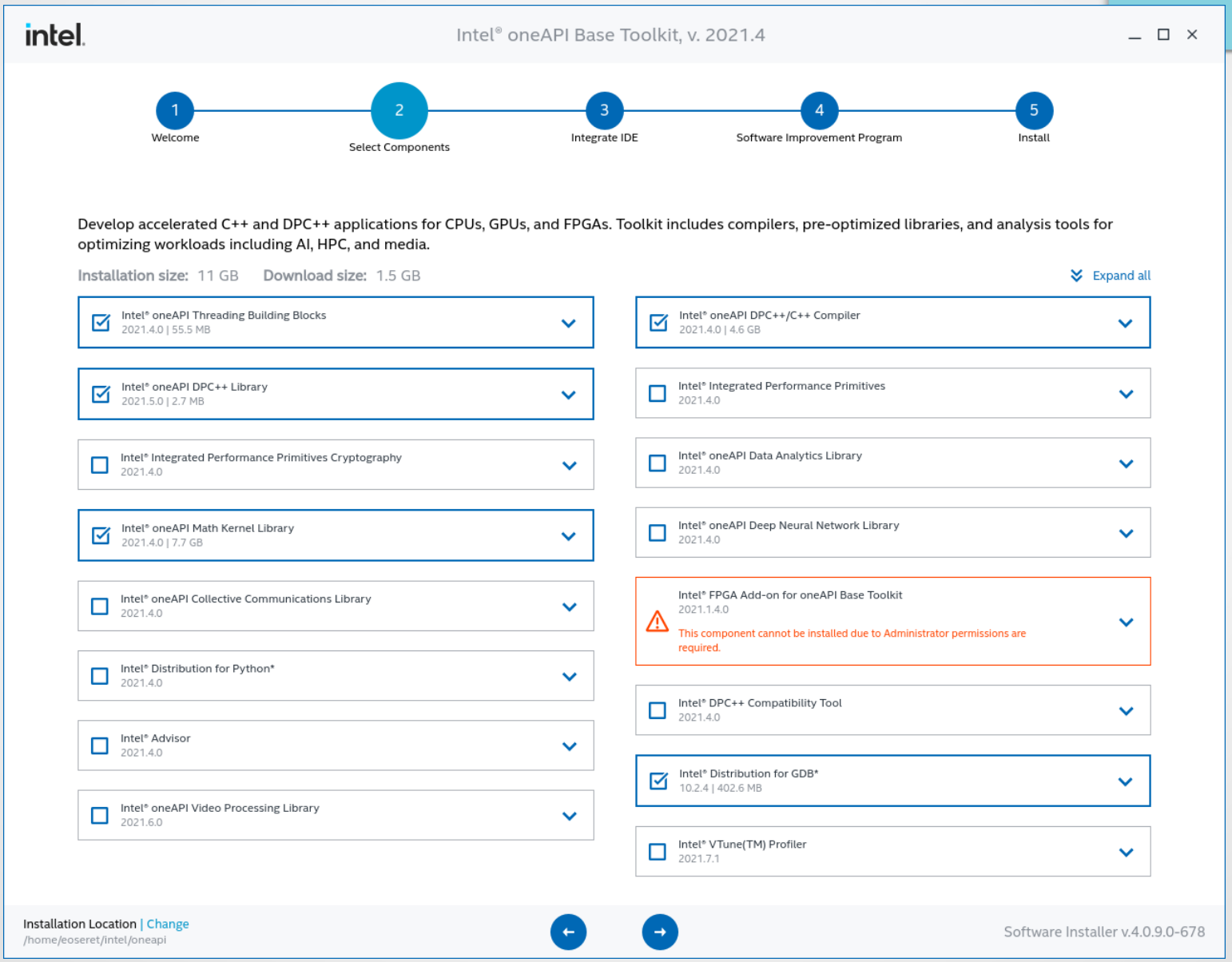
- Optimisé pour processeurs Intel
- Téléchargement (base toolkit) :
  - https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit-download.html
  - Utiliser la version Linux / online
  - Installation (ex si version 2021.4)

```
$ sudo bash l_BaseKit_p_2021.4.0.3422.sh
```

- Configuration
  - Choisir « custom installation » (rq : pour Fortran ou MPI, besoin kit HPC)
  - Tout décocher sauf DPC++/C++ compiler et ses dépendances (CF slide suivant). Je conseille de laisser MKL coché
- Utilisation

```
$ source /opt/intel/oneapi/setvars.sh
$ icx (C) / icpx (C++)
```

Compilateurs : Intel oneAPI



Compilateurs : AMD AOCC

- Optimisé pour processeurs AMD Zen
- Téléchargement :
  - https://developer.amd.com/amd-aocc/
  - Utiliser la version correspondant à votre Linux (en général sera .deb)
- Installation (ex si version 3.1)

```
$ sudo dpkg -i aocc-compiler-3.1.0_1_amd64.deb
```

- Utilisation

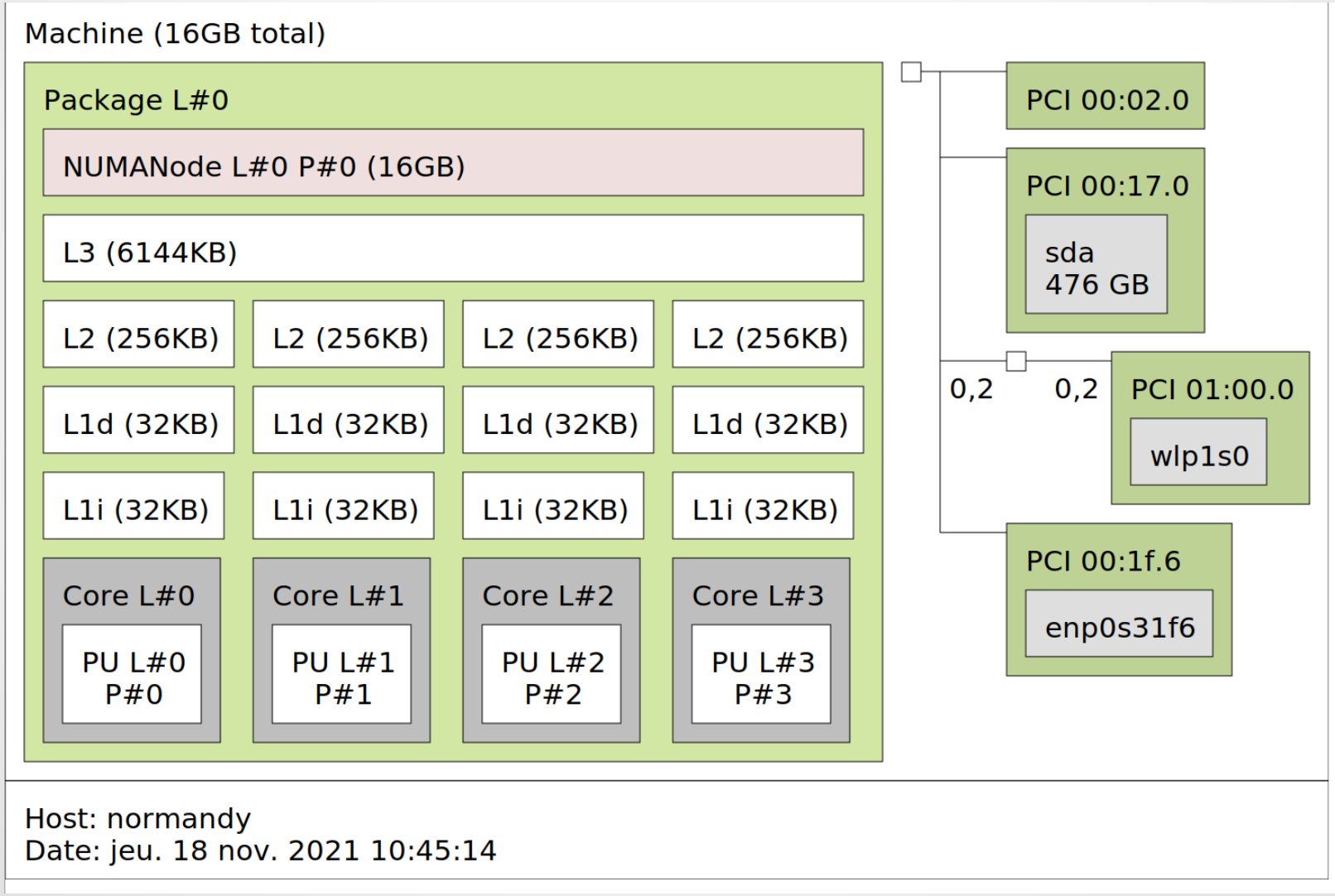
```
$ source /opt/AMD/aocc-compiler-3.1.0/setenv_AOCC.sh
$ clang (C) / clang++ (C++)
```

- Libm (math library) et BLAS : fourni séparément (AOCL)
  - https://developer.amd.com/amd-aocl/

Maîtrise environnement (Linux) lscpu

```
eoseret@normandy:~/MAQAO$ lscpu
Architecture :                          x86_64
Mode(s) opératoire(s) des processeurs : 32-bit, 64-bit
Boutisme :                              Little Endian
Address sizes:                          39 bits physical, 48 bits virtual
Processeur(s) :                         4
Liste de processeur(s) en ligne :       0-3
Thread(s) par cœur :                    1
Cœur(s) par socket :                    4
Socket(s) :                             1
Nœud(s) NUMA :                          1
Identifiant constructeur :              GenuineIntel
Famille de processeur :                 6
Modèle :                                158
Nom de modèle :                         Intel(R) Core(TM) i5-7440HQ CPU @ 2.80GHz
Révision :                              9
Frequency boost:                        enabled
Vitesse du processeur en MHz :          2800.083
Vitesse maximale du processeur en MHz : 2801,0000
Vitesse minimale du processeur en MHz : 800,0000
BogoMIPS :                              5599.85
Virtualisation :                        VT-x
Cache L1d :                             128 KiB
Cache L1i :                             128 KiB
Cache L2 :                              1 MiB
Cache L3 :                              6 MiB
Nœud NUMA 0 de processeur(s) :          0-3
Vulnerability Itlb multihit:            KVM: Mitigation: Split huge pages
Vulnerability L1tf:                     Mitigation; PTE Inversion; VMX conditional cache flushes, SMT dis
                                        abled
Vulnerability Mds:                      Mitigation; Clear CPU buffers; SMT disabled
Vulnerability Meltdown:                 Mitigation; PTI
Vulnerability Spec store bypass:        Mitigation; Speculative Store Bypass disabled via prctl and secco
                                        mp
Vulnerability Spectre v1:               Mitigation; usercopy/swapgs barriers and __user pointer sanitizat
                                        ion
Vulnerability Spectre v2:               Mitigation; Full generic retpoline, IBPB conditional, IBRS_FW, ST
                                        IBP disabled, RSB filling
Vulnerability Srbds:                    Mitigation; Microcode
Vulnerability Tsx async abort:          Mitigation; Clear CPU buffers; SMT disabled
Drapaux :                               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
                                         pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
                                         pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_goo
                                        d nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes6
                                        4 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pc
                                        id sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave
                                         avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpci
                                        d_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ep
                                        t vpid ept_ad fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms in
                                        vpcid rtm mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec
                                         xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_wi
                                        ndow hwp_epp md_clear flush_l1d
```

Maîtrise environnement (Linux) lstopo



Maîtrise environnement (Linux) cpupower

```
eoseret@normandy:~/MAQAO$ cpupower frequency-info
analyzing CPU 0:
  driver: acpi-cpufreq
  CPUs which run at the same hardware frequency: 0
  CPUs which need to have their frequency coordinated by software: 0
  maximum transition latency: 10.0 us
  hardware limits: 800 MHz - 2.80 GHz
  available frequency steps:  2.80 GHz, 2.80 GHz, 2.70 GHz, 2.40 GHz, 2.30 GHz, 2.20 GHz, 2.10 GHz, 1.90
GHz, 1.80 GHz, 1.70 GHz, 1.50 GHz, 1.40 GHz, 1.20 GHz, 1.10 GHz, 900 MHz, 800 MHz
  available cpufreq governors: conservative ondemand userspace powersave performance schedutil
  current policy: frequency should be within 800 MHz and 2.80 GHz.
                  The governor "powersave" may decide which speed to use
                  within this range.
  current CPU frequency: Unable to call hardware
  current CPU frequency: 900 MHz (asserted by call to kernel)
  boost state support:
    Supported: yes
    Active: yes
eoseret@normandy:~/MAQAO$ sudo cpupower frequency-set -g performance --min 2800MHz --max 2800GHz
Setting cpu: 0
Setting cpu: 1
Setting cpu: 2
Setting cpu: 3
eoseret@normandy:~/MAQAO$ cpupower frequency-info
analyzing CPU 0:
  driver: acpi-cpufreq
  CPUs which run at the same hardware frequency: 0
  CPUs which need to have their frequency coordinated by software: 0
  maximum transition latency: 10.0 us
  hardware limits: 800 MHz - 2.80 GHz
  available frequency steps:  2.80 GHz, 2.80 GHz, 2.70 GHz, 2.40 GHz, 2.30 GHz, 2.20 GHz, 2.10 GHz, 1.90
GHz, 1.80 GHz, 1.70 GHz, 1.50 GHz, 1.40 GHz, 1.20 GHz, 1.10 GHz, 900 MHz, 800 MHz
  available cpufreq governors: conservative ondemand userspace powersave performance schedutil
  current policy: frequency should be within 2.80 GHz and 2.80 GHz.
                  The governor "performance" may decide which speed to use
                  within this range.
  current CPU frequency: Unable to call hardware
  current CPU frequency: 2.80 GHz (asserted by call to kernel)
  boost state support:
    Supported: yes
    Active: yes
```

MAQAO: intro et utilisation

- Téléchargement & doc : http://www.maqao.org
- Installation/configuration (ex si version 2.14.1)

```
$ tar xf maqao.intel64.2.14.1.tgz
$ sudo cp maqao.intel64.2.14.1/maqao.intel64 /usr/local/bin/maqao
$ sudo sysctl -w kernel.perf_event_paranoid=1
$ sudo sh -c 'echo kernel.perf_event_paranoid=1 >>
/etc/sysctl.d/local.conf'
```

- Utilisation

```
Sans MAQAO :
$ ./myapp.exe 48 -i input.dat

Avec MAQAO (tout en ligne de commande) :
$ maqao oneview -R1 -- ./myapp.exe 48 -i input.dat

Avec MAQAO (recours à un fichier de config) :
$ maqao oneview -create-config
$ vim config.lua # compléter a minima executable et run_command
$ maqao oneview -R1 c=config.lua
```

OpenMP: intro et utilisation

- Permet d'exploiter plusieurs coeurs/threads (donc de « paralléliser »)
- Diverses constructions: régions (dont boucles) ou taches...
- C: `#pragma omp` (directives préprocesseur)
- Compilation et édition de lien avec `-fopenmp`

OpenMP: variables d' environnement pour l'affinité (« CPU binding »)

- `OMP_NUM_THREADS`: nombre de threads à utiliser dans les régions
- `OMP_PLACES=cores` **`OMP_PROC_BIND=spread`**
  - **Eloigner** le plus possible les threads
  - Données supposées partitionnées
  - On cherche à toucher le plus de caches L3 et canaux mémoire possible
  - Ex bi-sockets : T0 et T1 seront sur le premier coeur de chaque socket
- `OMP_PLACES=cores` **`OMP_PROC_BIND=close`**
  - **Regrouper** le plus possible les threads
  - Données supposées partagées
  - On cherche à réutiliser entre threads les données dans les caches L3
  - Ex bi-sockets : T0 et T1 seront sur les deux premiers coeurs du premier socket

OpenMP: overhead

- Coût fixe à amortir
- GOMP déconseillé
- Paralléliser à gros grain
  - boucle la plus externe
  - enchaînement/fusion plusieurs boucles dans même région
- Estimer travail par thread qui dépend de :
  - nombre d’itérations
  - coût de chaque itération
  - nombre de threads

OpenMP: parallélisation des boucles

- Programmes HPC généralement multithreadés à l'aide d'OpenMP (plutôt que TBB ou autres)
- Généralement seules les boucles sont parallélisées
-
```
#pragma omp parallel for
for (i=0; i<n; i++)
   a[i] = b[i] / c[i];
```

OpenMP: parallélisation des boucles

```
#pragma omp parallel
{
#pragma omp for
   for (i=0; i<n; i++)
      a[i] = b[i] / c[i];
...
#pragma omp for
   for (i=0; i<n; i++)
      c[i] = d[i] * e[i];
}
```

OpenMP: clause « reduction »

- ```
  float sum = 0.0f;
  #pragma omp parallel for reduction(+:sum)
  for (i=0; i<n; i++)
     sum += b[i] / c[i];
  ```
- En cas d’oubli, « race condition » et résultat faux !
  ```

OpenMP: clause schedule (omp for) 1/2

- `schedule(`**`static`**`)` : répartition **statique** (avant exécution) des itérations entre threads (si 4 threads, chacun prend 1/4 des itérations)
  - Coût ordonnancement nul
  - Idéal quand coût homogène entre itérations
- `schedule(`**`dynamic`**`)` : répartition **dynamique** des itérations. Chaque thread prend un paquet d'itérations (« chunk ») et en demande un autre dès qu'il l'a terminé
  - Coût ordonnancement d'autant plus élevé que la taille des chunks est petite (mais équilibrage de charge d'autant plus efficace)
  - Donc compromis à trouver entre coût ordo. et équilibrage
  - A utiliser quand les itérations ont des coûts fortement hétérogènes

OpenMP: clause schedule (omp for) 2/2

- `schedule(`**`guided`**`)` : idem `schedule(dynamic)` sauf que la taille des chunks est décroissante
  - Compromis entre `static` et `dynamic`
  - A tendance à donner des perfs aussi bonne que le meilleur choix de schedule, ou pas très éloignées
  - A utiliser comme choix « universel » (pas forcément le meilleur mais jamais complètement mauvais...)
- `schedule(`**`runtime`**`)` = `schedule(`**`OMP_SCHEDULE`**`)`

OpenMP: autres directives & clauses

- `omp parallel`: **`private, shared...`**
- `omp for`: **`private, nowait, collapse...`**
- Vectorisation
  - `omp` **`simd`** : vecto. d'une boucle
  - `omp` **`declare simd`** : vecto. d'une fonction
  - `omp` **`for simd`** : vecto. et parallélisation d'une boucle

OpenMP: liens

- Formations IDRIS
  - http://www.idris.fr/formations/openmp/
- OpenMP API Syntax Reference Card
  - https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5-2-web.pdf

MAQAO et OpenMP

- Utilisation avec un seul run

```
Sans MAQAO :
$ OMP_NUM_THREADS=4 ./myapp.exe 48 -i input.dat

Avec MAQAO (tout en ligne de commande) :
$ OMP_NUM_THREADS=4 maqao oneview -R1 -- ./myapp.exe 48 -i input.dat

Avec MAQAO (recours à un fichier de config) :
$ maqao oneview -create-config
$ vim config.lua # compléter a minima executable, envv_OMP_NUM_THREADS et
run_command
$ maqao oneview -R1 c=config.lua
```

- Exploration scalabilité

```
$ vim config.lua # modifier multiruns_params
multiruns_params = {
  { name= "1x2", envv_OMP_NUM_THREADS="2" },
  { name= "1x4", envv_OMP_NUM_THREADS="4" },
}
$ maqao oneview -R1 -WS c=config.lua
```

Vectorization

Emmanuel OSERET

Outline

- Introduction
- Impact on performance
- Getting best of auto-vectorization
- Compiler reports and tools
- Conclusion / take away
- Bibliography

Introduction

- Parallelism levels in HPC
- Why parallelize in processor ?
- Motivating example and definition
- Vector processing history
- x86 SIMD extensions

Parallelism levels in HPC clusters

- Nodes (« motherboards » packaged as blades)
- Sockets / processors (typically 2-4 sockets per node)
- Cores / (HW) threads (Thread Level Parallelism)
- (Pipelined) Execution units (Instruction Level Parallelism)
- **Vector units/registers (SIMD)**

Why parallelization (in processor) ?

- Power wall: cannot increase frequency
  - Base/nominal frequency (max ~3.5 GHz)
- Moores' law: more and more transistors
- More transistors is translated to
  - More cores
  - More execution units
  - **Wider vectors and execution units**
  - Larger data caches

Motivating example

```
for (i=0; i<n; i++)
  c[i] = a[i] + b[i]; // float
```

AVX-128: using only half of ymm registers

```
VMOVSS (b), %xmm0
VADDSS (a), %xmm0, %xmm1
VMOVSS %xmm1, (c)
```

a	1.2			
	+			
b	1.3			
	=			
c	2.5			

```
VMOVAPS (b), %xmm0
VADDPS  (a), %xmm0, %xmm1
VMOVAPS %xmm1, (c)
```

a	1.2	2.2	0.7	1.5
	+	+	+	+
b	1.3	0.6	0.1	1.2
	=	=	=	=
c	2.5	2.8	0.8	2.7

Definition

- Using “vector” / SIMD instead of scalar instructions
- **Same operation on independent values**
- In 95% cases, automatically applied by the compiler (on loops not too complex). But can be applied manually (assembly or intrinsics)
- **With scalar instruction cost, several (instead of a single) elements are processed.** Ex. for FP elements:
  - SSE2-4 (128 bits): 4 FP32 or 2 FP64
  - AVX1-2 (128/256 bits): 4/8 FP32 or 2/4 FP64

Vector processing History

- ILLIAC IV (delivered in 1972)
  - 150 MFLOPS, multiple ALUs
- CRAY-1 (1976)
  - 240 MFLOPS, superpipeline, 64 words vectors
- CRAY-2, X-MP, Y-MP (1985-88)
- Mainstream (x86/ARM) processors: **SIMD extensions**

X86 SIMD extensions (1/4)

- MMX
  - 8 64 bits registers (shared with x87)
  - Integer computation
  - Introduced with Pentium MMX (1996)
- SSE
  - 8/16 (*) 128 bits new registers xmm0-15
  - Single precision computation (FP32)
  - Introduced with Pentium III (1999)

(*) IA32: 8, x86-64: 16

X86 SIMD extensions (2/4)

- SSE2
  - Extension to double precision (FP64) and integer
  - Introduced with Pentium 4 (2001)
  - Remark: since all x86-64 processors support SSE2, **many compilers generate by default SSE2 code (even if target processor supports a more recent instruction set)**
- SSE3/SSSE3/SSE4.1/SSE4.2
  - 2004-2006
  - New instructions in general quite minor or for crypto

X86 SIMD extensions (3/4)

- AVX
  - Registers width doubled (128 → 256 bits) ymm0-15
  - Integer computations still limited to 128 bits
  - Introduced with Sandy Bridge [Core i3/5/7 2xxx] (2011)
- AVX2
  - Integer computations extended to 256 bits
  - FMA (Fused Multiplied Add: A×B+C)
  - GATHER (load x[ind[i-i+3]] to y[i-i+3])
  - Introduced with Haswell [Core i3/5/7 4xxx] (2013)

X86 SIMD extensions (4/4)

- AVX512
  - 32 512 bits registers zmm0-31
  - Conflict detection (histogram)
  - Masking (allows quite efficient vectorization of conditional expressions)
  - Introduced with KNL (Knights Landing, 2016) and Skylake-SP (2017)

Impact on performance

- 2 interpretations
  - Possible to increase performance by 2x/4x
  - **If not vectorized, you are loosing 50 or 75% of peak performance !**
- Quizz: what is Core i7 8700K (CoffeeLake) peak GFLOPS ?
  - AVX2, 2 FMA units
  - 6 cores
  - Assuming operating @ 3.7 GHz (base frequency) with no AVX penalty

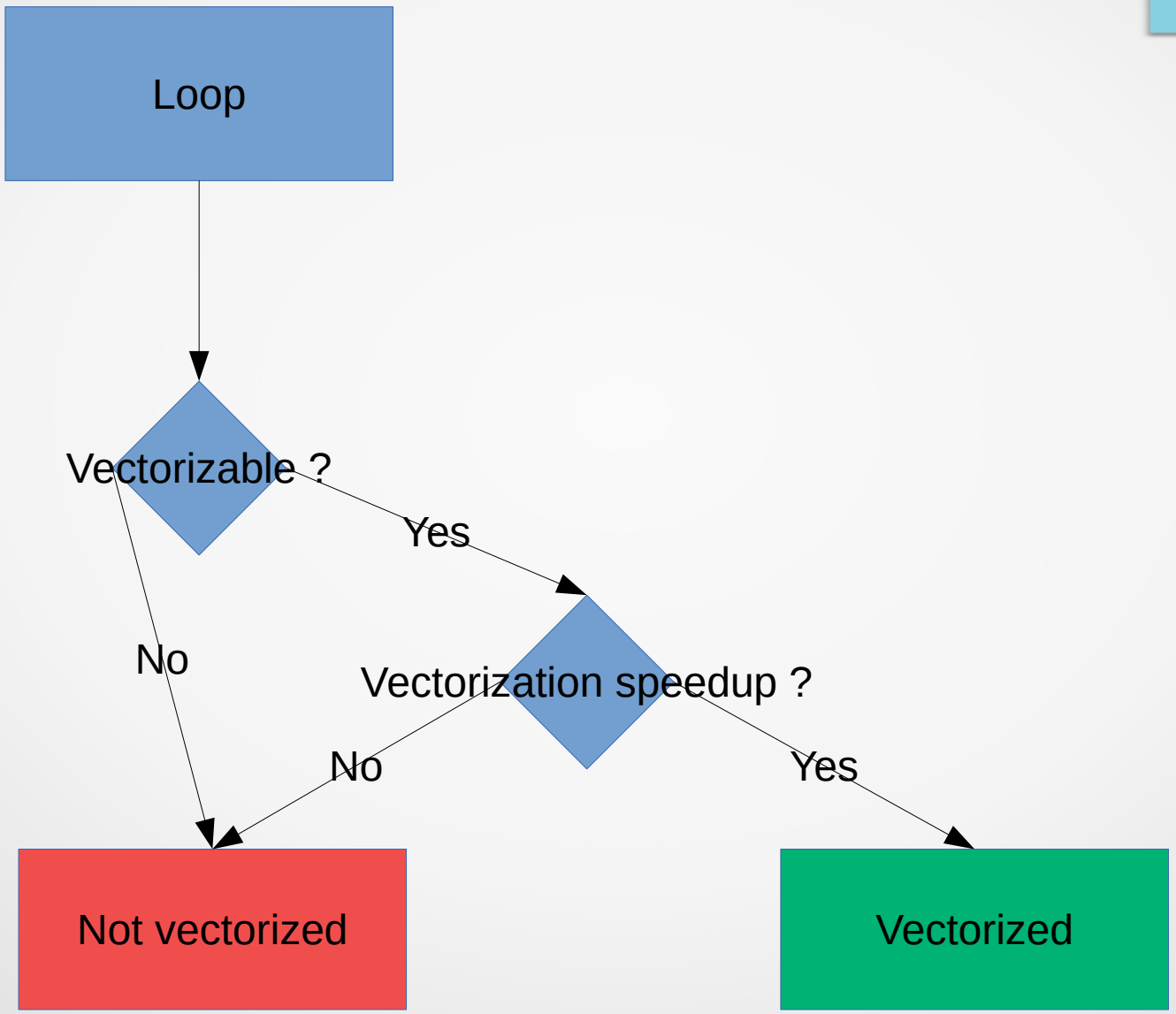
Impact on performance

- Core i7 8700K (CoffeeLake) peak GFLOPS
  - FP32: 6 (cores) x 2 (FMA units per core)
    x 2 (ADD+MUL) x 3.7E9 (cycles per second)
    x 8 (FP32 elements in a 256 bits vector)
    = 710 GFLOPS
  - FP64: 710/2 = 355 GFLOPS
- Issue: memory and even L3$ cannot feed SIMD units
  - Needs data in registers or private (L1/L2) caches

Getting best of auto-vectorization

- Understand vectorization steps
- Make your loop vectorizable
- Enable vectorizer
- Unlock vectorization speedup limiters
- Check vectorization performance impact
- Advanced

Understand vectorization steps



Make your loop vectorizable

- Elements can be processed independently
  - No loop carried dependencies
  - No aliasing in C
- Simple enough to the vectorizer
  - Avoid conditional structures, external function calls etc.

Array aliasing (C) (1/2)

- In C, arrays are passed as pointers => touched data can “alias” (overlap). No problem in Fortran: array datatype
- Possible cases about: `for(i) a[i]=b[i]+c[i];`
  - b = c, a[10] = b[0]
- Aliasing prevents vectorization in many cases or makes it more complex (compiler has to generate multiple versions and check pointer aliasing)

Array aliasing (C) (2/2)

- Workarounds
  - Fortran
  - C99 `restrict` keyword (or compiler specific equivalent)
  - Intel: `-fno-alias` (only for modules containing only non-aliasing code)
  - With C, two last workarounds generates wrong results (or undefined behavior) if input pointers alias

Enable vectorizer

- GNU: `-O3` (`-Ofast` if reductions)
  - Remark: default is -`O0` !
- Intel: `-O2`
  - Remark: default is -`O2` but -`O0` with `-g` !

Vectorization speedup limiters

- **Non stride 1 memory accesses**
- Unaligned memory accesses
- High latency memory accesses
- Non regular control flow (conditional structures)
- Non full-width execution units
- Vector width not fully used
  - e.g using XMM (128b) instead of YMM (256b) registers
- Low trip count

Non stride 1 memory accesses

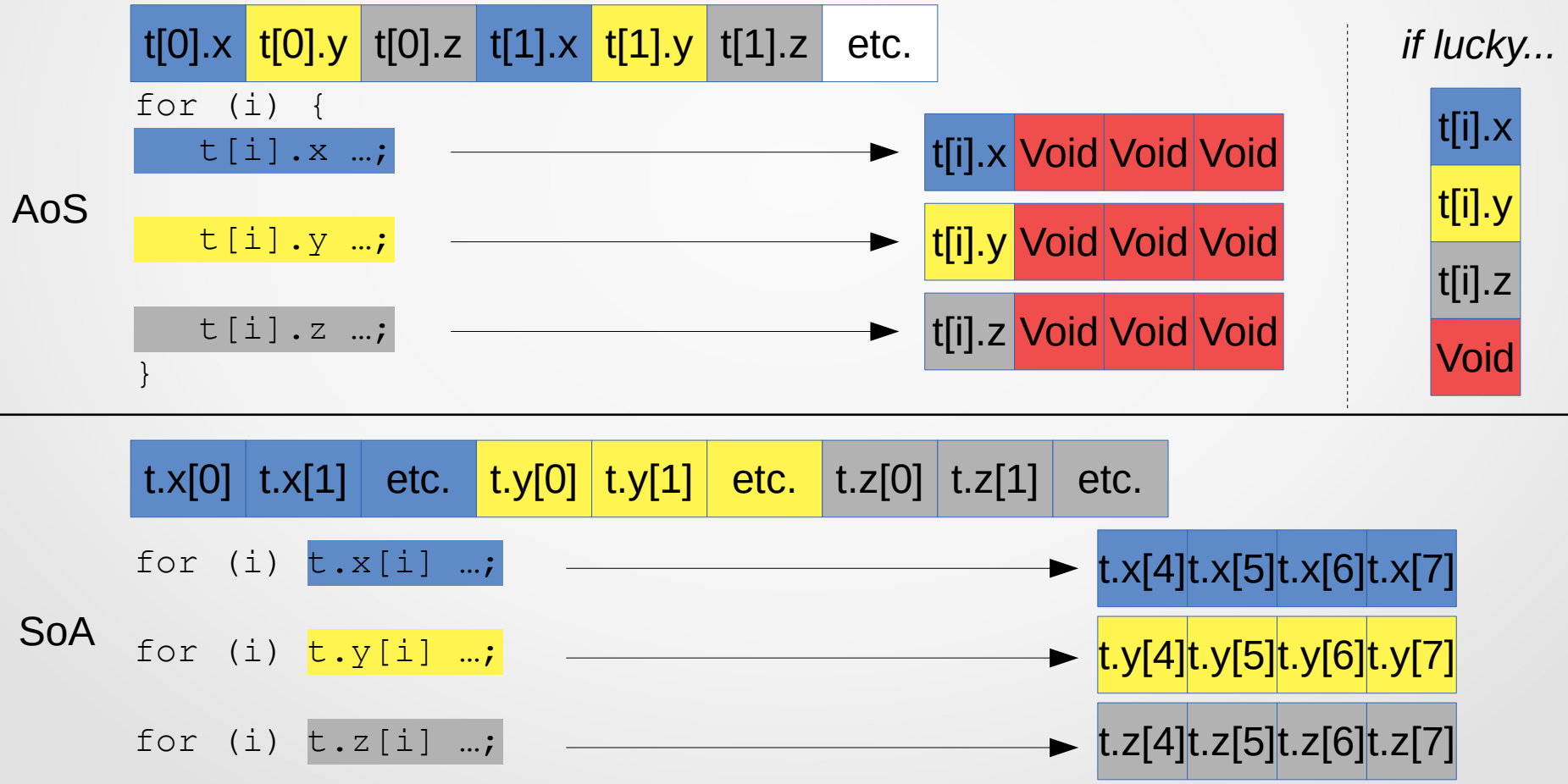
- Scalar instructions used to load/store sparse elements to/from a contiguous pack of elements
  - Some processors support GATHER/SCATTER instructions that replace N scalar load/stores into 1 vector instruction
  - Still slower than contiguous accesses !
  - `GATHER: dst[0:3] = src[ind[0:3]]`
  - `SCATTER: dst[ind[0:3]] = src[0:3]`

Make memory accesses stride 1 !

- AoS vs SoA
- Loop permutation

AoS vs SoA

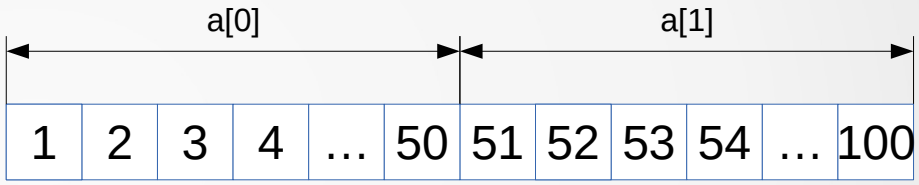
- AoS: Array of Structures, SoA: Structure of Arrays
- Use SoA if you can process each field in a different loop



Loop permutation (C)

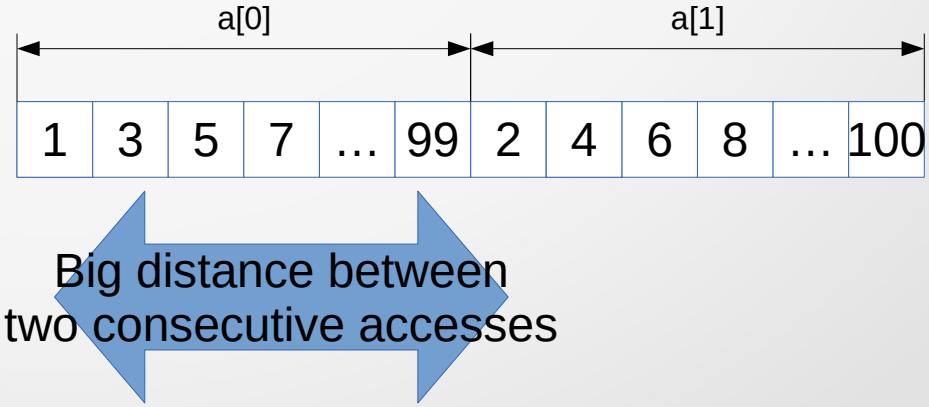
- Storage is row-major (row elements are contiguous)
- Stride 1:

```
for (i=0; i<2; i++)
  for (j=0; j<50; j++)
    a[i][j]= …
```



- Non stride 1:

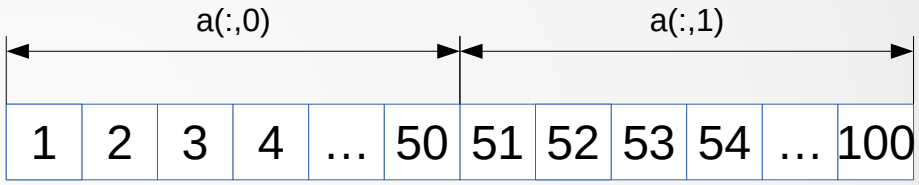
```
for (j=0; j<50; j++)
  for (i=0; i<2; i++)
    a[i][j]= …
```



Loop permutation (Fortran)

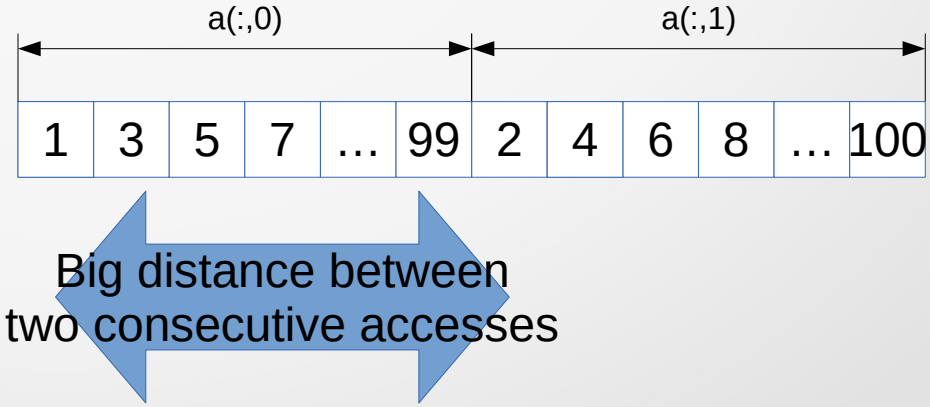
- Storage is column-major (col. elements are contiguous)
- Stride 1:

```
do i=1,2
  do j=1,50
    a(j,i)= …
```



- Non stride 1:

```
do j=1,50
  do i=1,2
    a(j,i)= …
```

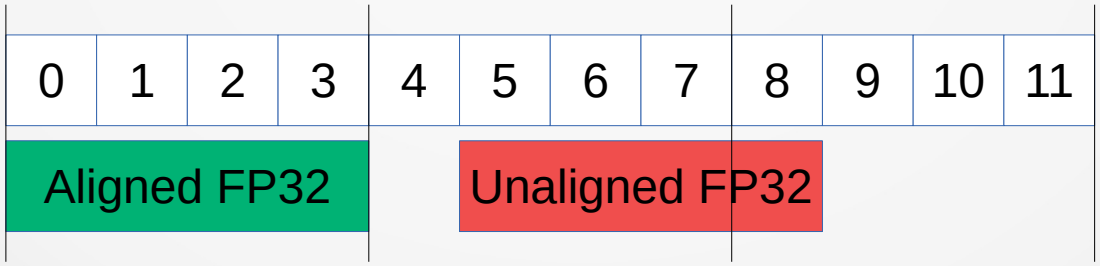


Vectorization speedup limiters

- Non stride 1 memory accesses
- **Unaligned memory accesses**
- High latency memory accesses
- Non regular control flow (conditional structures)
- Non full-width execution units
- Unused widest vector instructions
- Low trip count

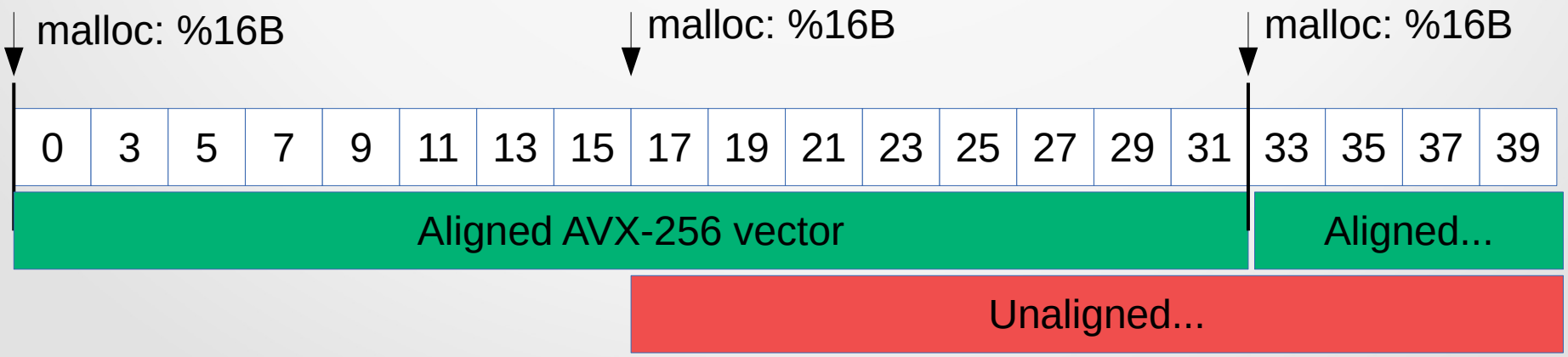
Data alignment (1/3)

- To be loaded/stored, any data has to be “aligned”
- Definition: address%N = 0 (N: alignment boundary)
- In general, N = sizeof (data)
  - FP32 elements has to be aligned on 4 bytes, etc.



Data alignment (2/3)

- Compiler ensures scalar data alignment
  - It is allocating automatic/static variables
  - Dynamically allocated segments are aligned at least on 8B, 16 in practice on Linux x86-64
- But not for SIMD data
  - 128, 256 bits or wider for the processor point of view



Data alignment (3/3)

- Unaligned loads/stores often available
  - Slower than aligned ones (especially in L1)
  - In recent processors, unaligned instructions on aligned data are as fast as with aligned instructions
  - Generated by default
- Need to:
  - Align your data/arrays (use aligned malloc (e.g `posix_memalign`) and pad multidimensional arrays
  - Inform your compiler of aligned data. Ex:
    - GNU: `__builtin_assume_aligned()`
    - Intel: `#pragma VECTOR ALIGNED`

Vectorization speedup limiters

- Non stride 1 memory accesses
- Unaligned memory accesses
- **High latency memory accesses**
- Non regular control flow (conditional structures)
- Non full-width execution units
- Unused widest vector instructions
- Low trip count

High latency memory accesses

- Loading/storing data from “slow” (high latency, low throughput) memory hierarchy levels (L3, RAM) prevents benefiting from vectorization speedup
  - Data access slower than vector processing
- Use regular data access patterns
  - To maximize hardware prefetchers efficiency: data in L3/RAM could be read in advance and made available in L1/L2 when effectively accessing them
- Blocks data to L1/L2 if reuse (temporal locality)

Vectorization speedup limiters

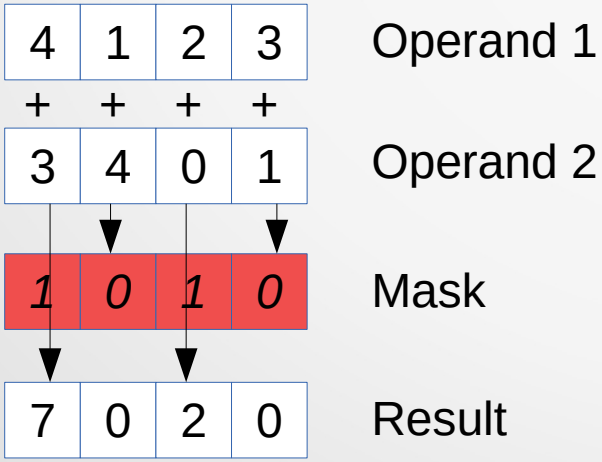
- Non stride 1 memory accesses
- Unaligned memory accesses
- High latency memory accesses
- **Non regular control flow (conditional structures)**
- Non full-width execution units
- Unused widest vector instructions
- Low trip count

Make your loop regular

- Hoist conditional structures out of the loop
- Use conditional writes
  - `if (t[x] < t[i]) t[x] = t[i];`
  - `=> t[x] = max (t[x], t[i]);`

Masks

- Possible to invalidate/ignore output of some elements
- These elements are defined via a mask (register)
- Useful to vectorize peel/tail loops
- Can remove branches
  - `if (i%2==0) a[i]=...`



Vectorization speedup limiters

- Non stride 1 memory accesses
- Unaligned memory accesses
- High latency memory accesses
- Non regular control flow (conditional structures)
- **Non full-width execution units**
- Unused widest vector instructions
- Low trip count

Non full-width units

- Native unit width < vector register width
- Most of Intel processors: native DIV/SQRT width = 128b
  - Avoid DIV/SQRT
  - E.g. replace `a[i]/b` with `a[i]*inv_b`
- On Sandy/Ivy Bridge: native load/store units width = 128b

Vectorization speedup limiters

- Non stride 1 memory accesses
- Unaligned memory accesses
- High latency memory accesses
- Non regular control flow (conditional structures)
- Non full-width execution units
- **Unused widest vector instructions**
- Low trip count

Specialize compilation for target

- GNU: `-march=native`, `-march=haswell`...
- Intel: `-xHost`, `-xCORE-AVX2`...
- These flags enable latest SIMD instruction sets supported by your proc. (otherwise, SSE2)
  - **But illegal exception if running on a processor without such instruction sets !**

Vectorization speedup limiters

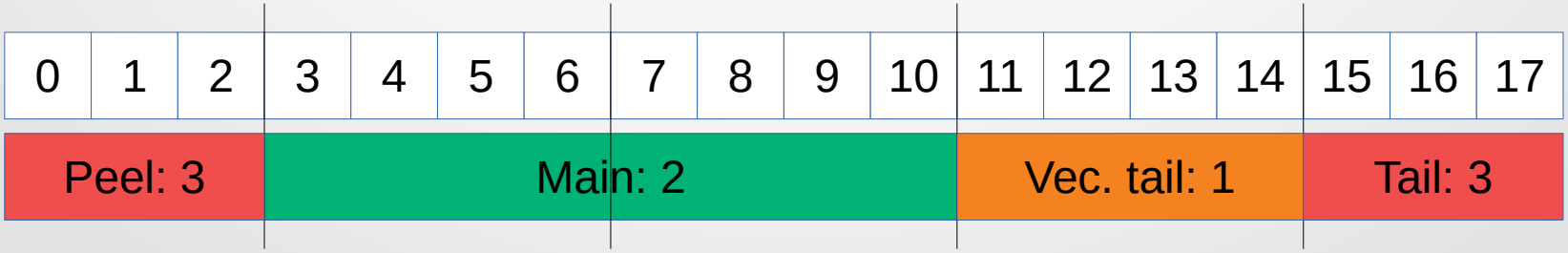
- Non stride 1 memory accesses
- Unaligned memory accesses
- High latency memory accesses
- Non regular control flow (conditional structures)
- Non full-width execution units
- Unused widest vector instructions
- **Low trip count**

Vectorized loop structure

- Peel loop (to sync to next alignment boundary)
  - Scalar or (block-vectorized + masking)
- Main loop version #1
  - Vectorized + unrolled, aligned for most arrays
- Main loop version #2
  - Vectorized + unrolled, aligned only for critical arrays
- *With no masking*: Vectorized remainder/tail loop
  - Vectorized but not unrolled
- Remainder/tail loop
  - Scalar or (block-vectorized + masking)

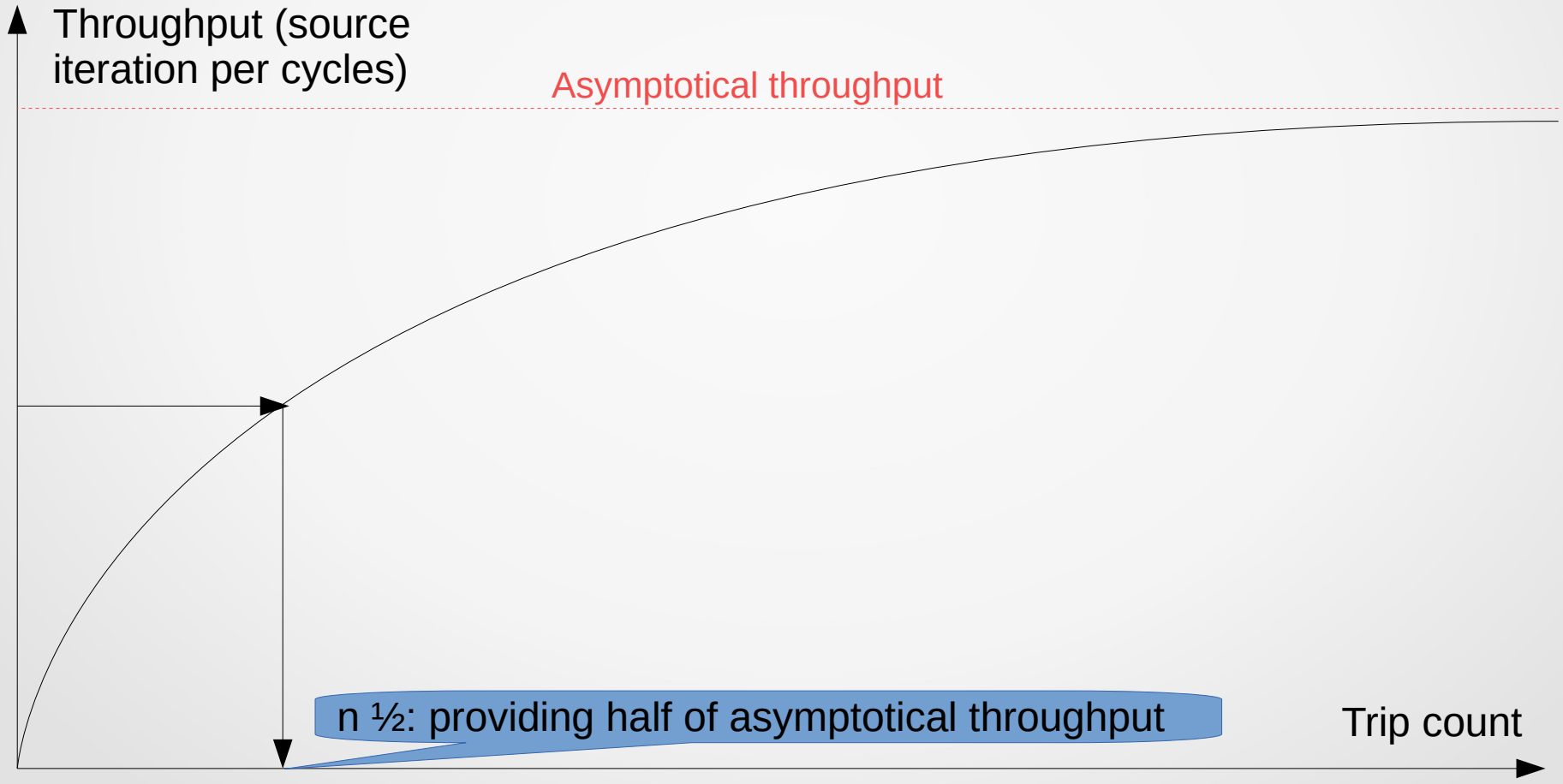
Vectorization overhead (1/3)

- Assuming 18 iterations, 4 elements vectors, 2x unroll
- Assuming 3 first elements before first alignment boundary
- Only 1 iteration in the fast loop !
- Unit cost (for 1 source loop iteration):
  - 1 in peel/tail loops (not vectorized)
    - => 3 for 3 elements
  - ¼ in main or vectorized tail (vectorized)
    - => 1 (resp. 2) for 4 (resp. 8) elements



Vectorization overhead (2/3)

- Remark: in reality, unrolling has impact (pipelining, loop control overhead)… => performance loss is greater !



Vectorization overhead (3/3)

- Tune vectorization and unrolling to trip count
- Inform that arrays are (or are not) aligned
  - `Intel + C: #pragma vector (un)aligned`
  - `Intel + Fort.: !DIR$ VECTOR (UN)ALIGNED`
  - `OpenMP: #pragma omp simd aligned(32)`
  - Compiler will probably remove peel loop
- Inform (Intel compiler) of your trip count
  - `C: #pragma loop count min(48) avg(55) max(95)`
  - `Fort.: !DIR$ LOOP COUNT MIN(48) AVG(55) MAX(95)`
  - Compiler will tune vector size (or disable vectorization) and unroll factor (or disable unrolling) according to its cost model

Check vectorization perf. impact

- By disabling vectorization + comparing performance with vectorized version
- GNU: `-fno-tree-vectorize -fno-openmp-simd`
- Intel: `-no-vec -no-simd -qno-openmp-simd`
- LLVM: `-fno-vectorize -fno-slp-vectorize`

Advanced: reductions (1/2)

- Vectorization does not remove inter-iteration dependencies
- But naturally parallelizes partial sums
- Partial sums = changed results (non associative FP math)
  - S = t[0] + t[1] + t[2]… (naïve, no partial sums)
  - S = (t[0]+t[4]+…) + (t[1]+t[5]+…) + (t[2]…) + (t[3]...)

0	1	2	3

\+ + + +

4	5	6	7

\+ + + +

8	9	10	11

\+ (etc.)

S0	S1	S2	S3	Horiz. ADD →	S0+S1+S2+S3	0	0	0

Advanced: reductions (2/2)

- Still 1 slot used in FPUs pipelines
- Loop unrolling needed (especially with in-order procs)
- 2-way ILP: pipeline + SIMD
- Remainer: `O3` not sufficient to make GNU compilers vectorize and unroll => `funroll-loops` + (`Ofast` or `fassociative-math`)
  - `Ofast` = `O3` + `ffast-math` + ...
  - `ffast-math` = `funsafe-math-optimizations` + ...
  - `funsafe-math-...` = `fassociative-math` + ...

Advanced: function calls

- Calling functions generally breaks vectorization
- Exceptions
  - Local functions with SIMD properties
    - C: `static`
  - Annotated external functions with SIMD properties
    - Generic: `#pragma omp declare simd`
    - Intel C: `__declspec(vector)`
    - Intel Fort.: `!DIR$ ATTRIBUTES VECTOR :: foo`
  - Math functions with vectorized versions provided by compiler or system (libm, Intel SVML...)

Advanced: forcing compiler (1/2)

- Ignore assumed vector dependencies
  - GNU + C: `#pragma GCC ivdep`
  - Intel + C: `#pragma ivdep`
  - Intel + Fort.: `!DIR$ IVDEP`
  - If you are wrong, **generated code will be wrong !**
- Disable vectorization (Intel)
  - C: `#pragma novector`
  - Fort: `!DIR$ NOVECTOR`
  - All loops: `-no-vec`

Advanced: forcing compiler (2/2)

- Force vec. (even if estimated as non profitable) [Intel only]
  - Intel + C: `#pragma vector always`
  - Intel + Fort: `!DIR$ VECTOR ALWAYS`
  - Intel (all loops): `-vec-threshold=0`
  - GNU (all loops): `-fvec-cost-model=unlimited`
- `IVDEP + VECTOR ALWAYS`
  - **OpenMP 4: `#pragma OMP SIMD (recommended)`**
  - Intel + C: `#pragma SIMD`
  - Intel + Fort: `!DIR$ SIMD`

Vectorizer reports

- GNU: `-fopt-info` (`-fopt-info-vec-all`)
- Intel: `-qopt-report` (`-qopt-report-phase=vec`)
- LLVM: `-Rpass=loop-vectorize -Rpass=loop-unroll`

Intel vectorizer reports

```
LOOP BEGIN at Driver.c(145,2)
   remark #25460: No loop optimizations reported
   LOOP BEGIN at Driver.c(148,3)
      remark #25460: No loop optimizations reported
      LOOP BEGIN at Driver.c(150,4)
         remark #15300: LOOP WAS VECTORIZED
      LOOP END
      LOOP BEGIN at Driver.c(150,4)
      <Remainder>
      LOOP END
   LOOP END
LOOP END
```

```
LOOP BEGIN at Multiply.c(45,2)
   remark #15541: outer loop was not auto-vectorized: consider using SIMD directive
   LOOP BEGIN at Multiply.c(55,3)
      remark #15344: loop was not vectorized: vector dependence prevents
vectorization. First dependence is shown below. Use level 5 report for details
      remark #15346: vector dependence: assumed FLOW dependence between  line 56 and
 line 56
   LOOP END
   LOOP BEGIN at Multiply.c(55,3)
   <Remainder>
   LOOP END
LOOP END
```

From: https://software.intel.com/en-us/node/590464

Vectorization-aware tools (MAQAO)

Code quality analysis

Source loop ending at line 10 in ...NDSON_test/CQA/matmul/kernel.c

It is composed of the loop 2

MAQAO binary loop id: 2

The loop is defined in /home/hpc/a2c06/lu23voj/MAQAO_HANDSON_test/CQA/matmul/kernel.c:9-10
In the binary file, the address of the loop is: 4009f0

8% of peak computational performance is used (0.67 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))

Gain | Potential gain | Hints | Experts only

Vectorization status

Your loop is not vectorized (all SSE/AVX instructions are used in scalar mode).
Only 25% of vector length is used.

Vectorization

Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED and could benefit from full vectorization.
By fully vectorizing your loop, you can lower the cost of an iteration from 3.00 to 0.75 cycles (4.00x speedup).

Since your execution units are vector units, only a fully vectorized loop can use their full power.

Proposed solution(s):

Two propositions:
- Try another compiler or update/tune your current one:
- Remove inter-iterations dependences from your loop and make it unit-stride.
* If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly:
C storage order is row-major: for(i) for(j) a[j][i] = b[j][i]; (slow, non stride 1) => for(i) for(j) a[i][j] = b[i][j]; (fast, stride 1)
* If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA):
for(i) a[i].x = b[i].x; (slow, non stride 1) => for(i) a.x[i] = b.x[i]; (fast, stride 1)

Bottlenecks

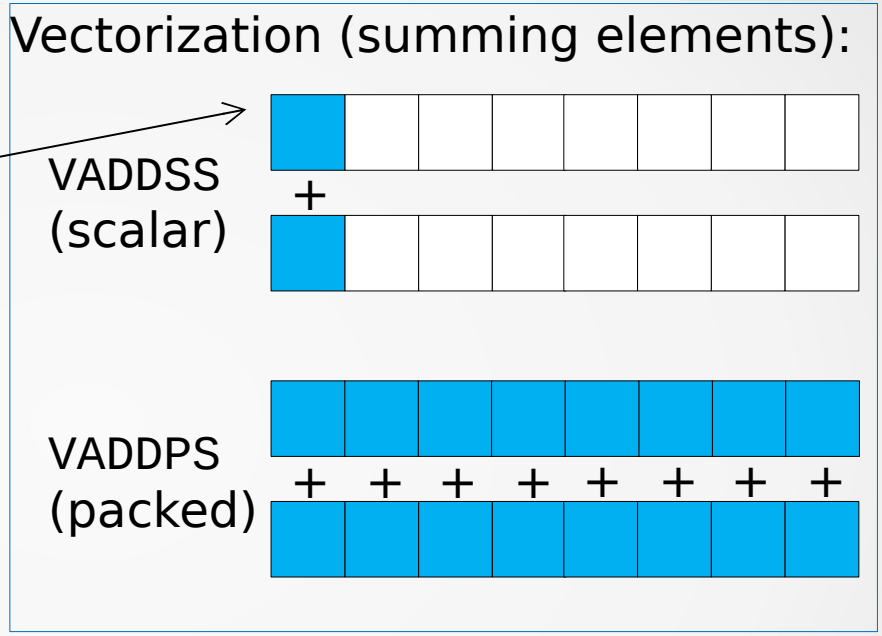
Detected a non usual bottleneck.

Proposed solution(s):

- Pass to your compiler a micro-architecture specialization option:
* use march=native.

Vectorization-aware tools (MAQAO)

```
Vectorization
-------------
(…) By fully vectorizing your loop,
you can lower the cost of an iteration
from 3.00 to 0.38 cycles (8.00x
speedup).(…)
 - Remove inter-iterations dependences
from your loop and make it unit-
stride.
  * If your arrays have 2 or more
dimensions, check whether elements are
accessed contiguously and, otherwise,
try to permute loops accordingly:
C storage order is row-major: for(i)
a[j][i] = b[j][i]; (slow, non stide 1)
=> for(i) for(j) a[i][j] = b[i][j];
(fast, stride 1)
  * If your loop streams arrays of
structures (AoS), try to use (…) SoA
```



- Accesses are not contiguous => let's permute k and j loops
- No structures here…

Vectorization-aware tools (Advisor)

Where should I add vectorization and/or threading parallelism? — Intel Advisor XE 2016

Summary | Survey Report | Refinement Reports | Annotation Report | Suitability Report

Program time: 26.54s | Vectorized | Not Vectorized | FILTER: All Modules | All Sources

Filters

Function Call Sites and Loops	Self Time	Total Time			Loop Type	Why No Vectorization?	Vector... Loops	Instruction Set Analysis	Optimizatio
[loop at loopstl.cpp:3962 in s ...	0.125s	0.125s			Expand	Expand	AVX	Inserts; Maske...	Unrolled
[loop at loopstl.cpp:6186 in ...	0.125s	0.125s		1	Collapse	Collapse	AVX		Unrolled
[loop at loopstl.cpp:6186 in s ...	0.062s	0.062s			Remainder				
[loop at loopstl.cpp:					Vectorized (Body)		AVX		Unrolled
[loop at loopstl.cpp:5625 in					Scalar	loop with early exits cannot be ...			

Loop may have several parts or versions

Scalar Loop. Not v... cannot be vectorized unless it meets search loop id... iteria
No loop transformations were applied

Loop summary

Vectorization and compiler optimization details

Top Down | Source | Loop Assembly | Assistance | Recommendations | Compiler Diagnostic Details

1 Issue: Ineffective peeled/remainder loop(s) present

All or some source loop iterations are not executing in the loop body. Improve performance by moving source loop iterations from peeled/remainder loops to the loop body.

Recommendations for optimization

Add data padding

Potential performance gain: Information not av...

Confidence this recommendation applies to your code: Information not available until Beta Update release

The trip count is not a multiple of vector length. To fix: Do one of the following:

- Increase size of objects and add iterations so the trip count is a multiple of vector length.
- Increase the size of static and automatic objects, and use a compiler option to add data padding.

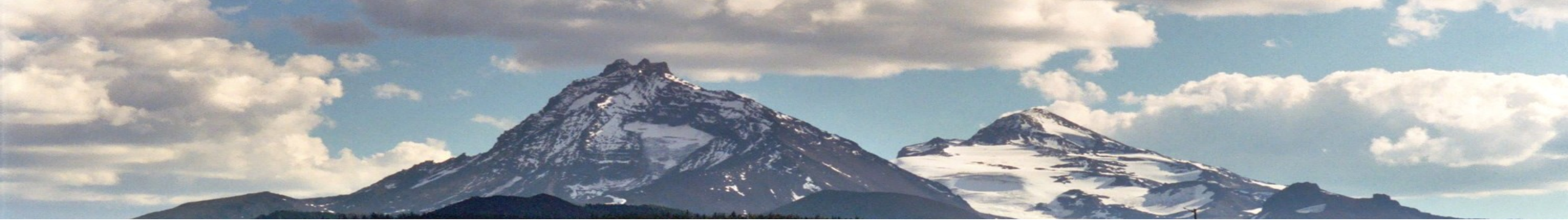
From: https://software.intel.com/en-us/articles/vectorization-advisor-faq

Conclusion

- Make your loops simple (regular…) and **stride 1**
- Check for sufficient loop **trip count**
- Block data to **L1 or L2**
- **Tune compiler** flags (`O2/O3/Ofast`, `march/xHost`)
- Check what your compiler is doing (reports, tools)

Bibliography

- https://software.intel.com (search by keywords)
  - Example: https://software.intel.com/en-us/node/590464 (“Generating a Vectorization Report”)
- https://en.wikipedia.org/wiki/SSE2 (AVX etc.)
- https://www.openmp.org/spec-html/5.1/openmpsu49.html
- https://gcc.gnu.org/onlinedocs/
- https://llvm.org/docs/Vectorizers.html
- http://maqao.org/



[A]rchitecture et [O]ptimisation de [C]ode pour microprocesseur hautes performances

Processor Architecture

Presented by

Emmanuel Oseret, Salah Ibnamar
Exascale Computing Research
University of Versailles St Quentin en Yvelines

William Jalby
University of Versailles St Quentin en Yvelines

Original version by *Allen D. Malony*, department of Computer and Information Science (University of Oregon), modified by *Emmanuel Oseret*.

Course Plan

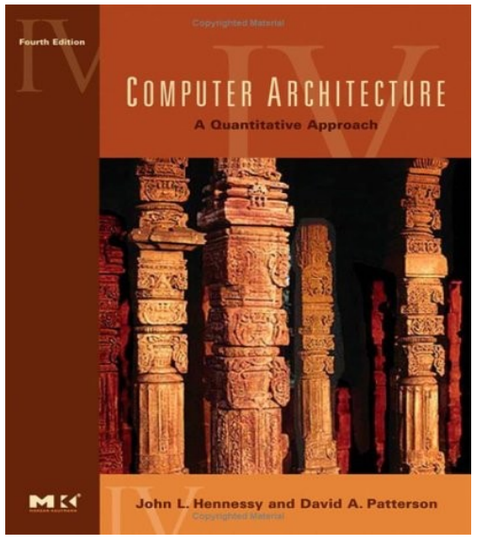
- Evolution of computer architectures
  - Scalar
  - Superscalar
  - Vector
- Hierarchical memory systems
  - Caches
  - Translation Lookaside Buffer (TLB)
- Code optimizations for CPU (micro) architectures
  - Scheduling
  - Loop transformations
- Multicore architectures

Origin of Course Material

- J. Fisher, HP Labs (Palo Alto, US)
- D. Patterson, UC Berkeley (US)
- D. Culler, UC Berkeley (US)
- S. Niar, Université de Valenciennes (FR)
- J.N. Amaral, University of Alberta (CA)
- R. Gupta, University of Arizona (US)
- S. Mahlke, University of Michigan (US)
- D. Etiemble, Université de Paris-Sud (FR)
- O. Sentieys, IRISA (FR)
- S. Touati, Sophia Antipolis (FR)

Origin of Course Materials

- J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, 5th Edition, ISBN: 9780123838728, Morgan Kaufmann, 2011.



- Wikipedia

WIKIPEDIA
The Free Encyclopedia

Introduction to CPU Architectures

- Scalar processors
  - Basic architecture
  - Pipeline (within an instruction)
  - Hazards and solutions
    - Data (out of order execution)
    - Control (branch prediction)
- Executing multiple instructions simultaneously
  - Pipelining between instructions
  - Overlapped execution
  - Multiple functional units
  - Out of order execution
  - Multi-issue execution
  - Superscalar and superpipelining
  - Very Long Instruction Word (VLIW)
  - Hardware multithreading (hyperthreading)
- SIMD and vector processing

Quantitative Principles

- Computers operate at a particular clock rate in discrete time (clock) events: *ticks*, *periods*, *cycles*
- Calculate the total time (T) to execute a program

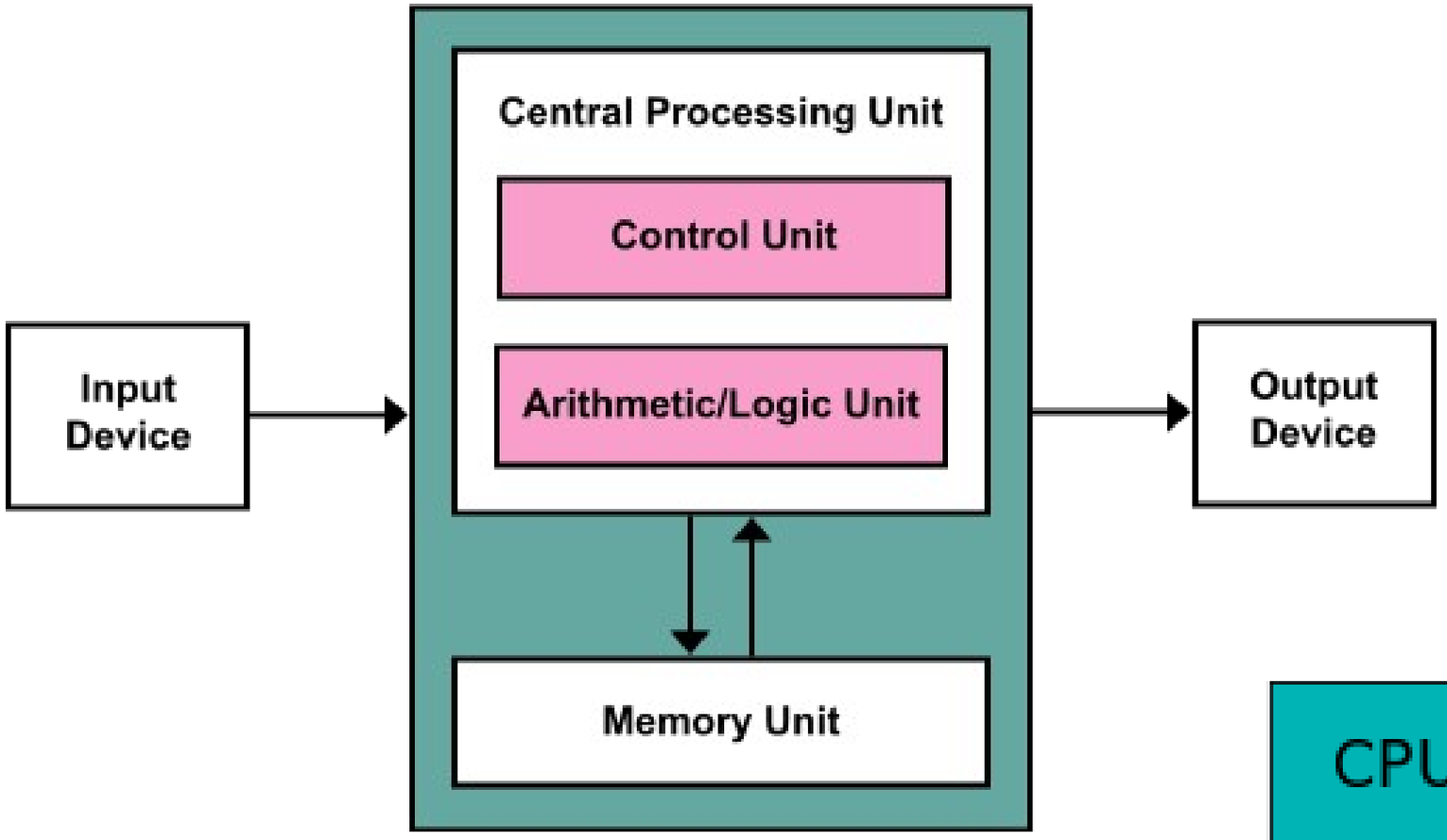
  T = (total clock cycles for program) / (clock rate)
- Let *CPI* be the # of cycles to execute an instruction
- Let *IC* be the total # instructions executed
- CPU performance equation (total execution time)

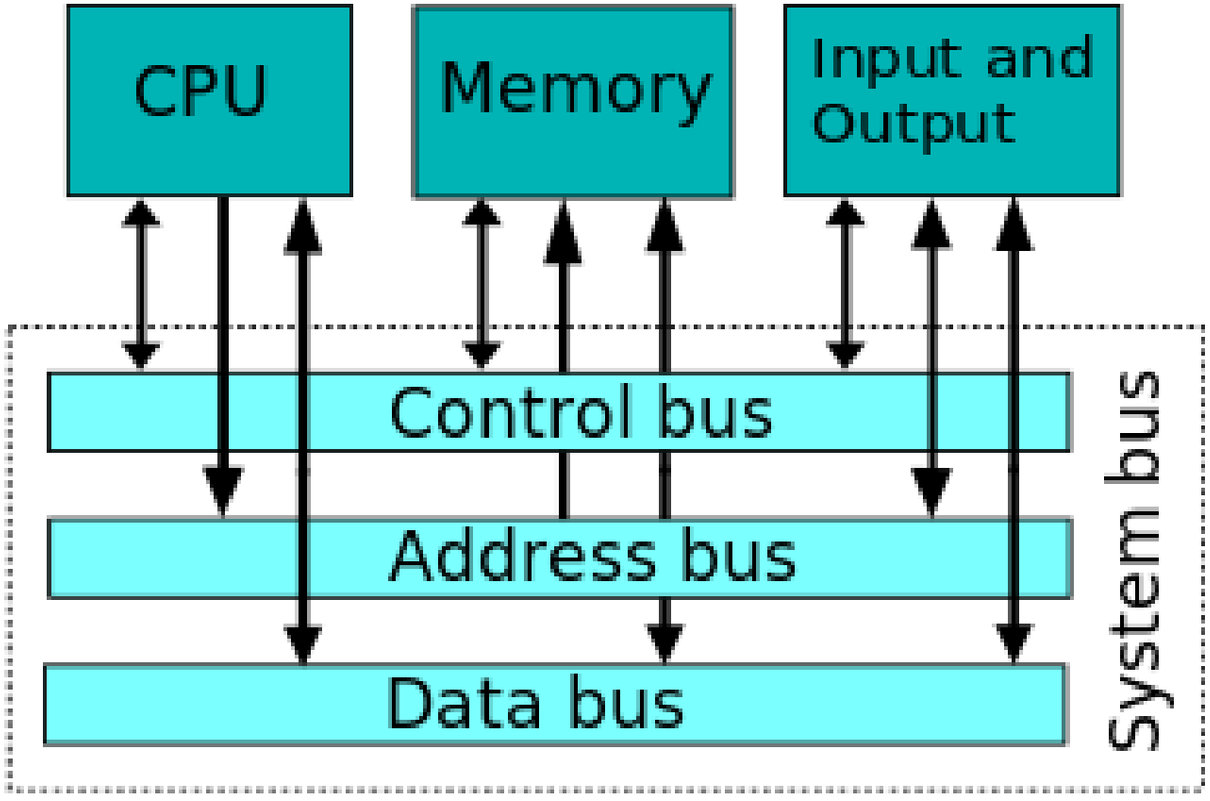
  *T = (IC * CPI) / (clock rate)*

"Iron Law" of CPU Performance

- CPU performance is dependent on 3 parameters:
  - *Clock cycle* (rate): hardware, organization
  - *Instruction count*: ISA, compiler
  - *Clock cycles per instruction*: ISA, organization
- These are not mutually exclusive
- Different instructions take different # cycles
- CPI is harder to determine because CPU architecture optimizations try to optimize instruction execution
- Instructions per cycle (*IPC = 1 / CPI*)
  - Objective is to increase the IPC

Von Neumann Computer Architecture





Source : Wikimedia

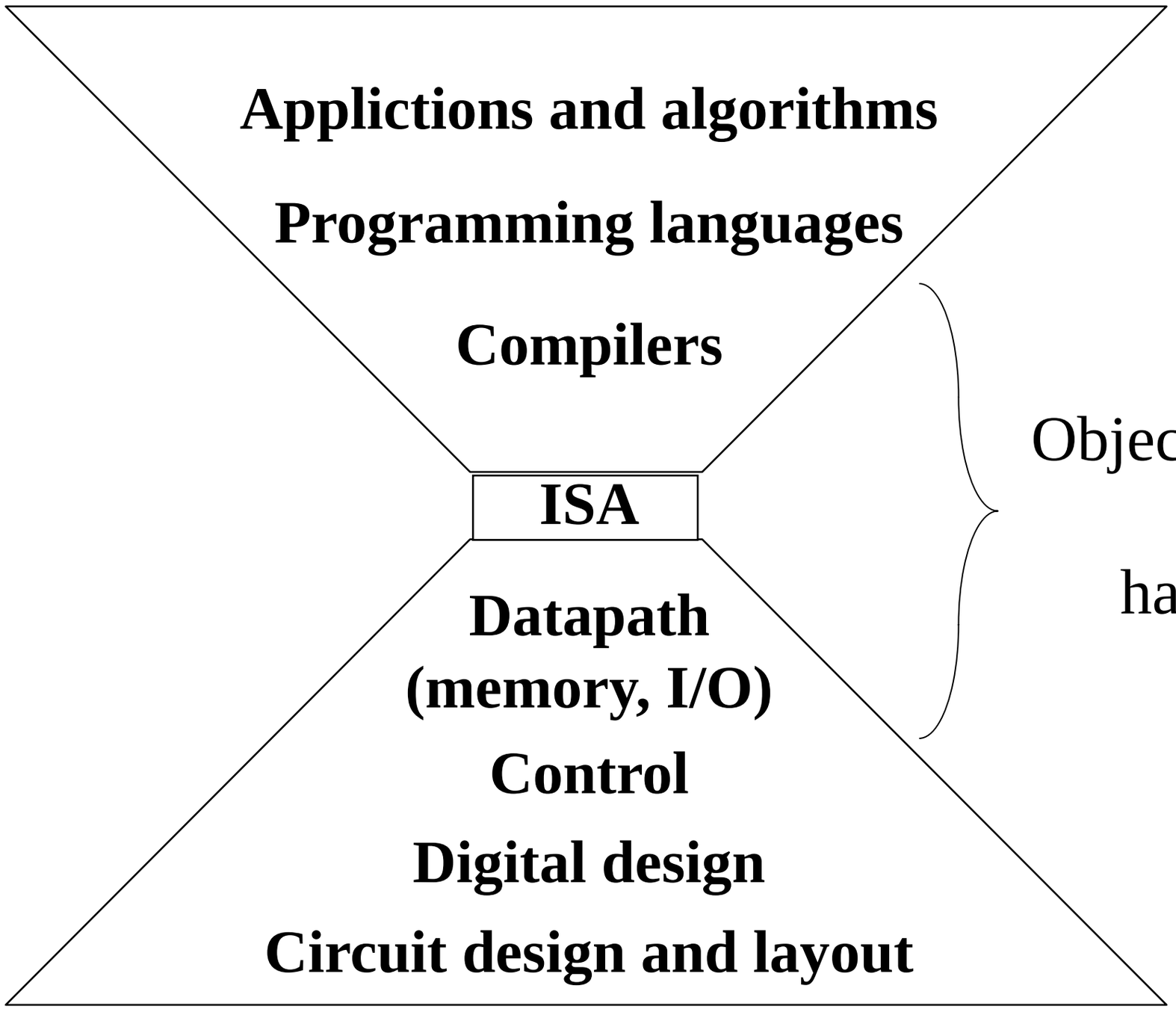
Computer Architecture

- Architecture
  - What is visible to the programmer
- Components
  - Central processing unit (CPU)
  - Instruction set architecture
    - logical and arithmetic operations
  - Memory model
    - CPU data registers
    - memory
  - Memory addressing modes
    - register-register
    - memory-register
    - memory-memory

Architecture versus Micro-architecture

- Processor architecture is concerned with what is visible to the programmer
- Micro-architecture defines:
  - Internal processor structures and mechanisms of a processor necessary to implement the ISA
  - What happens when an instruction is executed

Architecture Levels



Objective of the course is to look at the hardware/software interface

Source : David Patterson

Performance Improvement

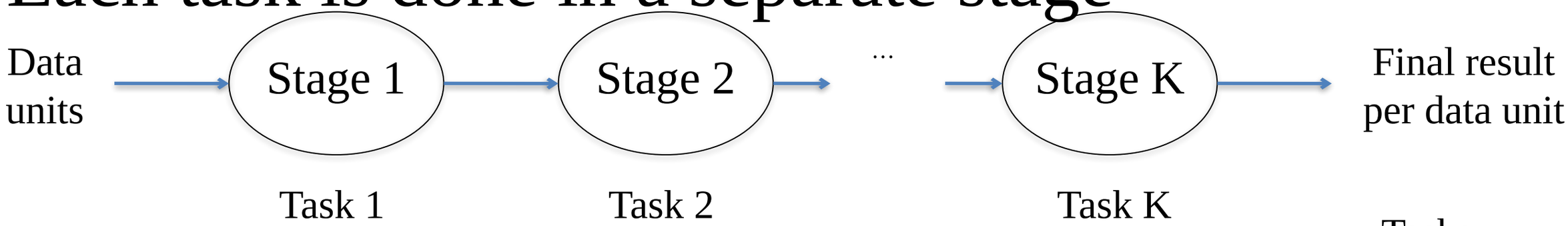
- For a particular architecture, how do we improve performance?
  - First idea might be to just increase the clock frequency
  - Improve the efficiency of components
    - reduces the time needed to perform the operations
  - Reduce the cost of certain operations in the processor
  - Must be concerned with the physical limits
    - IC technology used (circuit line width)
    - power consumption and heat dissipation
-

Pipeline

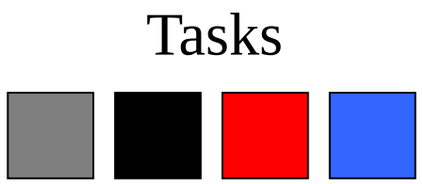
- A pipeline is a linear sequence of stages
- Data flows through the pipeline
  - From stage 1 to the last stage
  - Each stage performs some task
    - uses the result from the previous stage
  - Data is thought of as being composed of units (items)
  - Each data unit can be processed separately in pipeline
  - Sequence of stages (tasks) matters (functional operation)
  - Data sequence might matter
- Pipeline computation is a special form of *producer-consumer* parallelism
  - Producer tasks output data …
  - … used as input by consumer tasks

Pipeline Model

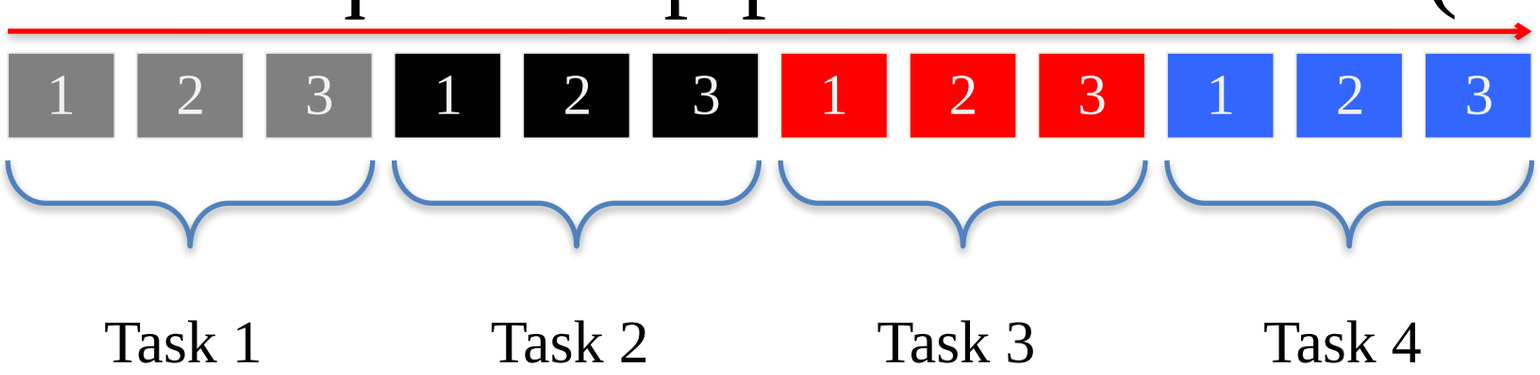
- *Stream* of data operated on by succession of tasks
  - Assumption: data input and output must be in sequence
- Each task is done in a separate stage

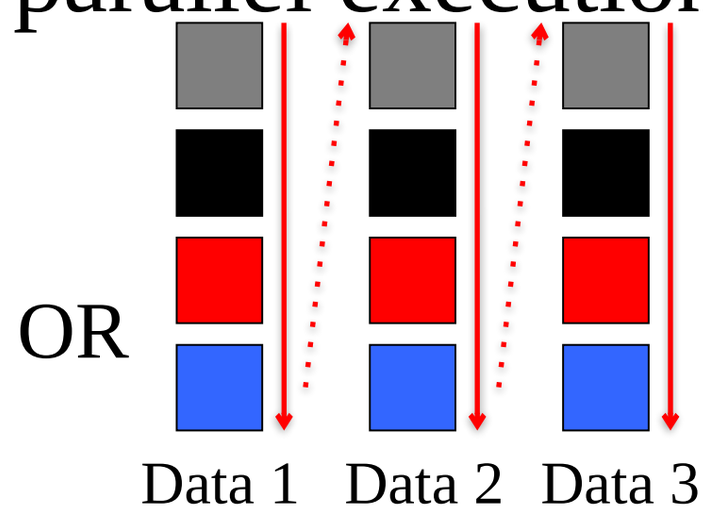


- Consider 3 data units and 4 tasks (stages)



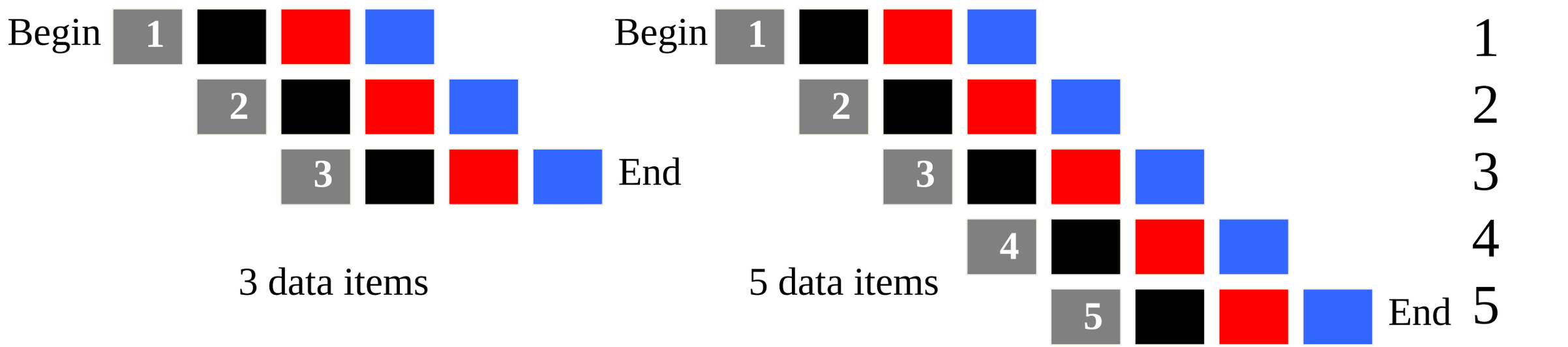
  - Sequential pipeline execution (no parallel execution)





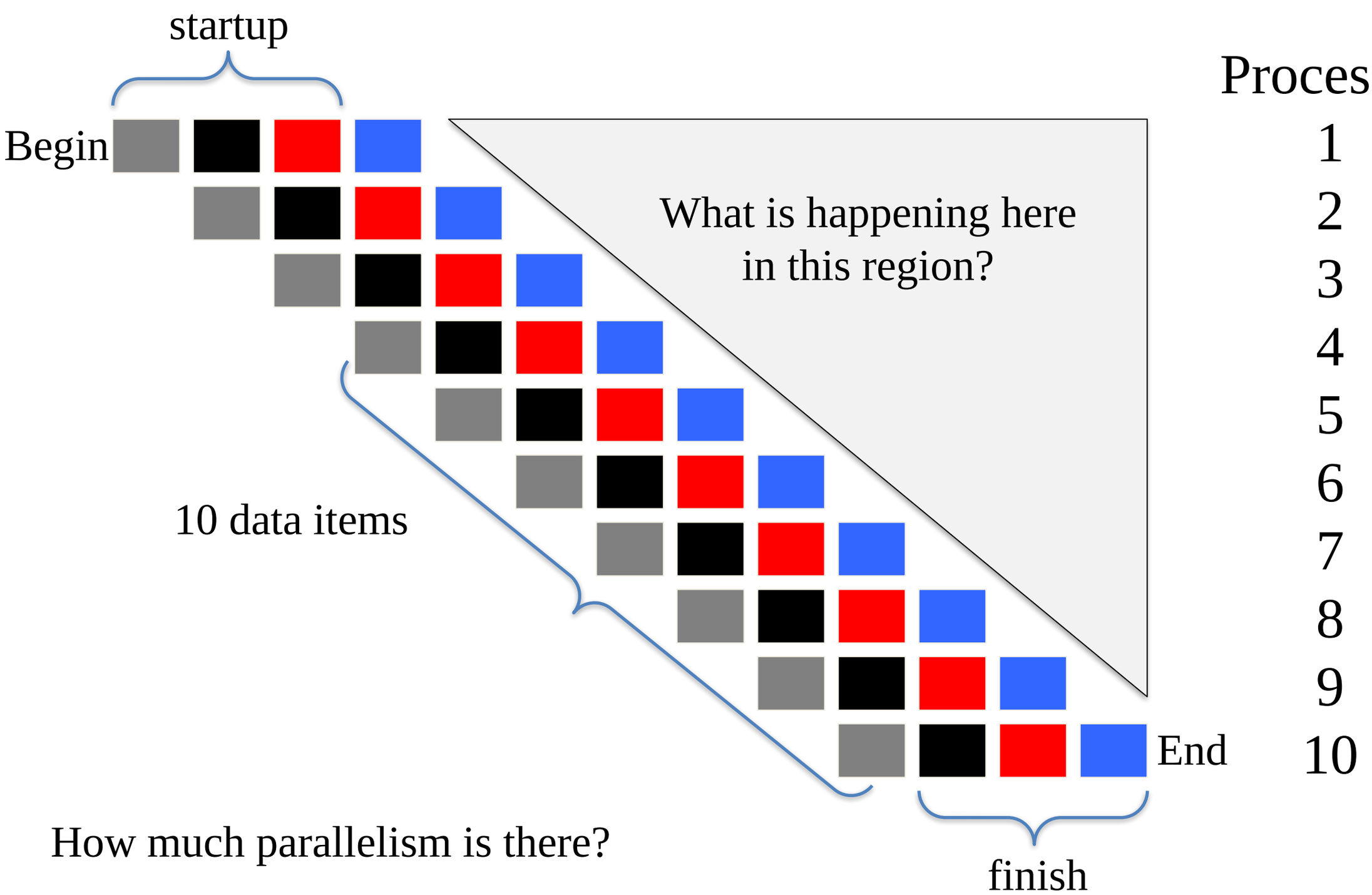
Where is the Concurrency? (Serial Pipeline)

- Pipeline with serial stages
  - Each stage runs serially (i.e., can not be done in parallel)
  - Assume that we can not parallelize the <u>tasks</u> (for now)
- What can we run in parallel?
  - Think about data parallelism
  - Provide a separate pipeline for each data item



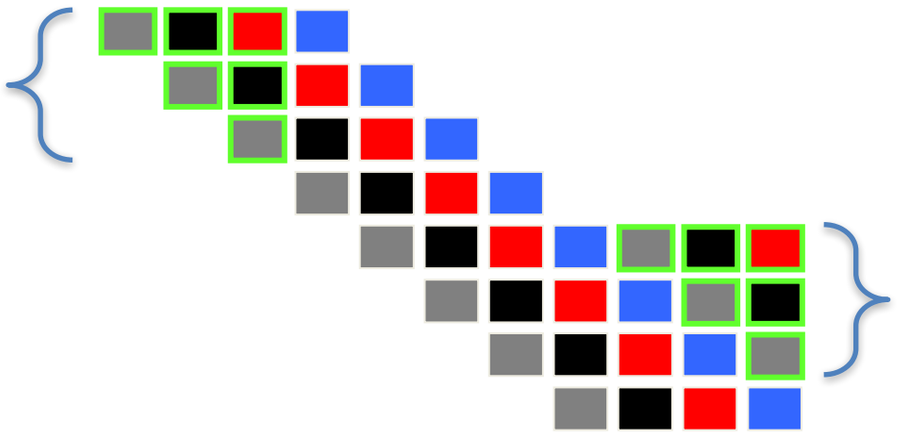
- What do you notice as we increase # data items?

Where is the Concurrency? (Serial Pipeline)



How much parallelism is there?

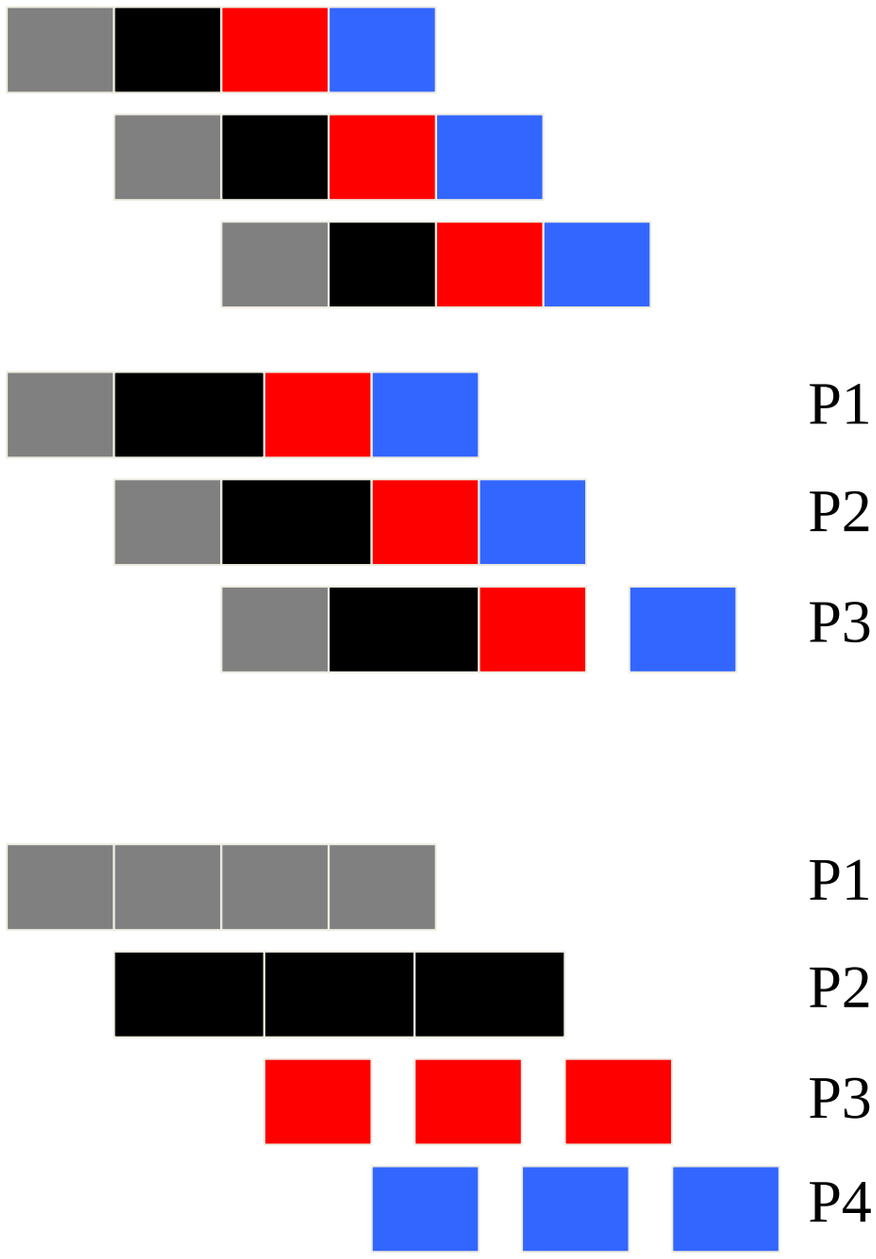
Pipeline Performance



- N data and T tasks
- Each task takes unit time t
- Sequential time = N*T*t = NT (t=1)
- *Parallel pipeline time*
  = start + finish + parallel
  = T-1 + T-1 + (N-2(T-1))/T
  = 2T-2 + N/T + (2T-2)/T = N/T + (2T-2)(1+1/T)
  = O(N/T) (for N>>T)
- Try to find a lot of data to pipeline
- Try to divide computation in a lot of pipeline tasks
  - More tasks to do (longer pipelines) = more parallelism
  - Shorter tasks to do (as a result of breaking apart tasks)
- Interested in pipeline throughput

Pipeline Performance

- *N* data and *T* tasks
- Suppose the tasks execution times are non-uniform
- Suppose a processor is assigned to execute a task
- What happens to the throughput?
- What limits performance?
- Slowest stage limits throughput … Why?
- Little's Law comes into play



Basic Throughput Quantities

- At all levels of the system (register files through networks), there are three fundamental (efficiency-oriented) quantities:
  - **Latency**
    - every operation requires time to execute
    - # stages (tasks) in a pipeline * time for each task
  - **Bandwidth**
    - # of (parallel) operations completed per cycle
  - **Concurrency**
    - total # of operations in flight

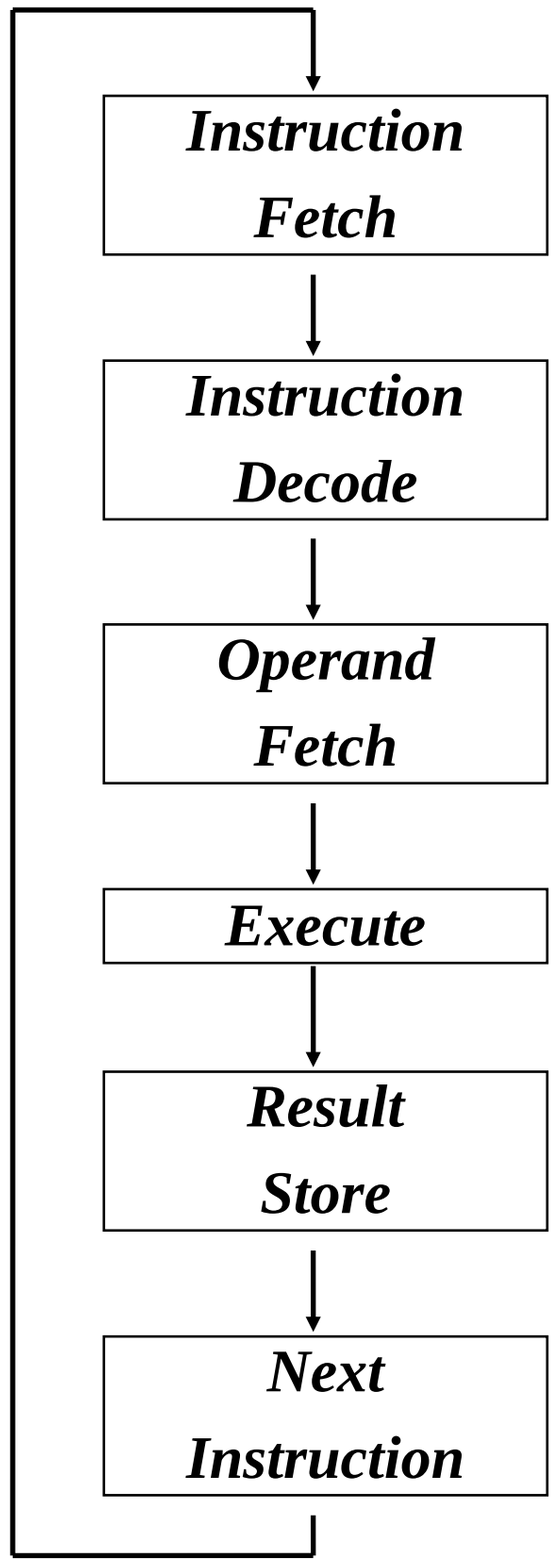
Little's Law

- In queueing theory, Little's Law expresses a relationship between latency and throughput in a stable system (e.g., a pipeline with infinite data)

  *occupancy (concurrency) = latency * throughput*

- Think of a water pipe
  - Throughput = rate at which water is put into the pipe
  - Latency = "length" of the pipe (processing time for data)
- If increase throughput, need more concurrency
  - How do you get it? … increase # pipeline stages
- If latency decreases, need more concurrency

Instruction Execution Cycle



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

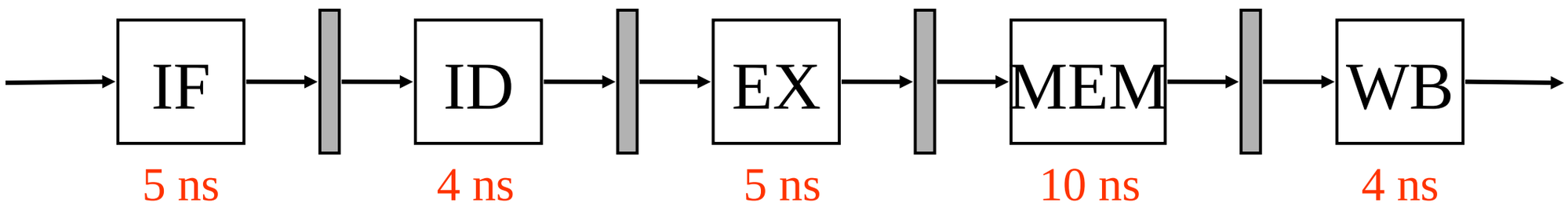
Compute result value or status

Deposit results in storage for later use

Determine successor instruction

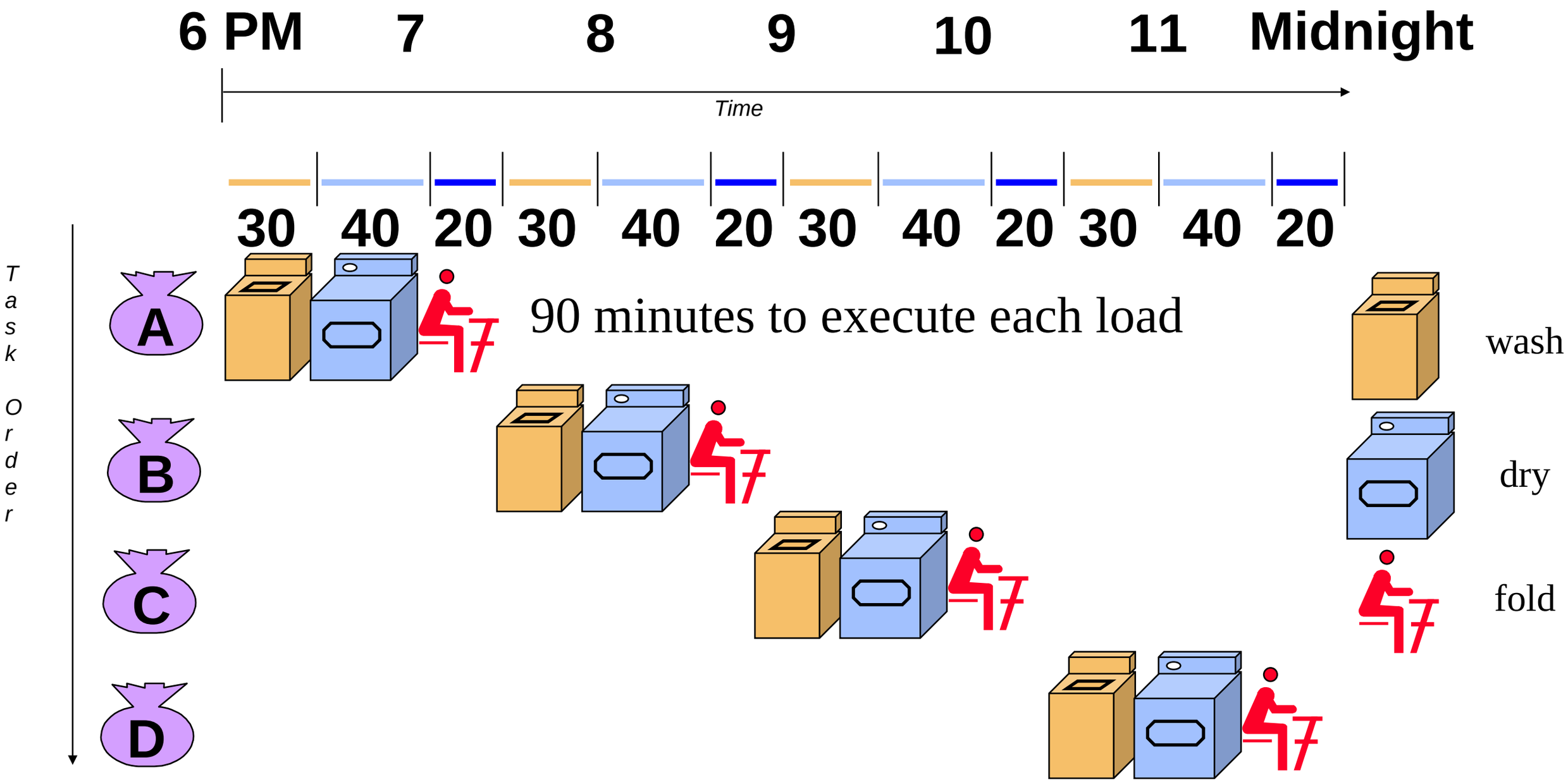
Instruction Execution Cycle (Stages)

- Consider steps for executing an instruction as stages:
  - **IF**: *Instruction Fetch* (load instruction from memory)
  - **ID**: *Instruction Decode*
  - **EX**: *Execution*
  - **MEM**: *Memory Access*
  - **WB**: *Write Back* (write result back to memory)



- Each instruction passes through all of these stages
- Different stages may take different time to execute

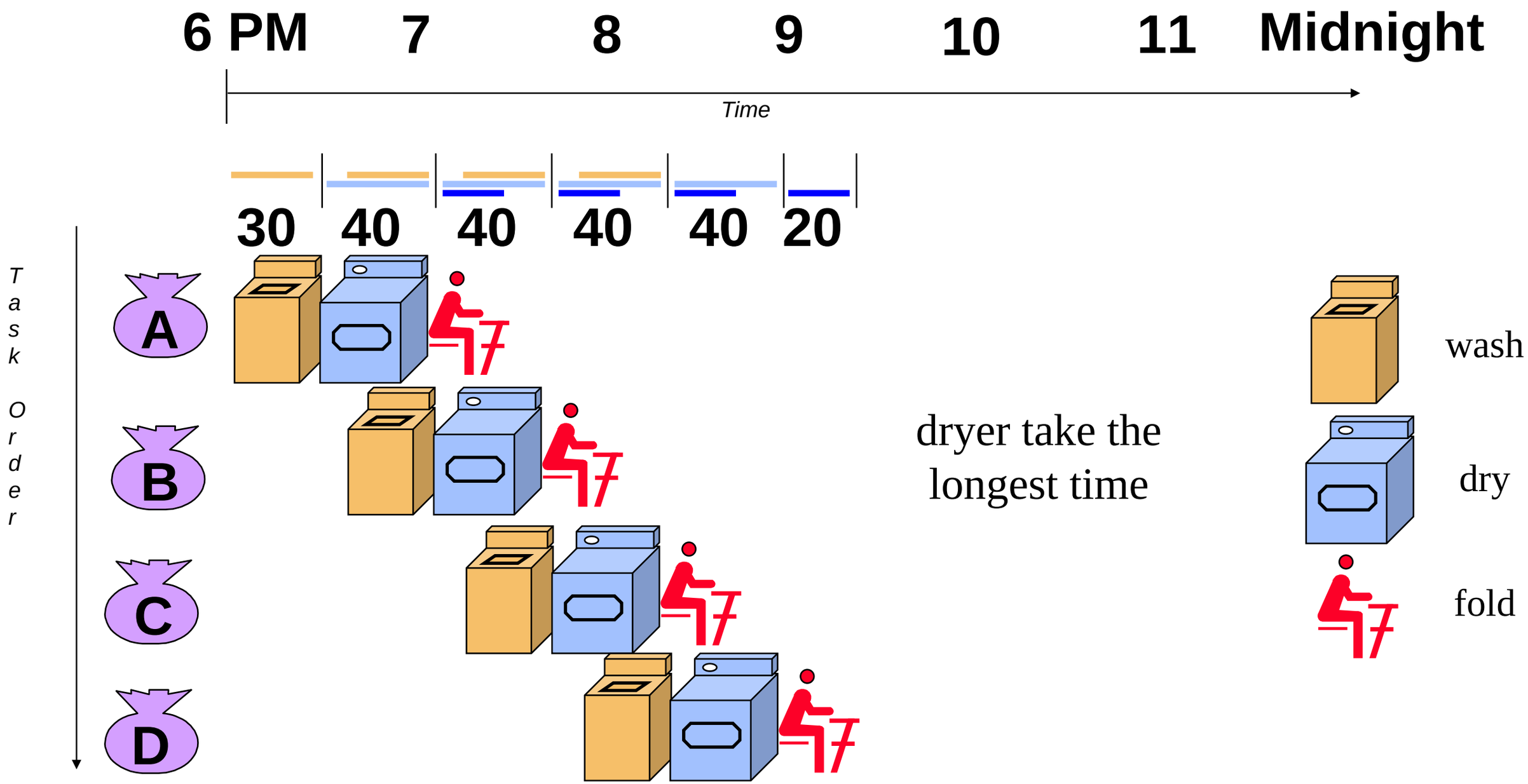
Consider a Sequential Laundry



- Sequential laundry takes 6 hours for 4 loads (jobs)
- How can we make this go faster?

Source : David Patterson

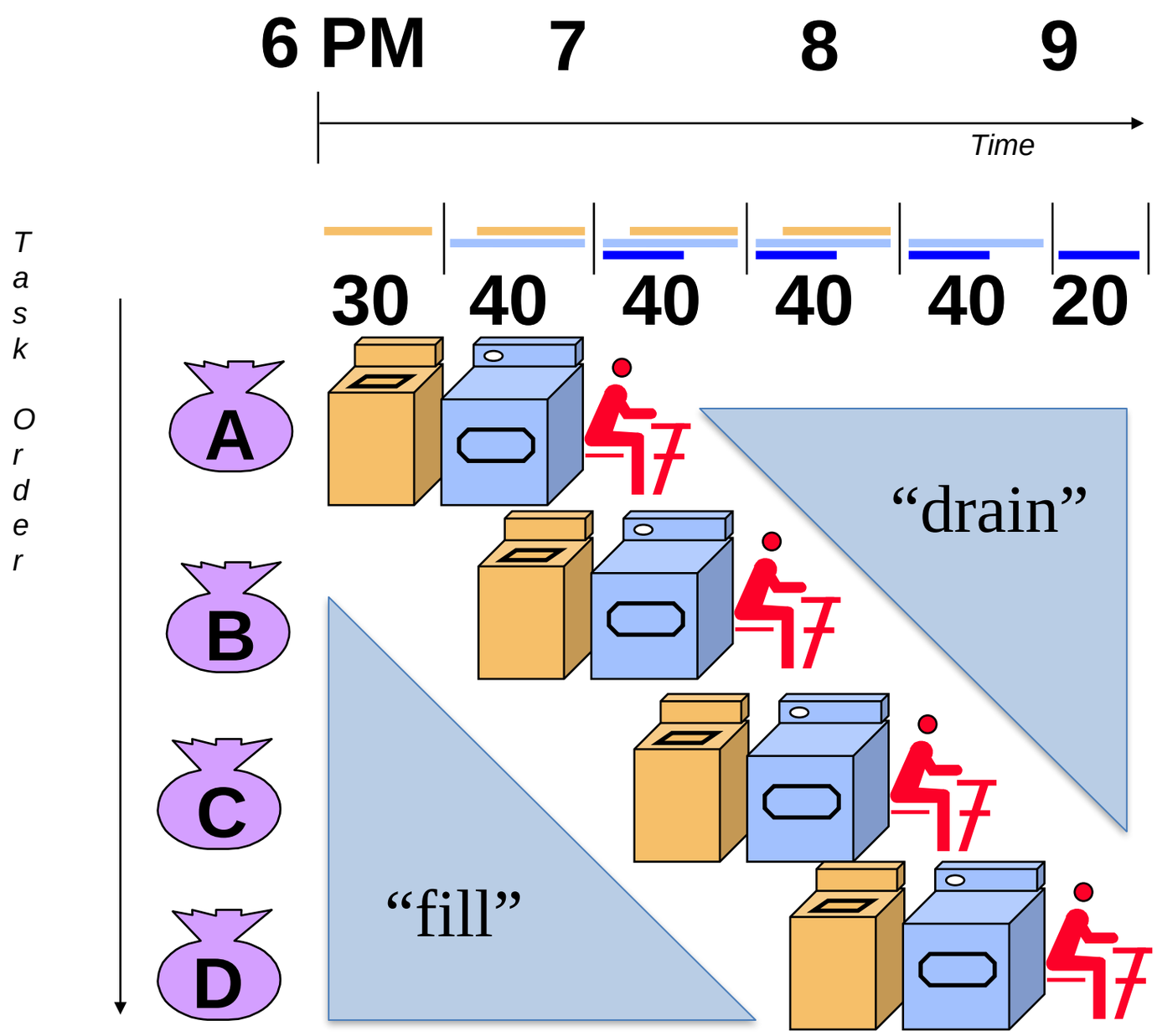
Laundry "Pipeline"



- Use resources as soon as they are available
- Pipelined laundry takes 3.5 hours for 4 loads

Source : David Patterson

Pipelining Lessons



- Pipelining does not help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" / "drain"

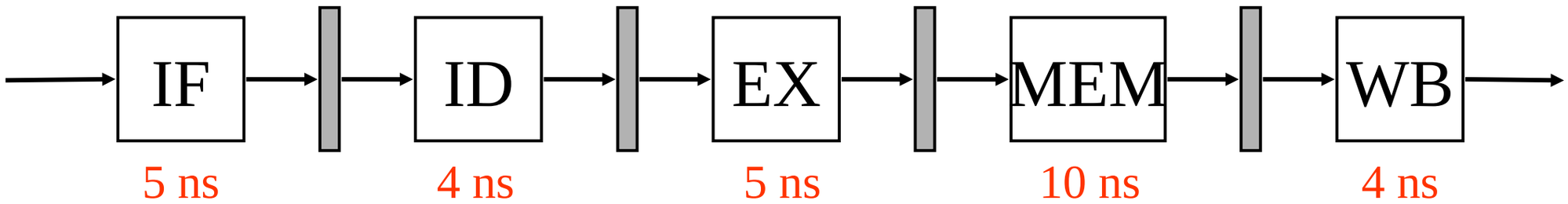
Pipeline Characteristics (for instructions)

- All instructions that are executed go through the same stages of the pipeline
- However, it is possible to overlap the execution of an instruction with other instructions
- *Latency* is the time it takes to execute a job
  - 90 minutes in the laundry case (wash + dry + fold)
- *Throughput* is the total number of jobs that can be completed in a certain amount of time

Source: J.N. Amaral

Sequential Pipeline Flow and Latency

- What is the latency of the pipeline?



L = *lat(IF)* + *lat(ID)* + *lat(EX)* + *lat(MEM)* + *lat(WB)*

= 5ns + 4ns + 5ns + 10ns + 4ns = 28ns

- Assume only 1 instruction is executing at a time
- How long does it take to execute the pipeline?

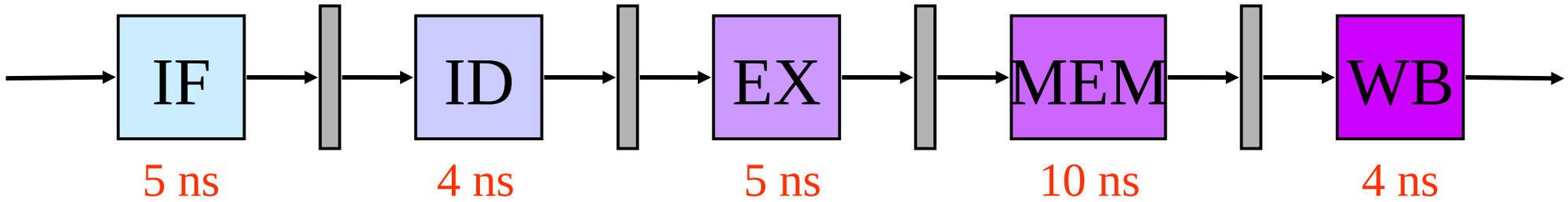
T = IC / L

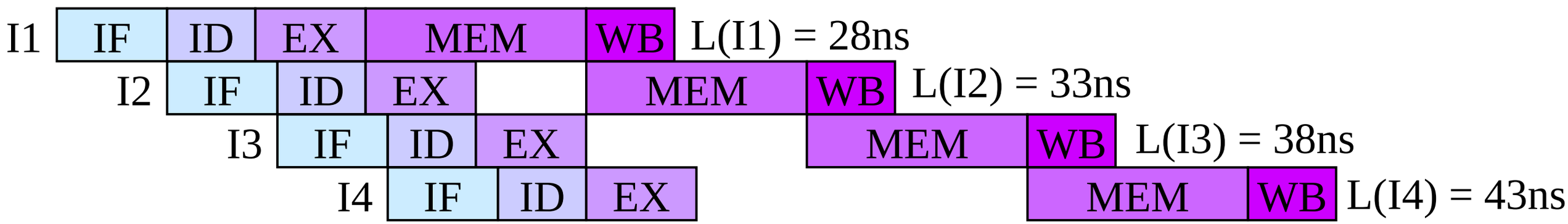
= IC / (5+4+5+10+4) ns

= IC / 28ns

Overlapping Instructions in the Pipeline

- Suppose we try to execute several instructions



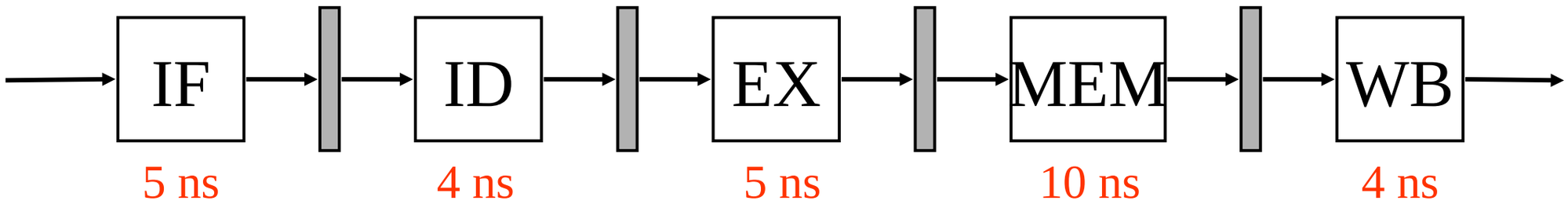


- Do you see any problems?
- Latencies are not constant
- Pipeline stages are not balanced

Source: J.N. Amaral

Pipeline Flow and Latency

- What is the latency of the (sequential) pipeline?



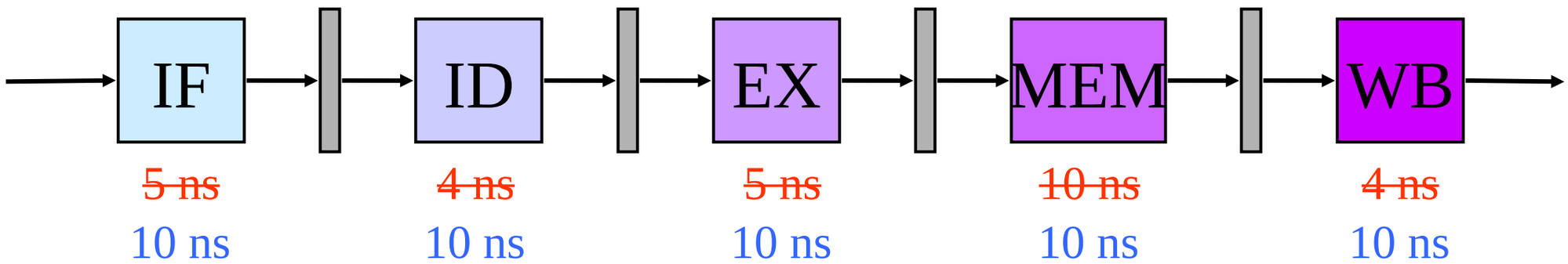
L = *lat(IF)* + *lat(ID)* + *lat(EX)* + *lat(MEM)* + *lat(WB)*
= 5ns + 4ns + 5ns + 10ns + 4ns = 28ns

- What if you overlap instructions?
- How long does it take to execute the pipeline?

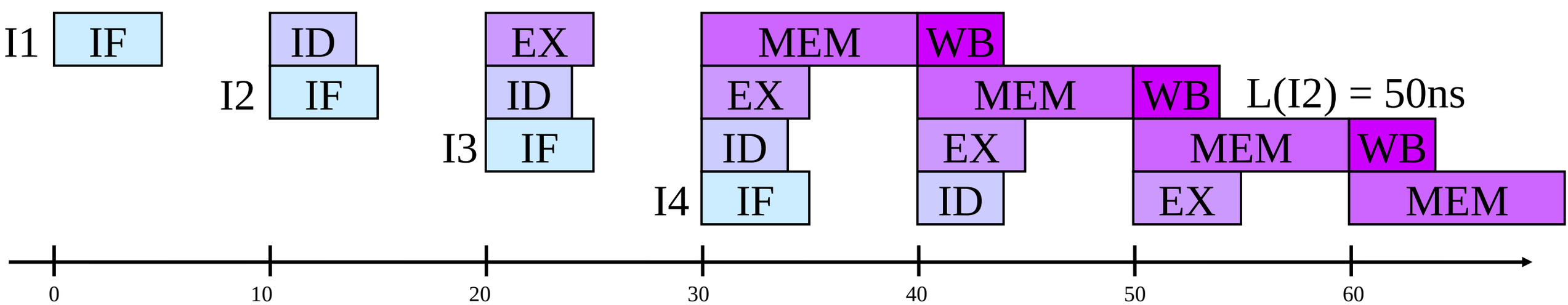
T = IC / max [*lat(IF),lat(ID),lat(EX),lat(MEM),lat(WB)*]
= IC / max [5,4,5,10,4]*ns*
= IC / 10*ns*

Balance the Instruction Pipeline Stages

- Make each instruction stage take the same time
  - Set to the longest stage



- Latencies are now the same ("balanced")

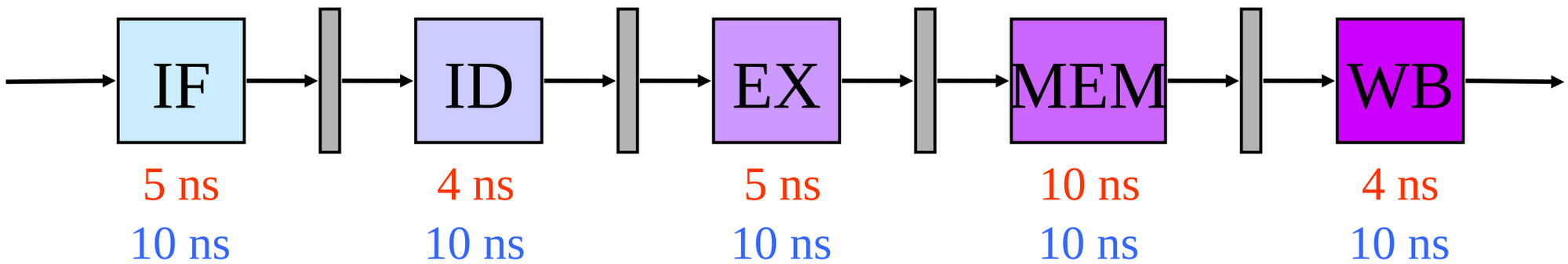


L(I1) = L(I2) = L(I3) = L(I4) = 50ns (> 28ns)

Source: J.N. Amaral

Computing CPU Time

- With a balanced pipeline, it is easier to calculate the CPU execution time for a program



- Suppose we execute 20000 instructions

$$T_{non-pipe} = 20000 \times 28ns = 560000ns = 560\mu s$$

$$T_{pipe} = 50 + 19999 \times 10ns = 200040ns = 200{,}04\mu s$$

Cost for startup and shutdown

Source: J.N. Amaral

Pipelining Speedup (Acceleration)

- Pipelining can deliver a performance improvement
- Measure by speedup:

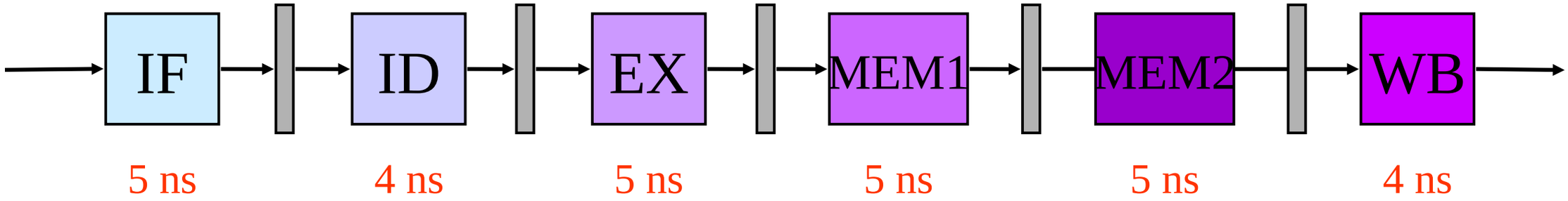
$$Speedup_{pipe} = \frac{ExecTime_{non\text{-}pipe}}{ExecTime_{pipe}} = \frac{560\mu s}{200{,}04\mu s} \approx 2.8$$

- Pipeline performance can be improved by:
  - Increasing the # stages
  - Decreasing the time of the slowest stage
  - Reducing the imbalance between stages

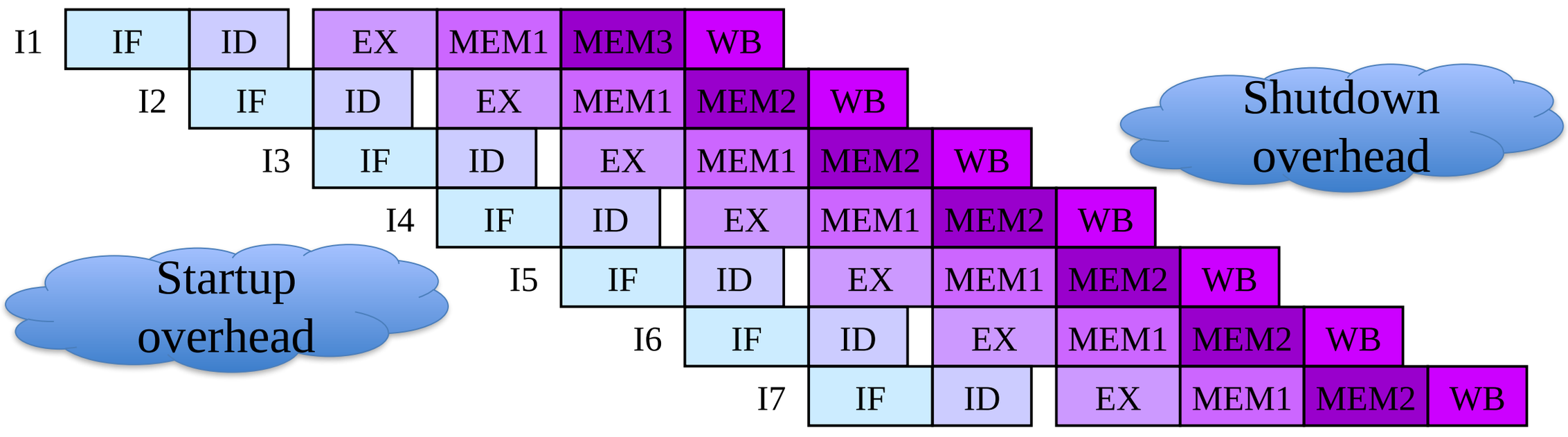
Source: J.N. Amaral

Split a Stage to Reduce Maximum Delay

- Suppose we split the memory access stage



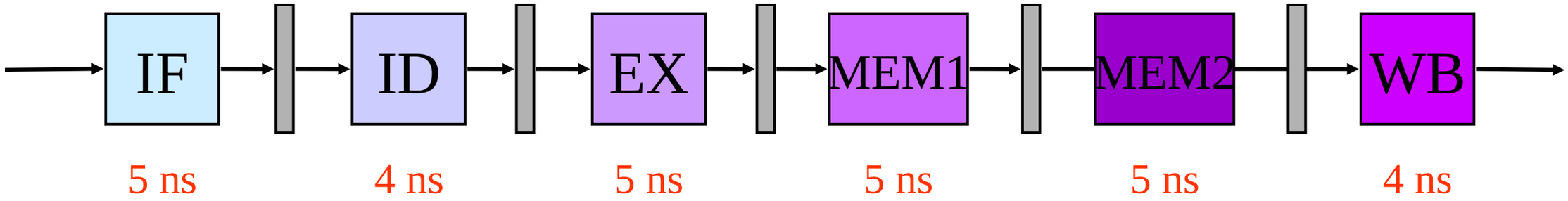
- Sequential latency is the same
- Pipeline execution of multiple instructions



Source: J.N. Amaral

Split a Stage to Reduce Maximum Delay

- Suppose we split the memory access stage



- Sequential latency is the same
- Pipeline (balanced) latency is reduced

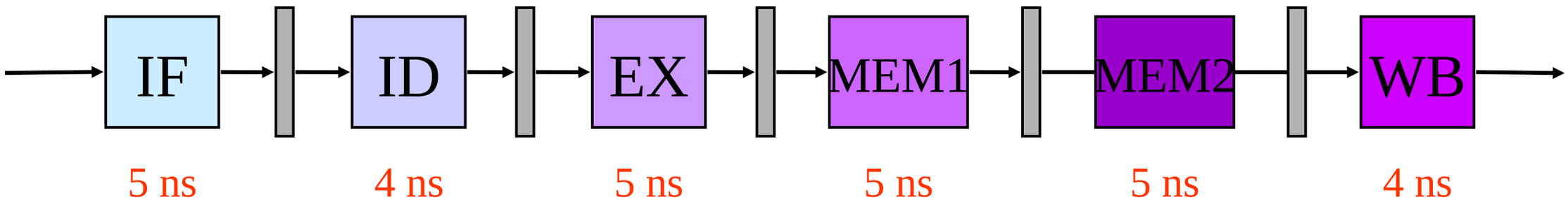
$$L = 6 \times 5ns = 30ns$$

- Total execution time is smaller (by 2 times!)

T = IC / max [*lat(IF),lat(ID),lat(EX),lat(M1),lat(M2),lat(WB)*]
= IC / max [5,4,5,5,5,4]*ns*
= IC / 5*ns*

Recalcuate the Pipeline Speedup

- Now pipeline stages have less delay variance



- We can recalculate the total time and speedup

$$ExecTime_{pipe} = 30 + 19999 \times 5ns = 100000ns = 100{,}25\mu s$$

$$Speedup_{pipe} = \frac{ExecTime_{non\text{-}pipe}}{ExecTime_{pipe}} = \frac{560\mu s}{100{,}025\mu s} \approx 5.6$$

- Notice that the startup and shutdown overhead improved because of the smaller maximum stage

Source: J.N. Amaral

Theoretical Performance

- Pipeline speedup is determined by
  - # stages (N)
  - Maximum stage delay (D)

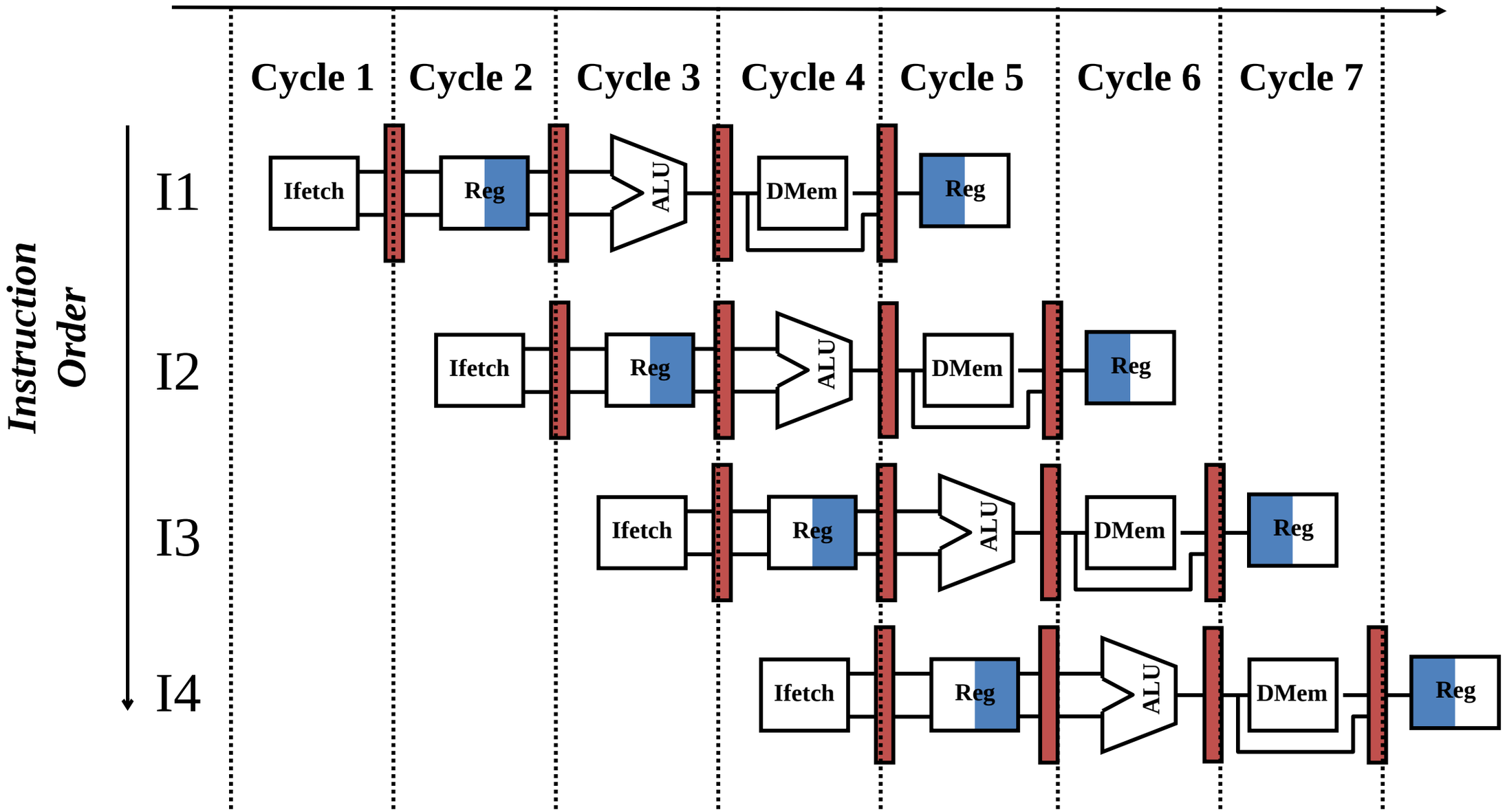
- Flow rate of the pipeline is $1/D$
- Latency of the pipeline is $N*D$

- Ignoring startup and shutdown, the maximum speedup of a pipeline is N

Source: J.N. Amaral

Instruction Execution Pipeline Hardware

- Consider the hardware for instruction execution
  - Resources required for operation



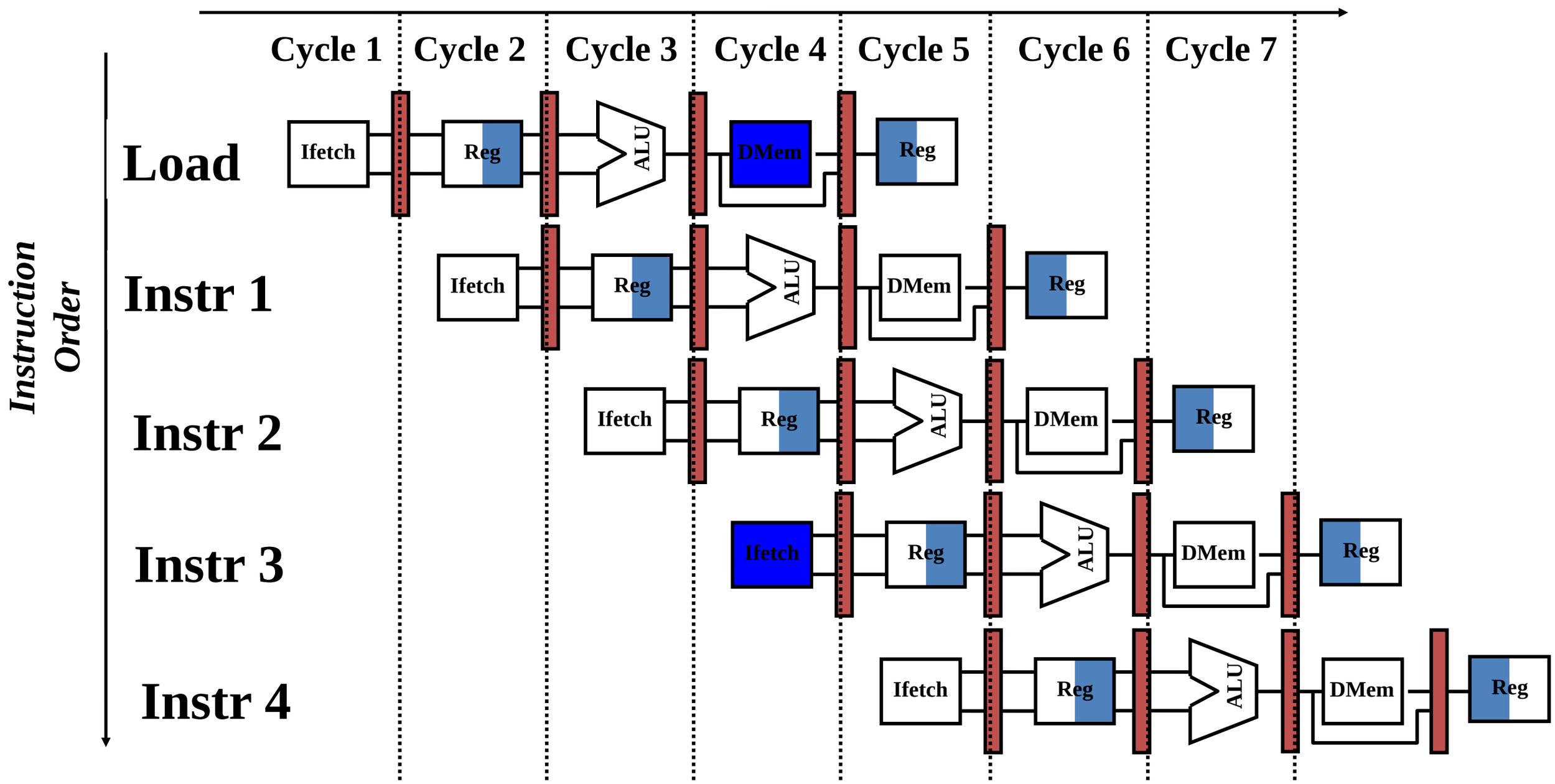
Pipeline Limitations

- Hardware that implements the instruction pipeline must be used for multiple instructions
- Limited hardware resources will place constraints on instruction execution
- Certain situations (*hazards*) can prevent the next statement from executing during its clock cycle
  - *Structural hazards*: hardware can not support a combination of instructions (same hardware resource is accessed)
  - *Data hazards*: execution depends on the result of a previous statement that is still in the pipeline
  - *Control hazard*: loading of instructions is delayed by a decision of change of control flow

Source: J.N. Amaral

Structural Hazards

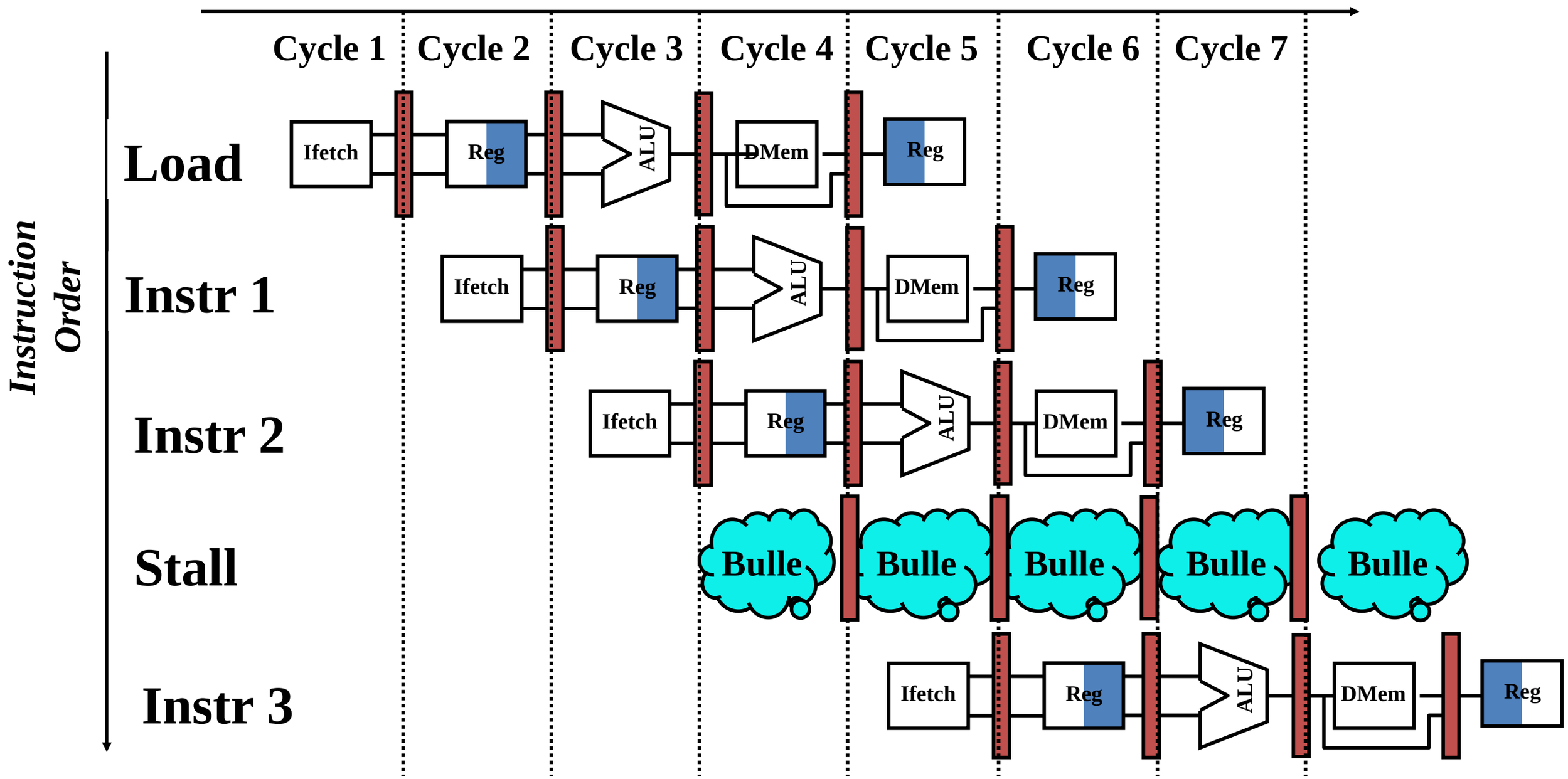
- Instruction fetch and data RW/WR can conflict



Source: J.N. Amaral

Structural Hazards (2)

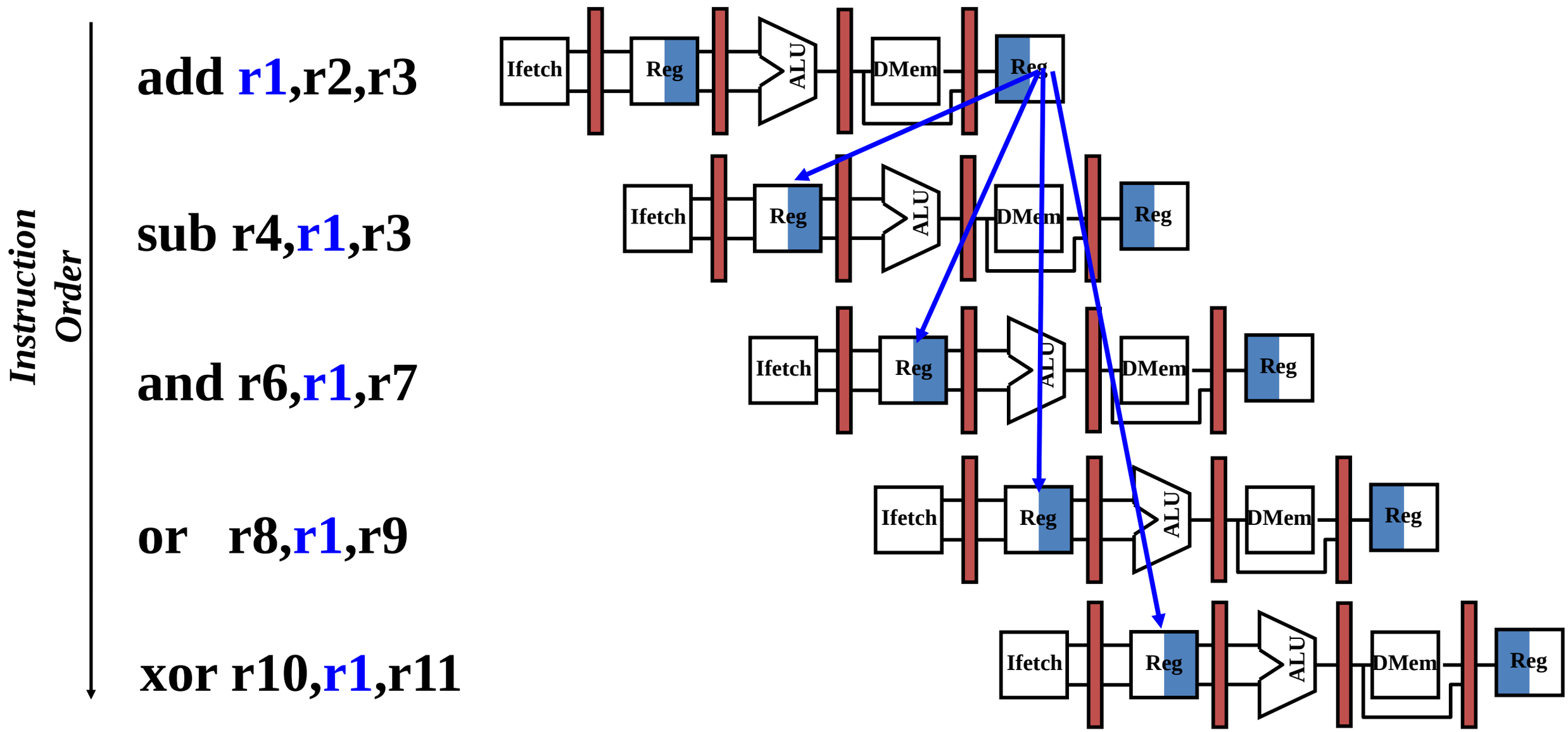
- Hazard causes a stall
  - A “bubble” is inserted and flows through the pipeline



Source: J.N. Amaral

Data Hazards

- CPU registers are used for fast data access
  - # registers is finite and there can be conflicts



Data Dependencies and Pipeline Control

- *Read After Write (RAW)* : flow dependency

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- *Write After Read (WAR)* : anti-dependency

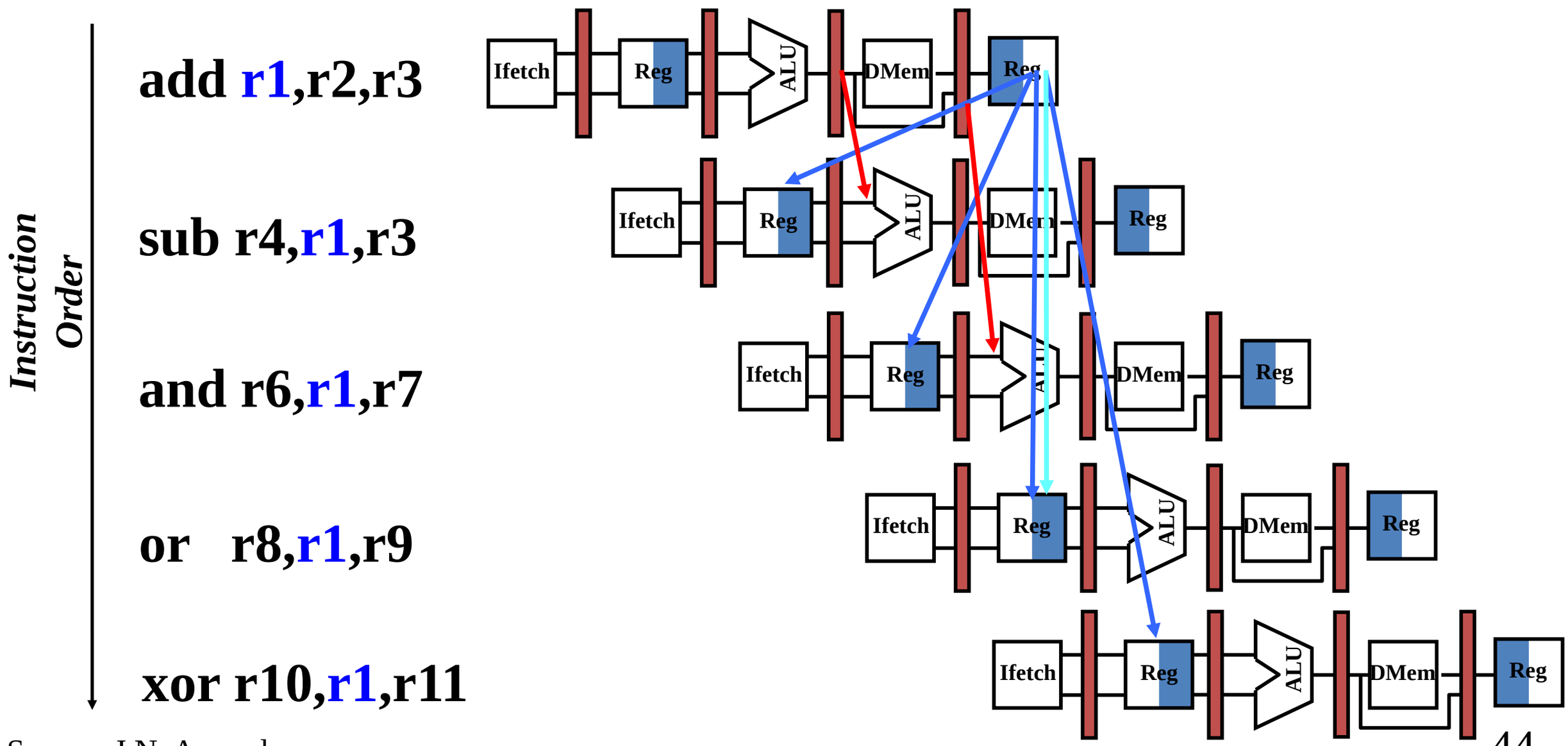
```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- *Write After Write (WAW)* : output dependency

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

Avoiding Data Hazards w/ Pipeline Control

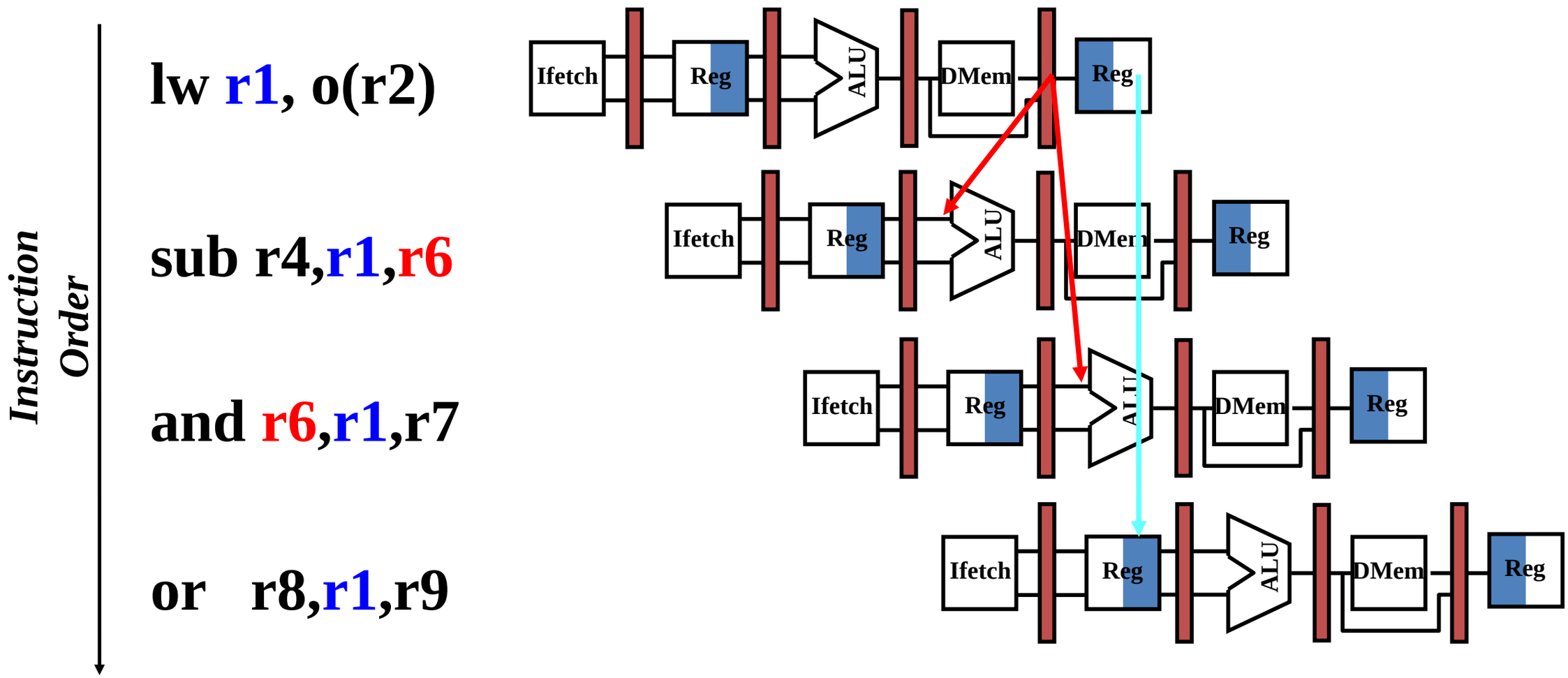
- Short circuit (link) pipeline stages with:
  - *Bypassing* – send value to directly to ALU input
  - *Forwarding* – use 2nd half of register write for read



Source: J.N. Amaral

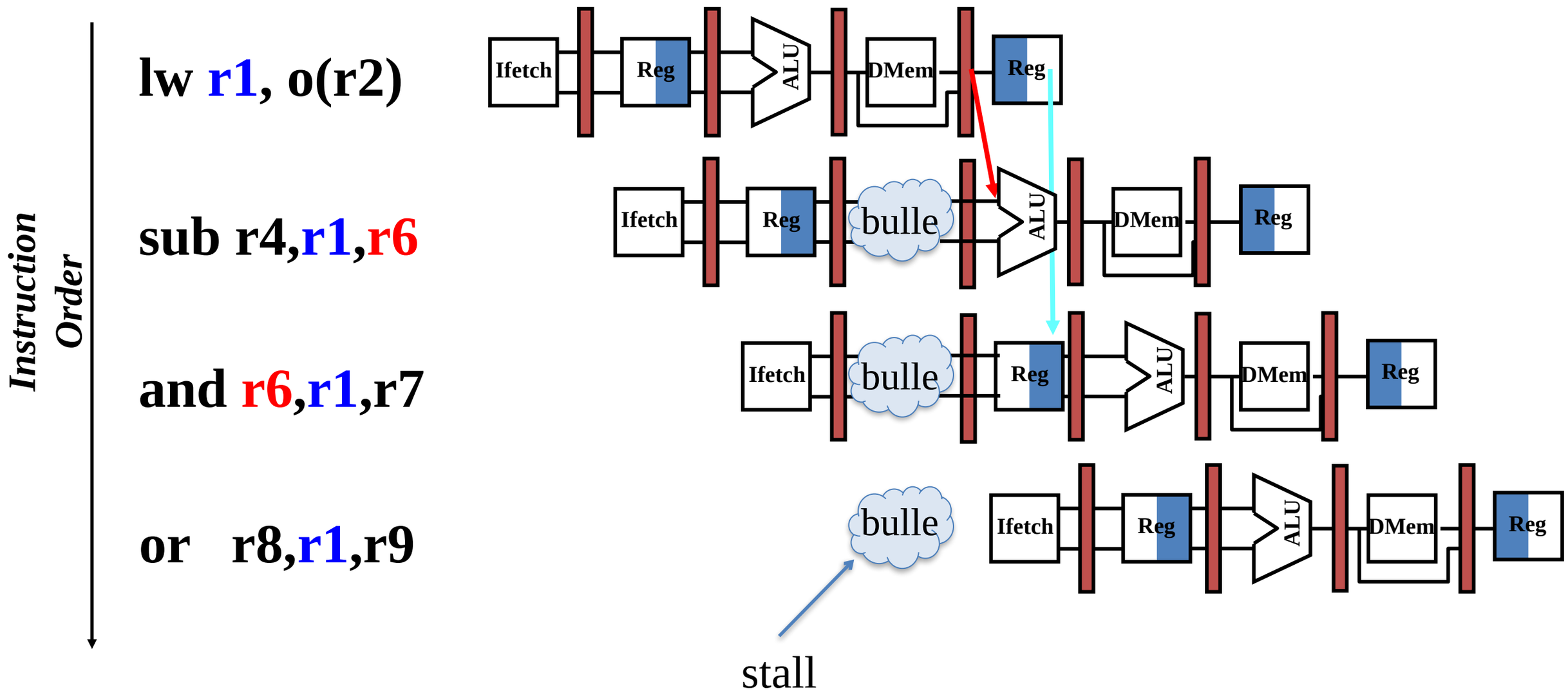
Avoiding Data Hazards (2)

- It might not be possible to bypass or forward without introducing a stall



Avoiding Data Hazards (3)

- Stall shifts the pipeline stages to make bypassing and forwarding possible



Avoid Data Hazards in Code Generation

- Suppose there is the following code:

  a = b + c;

  d = e - f;

  - Assume a, b, c, d, e, f are stored in memory
- Try to generate code

Slow:

```
LW Rb, b
LW Rc, c
ADD Ra, Rb, Rc
SW a, Ra
LW Re, e
LW Rf, f
SUB Rd, Re, Rf
SW d, Rd
```

Faster:

```
LW Rb, b
LW Rc, c
LW Re, e
ADD Ra, Rb, Rc
LW Rf, f
SW a, Ra
SUB Rd, Re, Rf
SW d, Rd
```

Why is this faster?

Try to Avoid Hazards in Code Generation

- Look at the different assembly code and identify the hazards in each and compare

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LW Rb, b	IF	ID	EX	MEM	WB										
LW Rc, c		IF	ID	EX	MEM	WB									
ADD Ra, Rb, Rc			IF	ID	stall	EX	MEM	WB							
SW a, Ra				IF	stall	ID	EX	MEM	WB						
LW Re, e					stall	IF	ID	EX	MEM	WB					
LW Rf, f							IF	ID	EX	MEM	WB				
SUB Rd, Re, Rf								IF	ID	stall	EX	MEM	WB		
SW d, Rd									IF	stall	ID	EX	MEM	WB	

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LW Rb, b	IF	ID	EX	MEM	WB										
LW Rc, c		IF	ID	EX	MEM	WB									
LW Re, e			IF	ID	EX	MEM	WB								
ADD Ra, Rb, Rc				IF	ID	EX	MEM	WB							
LW Rf, f					IF	ID	EX	MEM	WB						
SW a, Ra						IF	ID	EX	MEM	WB					
SUB Rd, Re, Rf							IF	ID	EX	MEM	WB				
SW d, Rd								IF	ID	EX	MEM	WB			

Source: J.N. Amaral

Code Scheduling

- The compiler can try to *schedule* the pipeline to avoid stalls due to data hazards
  - Rearranging the code sequence to eliminate the hazard
  - Referred to as *pipeline* or *instruction* scheduling
- Takes into account what is known about how the CPU pipeline hardware operates
- In addition to being useful for data hazards, it is important for control hazards
- This is known as *static scheduling* because it is done before program execution

Pipelines with Multicycle Operations

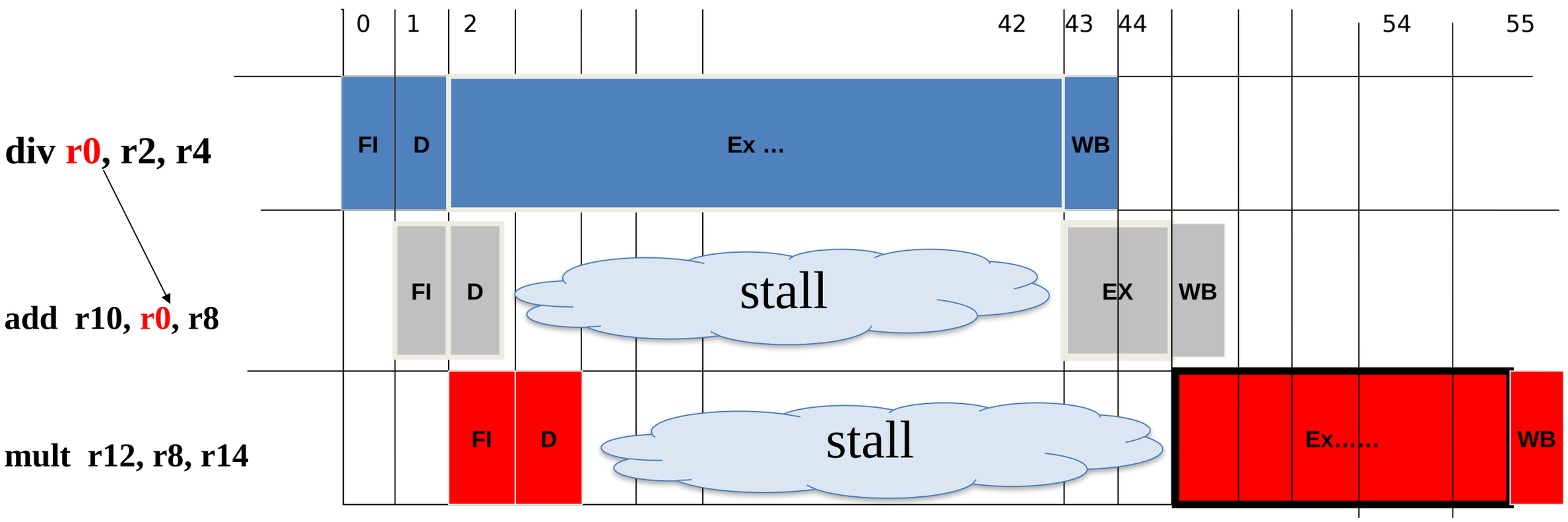
- Some operations (like floating point) are difficult to do in a single clock cycle
  - Often require multiple clock cycles
  - Hardware complexity is more severe
- Multiple functional units can be used
- It is also possible to pipeline a functional unit
  - It is implemented in stages
  - Allows new operations to enter the functional unit as soon as the initial stage is available

Definitions

- *In-order issue* (fetch)
  - All instructions MUST be issued (fetched) in the same order as they appear in the program
- *In-order execution*
  - All instructions MUST enter their execution cycle in the same order as they appear in the program
- *In-order completion*
  - All instructions MUST exit (complete) the pipeline in the same order as they appear in the program
- *Out-of-order execution*
  - Instructions are not required to enter their execution cycle in the same order as they appear in the program
- *Out-of-order completion*
  - Instructions are not required to exit (complete) the pipeline in the same order as they appear in the program

In-Order Instruction Execution

- All instructions MUST execute their pipeline stages in the same order as they appear in the program
- Consider a 4-stage pipeline: IF, D, EX, WB



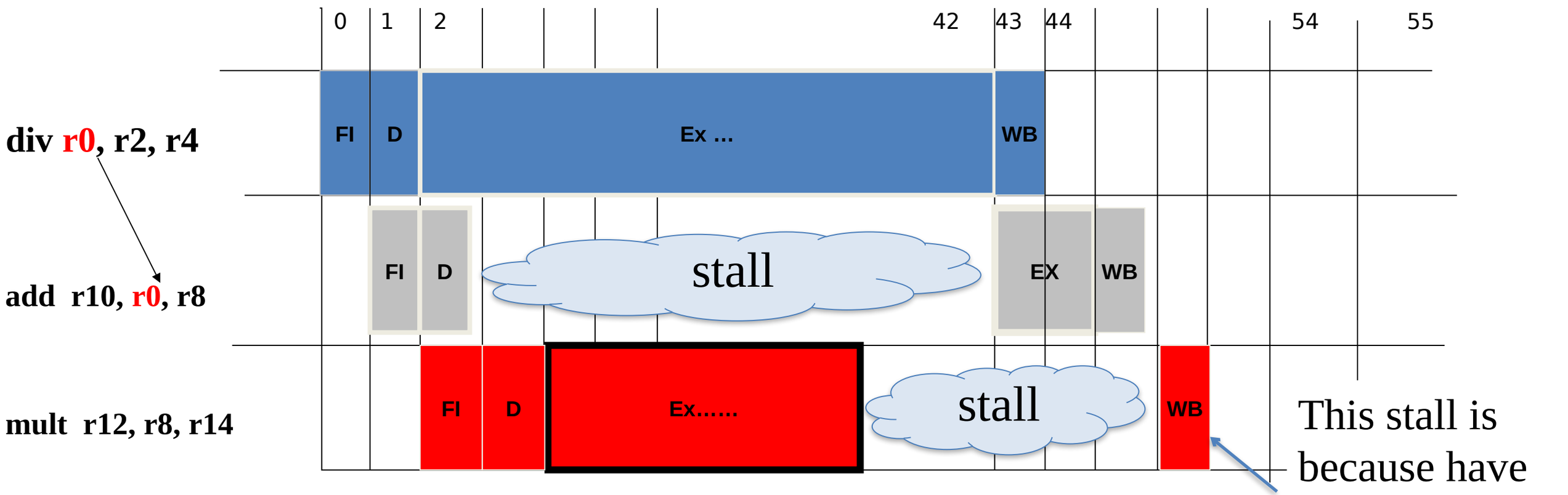
40 cycles div, 2 cycles add , 10 cycles mult

Total execution: 55 cycles

Source: Smail Niar

Out-of-Order Instruction Execution

- Given separate divide and multiply functional units
- It is possible to use both units at the same time
- It must be possible for instructions to execute pipeline stages out-of-order relative to each other



Total execution: 46 cycles (versus 55)

Source: Smail Niar

Data Hazards and Dynamic Scheduling

- If there is a data dependence that cannot be hidden by forwarding, the hazard detection hardware will stall the pipeline
- Static scheduling can help to minimize the actual hazards and resultant stalls
- An early improvement in processor architecture used *dynamic scheduling* to detect and control dependencies during pipeline execution
  - Works with multiple functional resources
  - Manages pipelined, out-of-order instruction execution
  - Significant increase in hardware complexity

Dynamic Scheduling Concept (1)

- If instruction *j* depends on a long-running instruction *i* currently executing in the pipeline, then all instructions after *j* must be stalled until *i* finishes and j can execute.
- Instead, we would like an instruction to begin execution as soon as its data operands are available
- To do so, the pipeline must be able to do out-of-order execution
- Out-of-order execution => out-of-order completion

Dynamic Scheduling Concept (2)

- Split ID into 2 stages:
  - *Issue*: decode instruction, check for structural hazards
  - *Read operands*: wait until no data hazards, then read operands
- Distinguish between when instruction *begins execution* and when it *completes execution*
  - Inbetween these two the instruction is *in execution*
- Given hardware support, multiple instructions can be in execution at the same time

Dynamic Scheduling with a Scoreboard

- In a dynamically scheduled pipeline, all instructions pass through the *issue* stage in-order
- However, they can be stalled or bypass each other in the *read operands* stage
- A technique called *scoreboarding* allows instructions to execute out-of-order when there are:
  - Sufficient resources (there can be multiple resources)
  - No data dependences (all operands are available)
- The goal of the scoreboard is to achieve a CPI=1 when there are no structural hazards

Scoreboarding Operation

- Scoreboarding takes full responsibility for instruction issue and execution, including all hazard detection
- Multiple functional units are needed since multiple instructions will be in their execution stage at the same time
- Scoreboard keeps a dynamic record of data dependencies
- Scoreboard determines when the instruction can read its operands and begin execution
- If an instruction cannot execute immediately, the scoreboard monitors every change in the HW and decides when the instruction can execute
- Scoreboard controls when an instruction can write its result to a destination register

Three Parts of a Scoreboard

- Instruction status
  - Which of the 4 steps the instruction is in
- Functional unit status
  - Indicates the state of the functional units
  - *Busy*: indicates whether the unit is busy or not
  - *Op*: operation to perform on the unit
  - *Fi*: destination register
  - *Fj*, *Fk*: source registers
  - *Qj*, *Qk*: functional units producing *Fj*, *Fk*
  - *Rj*, *Rk*: flags indicating when *Fj*, *Fk* are ready
- Register result status
  - Indicates which functional unit will write each register

Scoreboard Factors

- A scoreboard is limited by:
  - Amount of instruction independence
    - *true* data dependencies can not be avoided
  - Number of scoreboard entries
    - this determines how far ahead the pipeline can look for independent instructions
    - this set of instructions is call the *window*
  - Number and types of functional units
  - Presence of anti- (WAR) and output (WAW) dependencies

CDC 6600 Scoreboard

- The CDC 6600 was the first machine to conceive of using a scoreboard
- Organization
  - 16 functional units
    - 4 FP, 5 Mem, 7 Int
  - Load-store memory
    - not registers!
  - 4 groups of functional units each with a data trunk
  - Only 1 unit in a group could use the data trunk at a time
- CDC 6600 was a very early machine

Tomasulo Algorithm

- CPU architecture for out-of-order execution
- Hardware technique developed by Robert Tomasulo in the 1960's at IBM for the 360/91
  - Improved on the notion of *scoreboarding* as a technique to keep track of dependencies and functional unit use

- Key ideas
  - Treat memory operations as functional units
  - Abstract functional units to *reservation stations*
  - Renaming to eliminate false dependencies
- Limited hardware (no caches, small # FP registers)

Tomasulo Organization

- Each instruction is divided into 3 steps
  - *Issue*: taking instruction into account and waiting for input operands
  - *Execution*: Effective execution of the operation on a functional unit
  - *Write*: writing the result and passing information to the following instructions
- Technology based on a *Common Data Bus* (CDB) for the transfer of information
- Use of reservation stations

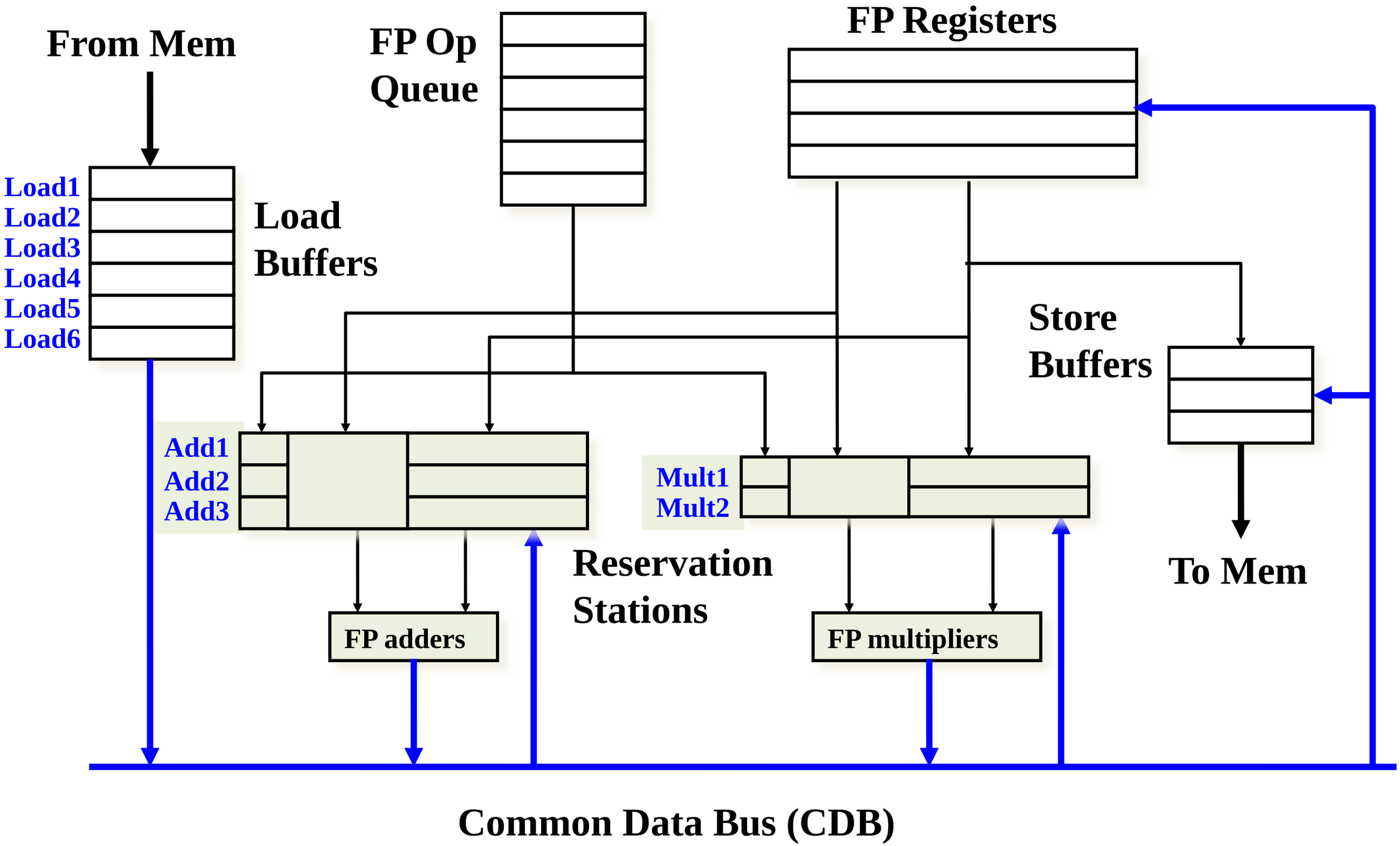
Reservation Stations

- Abstract functional units by adding input buffers
  - Can store more instructions waiting for an operand
- Components of a station = an instruction
  - *Busy*: Is this station busy?
  - *Op*: Assembly operation using this station
- Management of the operands for instruction
  - *Vj*, *Vk*: value of the operands already present
  - *Qj*, *Qk*: name of the reservation station that will produce the operand

Register Bank Extension

- Register bank contains all the processor registers
- Addition of a flag to know which reservation station will produce a value *(Qi)*
- Result
  - Dynamic renaming of registers
  - Suppression of false dependencies (WAR and WAW)

Tomasulo Organization



Tomasulo Example

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time Name	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
Add1	No					
Add2	No					
Add3	No					
Mult1	No					
Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
0	*FU*									

Source: D. Culler

Tomasulo Example Cycle 1

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations:

Time Name	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
Add1	No					
Add2	No					
Add3	No					
Mult1	No					
Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
1	*FU*				Load1					

Source: D. Culler

Tomasulo Example Cycle 2

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
2	*FU*		Load2		Load1					

Note: Can have multiple loads outstanding

Source: D. Culler

Tomasulo Example Cycle 3

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
3	*FU*	Mult1	Load2		Load1					

Note: registers names are removed ("renamed") in Reservation Stations; MULT issued

Load1 completing; what is waiting for Load1?

Source: D. Culler

Tomasulo Example Cycle 4

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	Yes	SUBD	M(A1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
4	*FU*	Mult1	Load2		M(A1)	Add1				

Load2 completing; what is waiting for Load2?

Source: D. Culler

Tomasulo Example Cycle 5

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
5	*FU*	Mult1	M(A2)		M(A1)	Add1	Mult2			

Timer starts down for Add1, Mult1

Source: D. Culler

Tomasulo Example Cycle 6

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
6	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

Issue ADDD here despite name dependency on F6?

Source: D. Culler

Tomasulo Example Cycle 7

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
7	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

Add1 (SUBD) completing; what is waiting for it?

Source: D. Culler

Tomasulo Example Cycle 8

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock **8**

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
FU	Mult1	M(A2)		Add2	(M-M)	Mult2			

Source: D. Culler

Tomasulo Example Cycle 9

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
9	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Source: D. Culler

Tomasulo Example Cycle 10

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
10	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Add2 (ADDD) completing; what is waiting for it?

Source: D. Culler

Tomasulo Example Cycle 11

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

	Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
		Add1	No					
		Add2	No					
		Add3	No					
	4	Mult1	Yes	MULTD	M(A2)	R(F4)		
		Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
11	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Write result of ADDD here?

All quick instructions complete in this cycle!

Source: D. Culler

Tomasulo Example Cycle 12

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
		Add1	No					
		Add2	No					
		Add3	No					
	3	Mult1	Yes	MULTD	M(A2)	R(F4)		
		Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
12	FU	Mult1	M(A2)		(M-M+N	(M-M)	Mult2			

Source: D. Culler

Tomasulo Example Cycle 13

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
		Add1	No					
		Add2	No					
		Add3	No					
	2	Mult1	Yes	MULTD	M(A2)	R(F4)		
		Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
13	FU	Mult1	M(A2)		(M-M+N	(M-M)	Mult2			

Source: D. Culler

Tomasulo Example Cycle 14

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
14	FU	Mult1	M(A2)		(M-M+N	(M-M)	Mult2			

Source: D. Culler

Tomasulo Example Cycle 15

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
15	*FU*	Mult1	M(A2)		(M-M+N	(M-M)	Mult2			

Mult1 (MULTD) completing; what is waiting for it?

Source: D. Culler

Tomasulo Example Cycle 16

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
16	*FU*	M*F4	M(A2)		(M-M+N	(M-M)	Mult2			

Just waiting for Mult2 (DIVD) to complete

Source: D. Culler

Tomasulo Example Cycle 55

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
1	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
55	*FU*	M*F4	M(A2)		(M-M+N	(M-M)	Mult2			

Source: D. Culler

Tomasulo Example Cycle 56

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
56	*FU*	M*F4	M(A2)		(M-M+N	(M-M)	Mult2			

Mult2 (DIVD) is completing; what is waiting for it?

Source: D. Culler

Tomasulo Example Cycle 57

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	57
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
56	*FU*	M*F4	M(A2)		(M-M+N	(M-M)	Result			

Once again: In-order issue, out-of-order execution and out-of-order completion.

Source: D. Culler

Tomasulo Benefits and Drawbacks

- Benefits
  - Register renaming effectively allowed building data flow dependency graph on the fly
  - Elimination of stalls for WAW and WAR hazards
  - Effectively achieves dynamic loop unrolling
  - Reservation stations overcome limitations of pure scoreboarding
- Drawbacks
  - Hardware complexity
  - Many associative stores (CDB) at high speed
  - Performance limited by Common Data Bus
    - each CDB must go to multiple functional units
    - completion of functional units limited to one per cycle
  - Non-precise

Instruction-Level Parallelism (ILP)

- Instruction-level parallelism (ILP) is the overlapped execution of instructions (i.e., execution of instructions in parallel)

$$CPI_{pipe} = CPI_{ideal} + Stalls_{structural} + Stalls_{RAW} + Stalls_{WAR} + Stalls_{WAW} + Stalls_{control}$$

- Exploit independence among instruction sequences
- Amount of parallelism available within a basic block is quite small
- Must exploit ILP across multiple basic blocks
- Control hazards are serious limitations to ILP

Consider a Code with Branches

- If the branch is taken, 3 cycles are lost
- Question is which control path is more likely

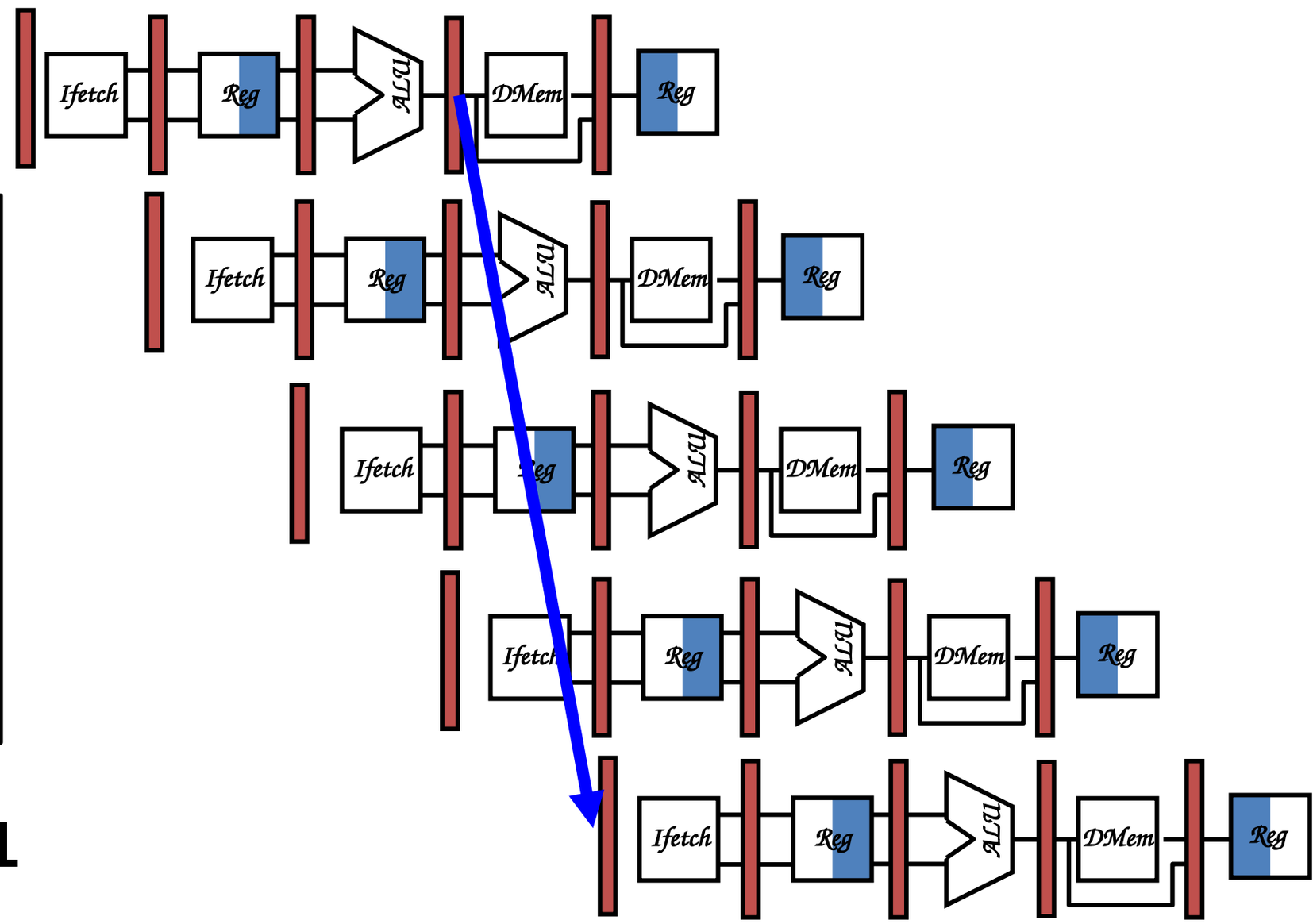
```
10: beq r1,r3,36

14: and r2,r3,r5

18: or  r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11
```



Source: J.N. Amaral

Branch Penalties

- If 30% of control decisions is to branch, a penalty of 3 cycles is not negligible
- Two-part solution:
  - Determine early if the branch it to me made
  - Calculate the branch target address as soon as possible
- Typically, a branch tests a register (= 0 or ≠ 0)
- Possible approach:
  - Test on zero is moved to the ID / RF stage
  - Addition of an adder to calculate the new PC in the ID / RF stage

Four Approaches for Control Hazards (1)

- Approach 1: Wait until the branch direction is identified
- Approach 2: Predict that the branch is not taken
  - Execute successive instructions in sequence
  - Purge instructions in the pipeline if the branch was made
  - Runs instructions in sequence is more efficient
- Approach 3: Predict that the connection would be made
  - Branch would not have yet calculated their target address
  - Results in a 1 cycle penalty

Source: J.N. Amaral

Four Approaches for Control Hazards (2)

- Approach 4: Delay the branch
  - Let *N* instructions execute after the branch instruction
  - Then branch if necessary
  - Violate the sequential semantics

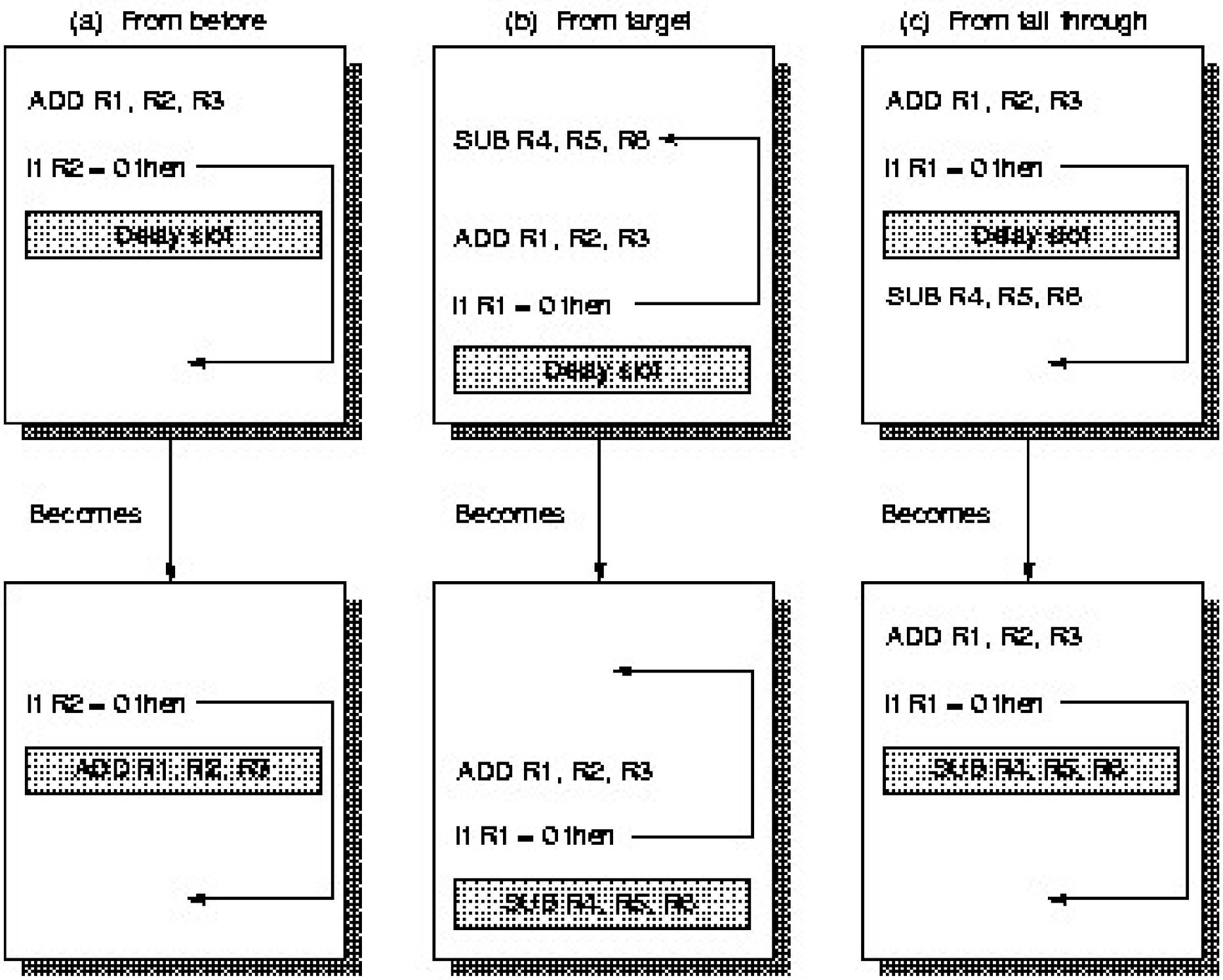
Source: J.N. Amaral

Delayed Branches (1)

- The compiler or programmer must find N instructions to fully exploit the delay
  - Filling the *delay slot*
- Where can these instructions be found?
  - Take the n instructions that precede it sequentially
  - Take N instructions from the target address
    - must be only for unconditional, or speculated as taken
  - Take an instruction that follows it

Source: J.N. Amaral

Delayed Branches (2)



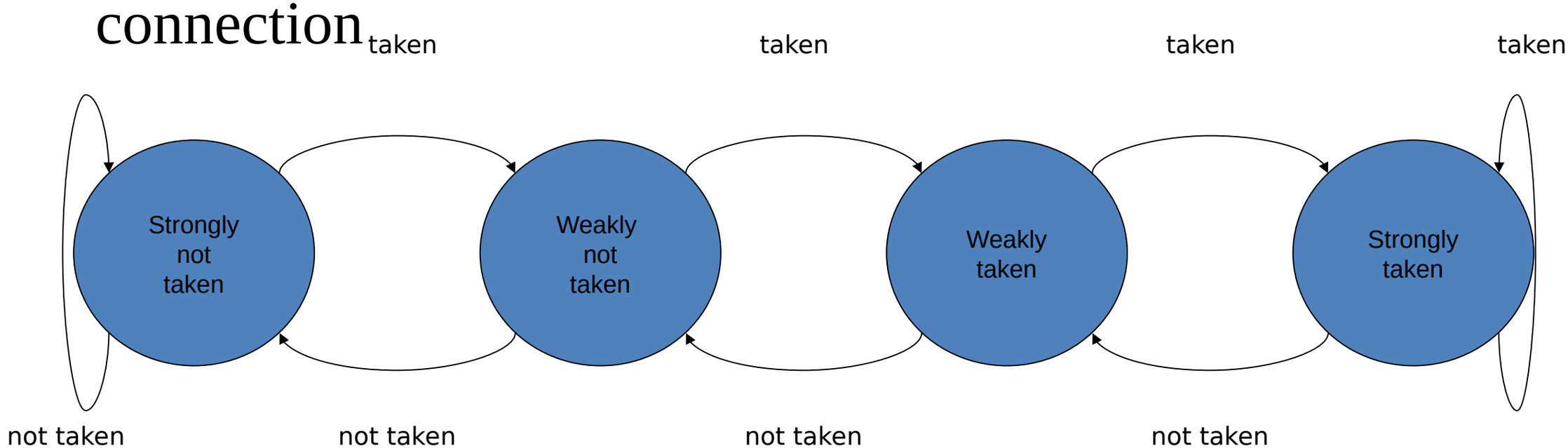
Source: J.N. Amaral

Branch Prediction (1)

- Most used solution on modern processors
- Idea is predict the result of the branch (outcome) taking into account two indications
  - History of the same branch
  - History of the last branch encountered
- Anticipates if a branch is going to be taken or not
- Suppose, for example
  - 98% of connections correctly predicted
  - 2% will be expensive in terms of cycles lost, but they are infrequent

Branch Prediction (2)

- Finite state automaton for predicting a branch
- Consider a 2-bit counter
  - 2-bit saturating up-down counter
  - Change prediction with misprediction in *Weakly* state
  - Predict a branch not taken in *not taken* states
  - Predict a branch taken in *taken* states
  - Update of the current state according to the actual result of the connection



Branch Prediction (3)

- Adding a branch history level
- Storage in *Branch History Register* (BHR)
- Whenever a branch is encountered and decided
  - Shift the BHR to the left one bit
  - Addition of a low 0 if this branch has not been taken
  - Added a 1 otherwise

Two-level Branch Prediction

- Use BHR as an index in a *prediction history table (PHT)*
  - Each entry in the PHT contains a 2-bit counter
  - Basic principle used in modern HPC processors
- Two-level prediction scheme
  - Read the contents of the BHR
  - Access to PHT based on the BHR index
  - Predict the branch according to the associated counter
  - Once the branch result is known, update the BHR and the associated counter
- For more details :
  - Yeh and Patt, *Two-Level Adaptive Training Branch Prediction*, MICRO 1991
  - Yeh and Patt, *Alternative Implementation of Two-Level Adaptive Branch Prediction*, ISCA 1992

Speculative Execution

- Allows the processor to execute instructions beyond the branches speculatively
- Then, if the branch was not correctly predicted, the execution of the speculated operations is canceled (*pipeline dump*)
- If the branch was correctly predicted, the results of the speculated operations are validated in the architectural registers (last step of the pipeline)

Predicates

- Branch predictors and speculative execution are micro-architectural techniques
- Using predicates is an architectural technique
- Idea is to associate boolean registers (predicates) with assembler instructions
- The processor pipeline executes predicated statements regardless of the value of the predicates
- However, the pipeline validates or not the results of the execution according to the value of the predicate
- Predicates are set as a result of conditional statements

Limitations

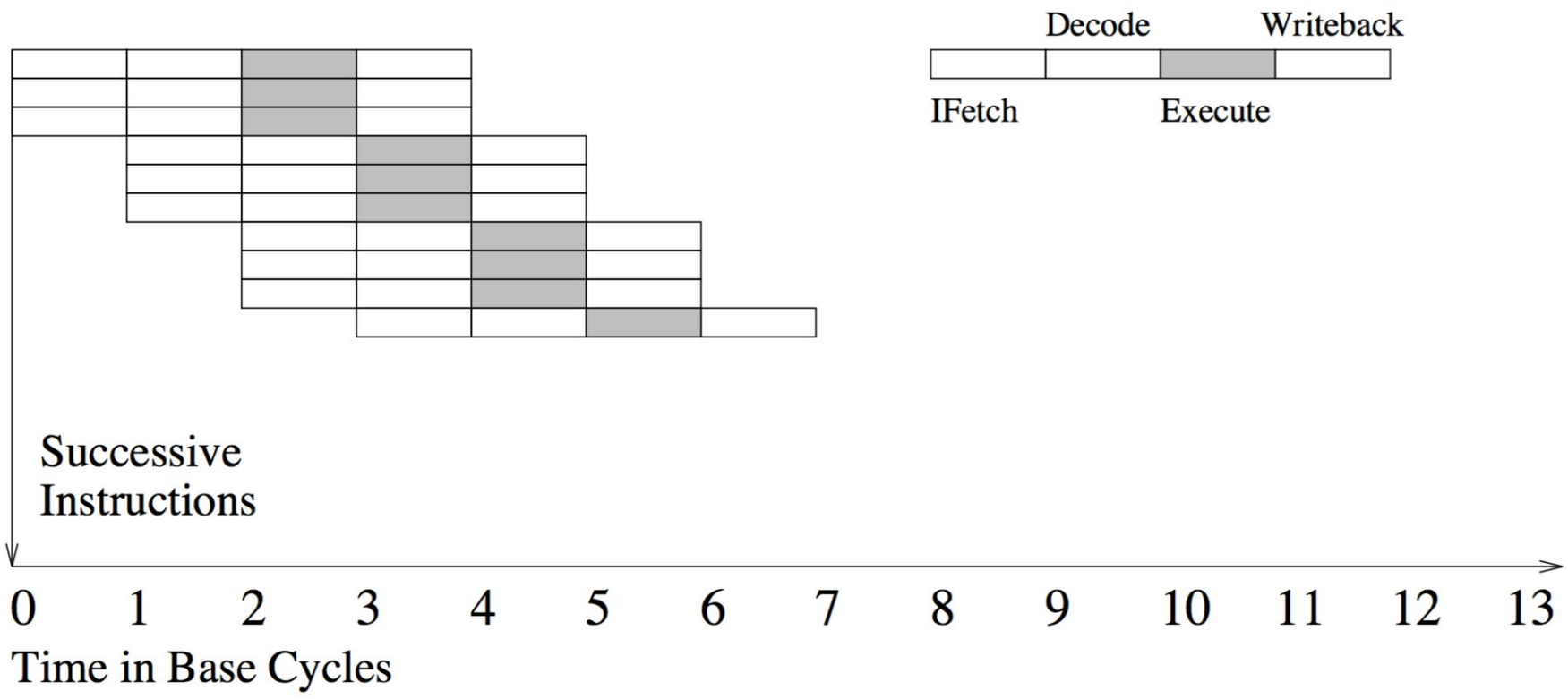
- Maximum rate of one instruction per cycle (IPC = 1)
  - Despite micro-architectural optimizations described above
- In practice, several concerns
  - Conflicts over functional units,
  - Memory access latency
  - Data Dependencies
  - Branches
- Solutions:
  - Temporary storage of the last data accessed in a fast memory (cache and memory hierarchy)
  - Execute multiple instructions at a time

Instruction-Level Parallelism

- Where can we find independent instructions (or independent portions of) to execute?
  - Opportunities for splitting up instruction processing and executing multiple parts of instructions simultaneously
- Ideas so far:
  - Pipelining within instruction and between instructions
  - Overlapped execution and multiple functional units
  - Out of order execution
- Other ideas to consider:
  - Multi-issue execution
  - Superscalar processing
  - Superpipelining
  - Superscalar superpipelining
  - Very Long Instruction Word (VLIW)
  - Hardware multithreading (hyperthreading)

Superscalar Processor Architecture

- A superscalar CPU of degree *N* can issue *N* instructions per cycle
  - Called *multi-issue* execution
- In order to fully utilize a superscalar machine of degree *N* there must be *N* instructions that can execute at all times
- More complex hardware is clearly required to support all the instructions executing
- Superscalar machine were originally conceived of as alternatives to vector machine



Optimizations for Superscalar Processors

- Superscalar processors have a limited window of dynamic optimization
- Limitations of ILP
- Must rely more on the compiler for better IPC
  - Compiler and / or programmer has a more global view
- Superscalar can benefit from dynamic scheduling
- Combination of compiler and hardware can achieve good IPC performance

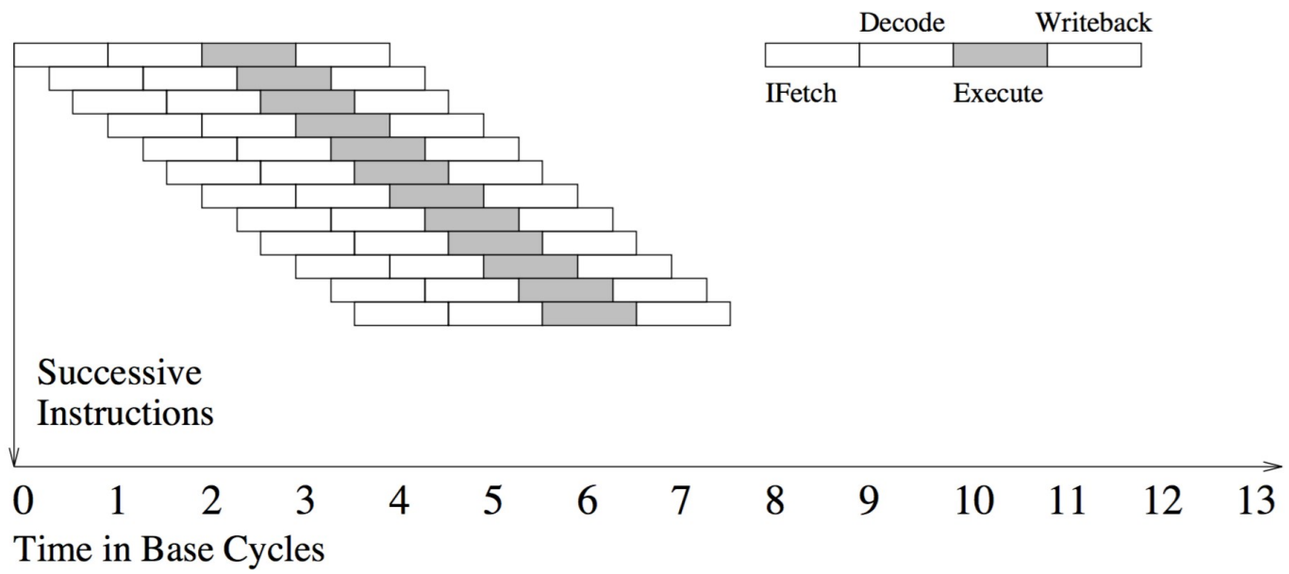
Limitations of Superscalar Processors

- Scalable but limited architectures
  - Ports on registers
  - Pipeline Depth
  - Degrees of parallelism (max current issue = 6)
  - Limits of parallelism in a bounded window
  - High cost
  - Energetic efficiency
- There was a migration from superscalar to multiple cores, hardware multithreading, and so on

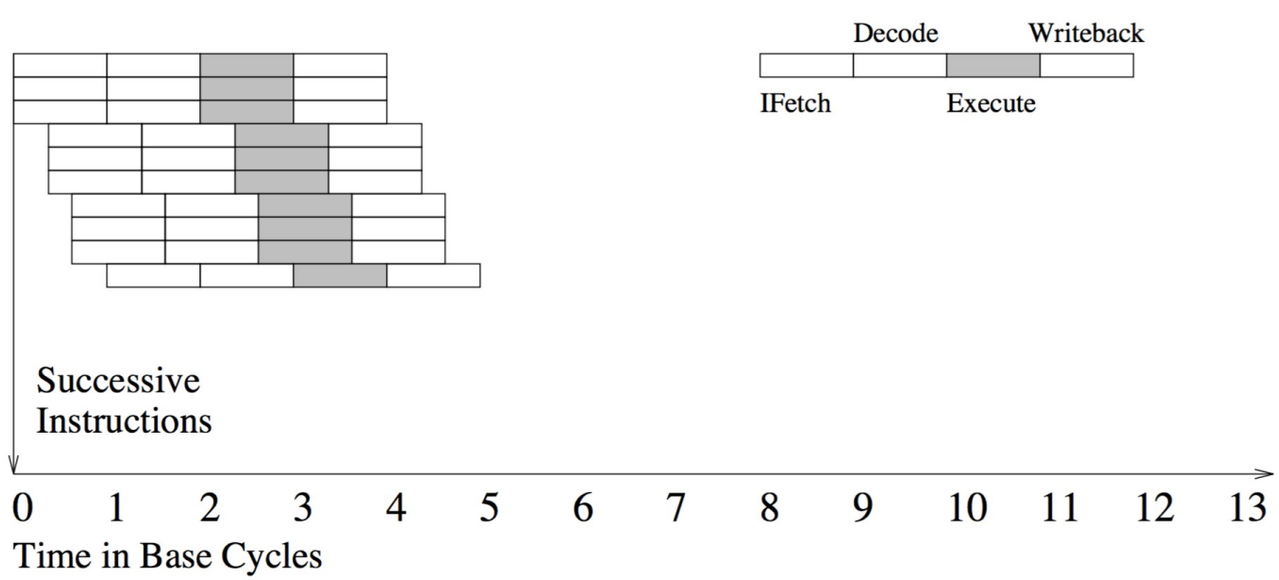
Superpipeline and Superscalar Superpipeline

- Try to reduce the cycle time by dividing each pipeline stage into multiple sub-stages
- Can combine with superscalar

superpipeline



superpipeline
superscalar



Compilers Can Help

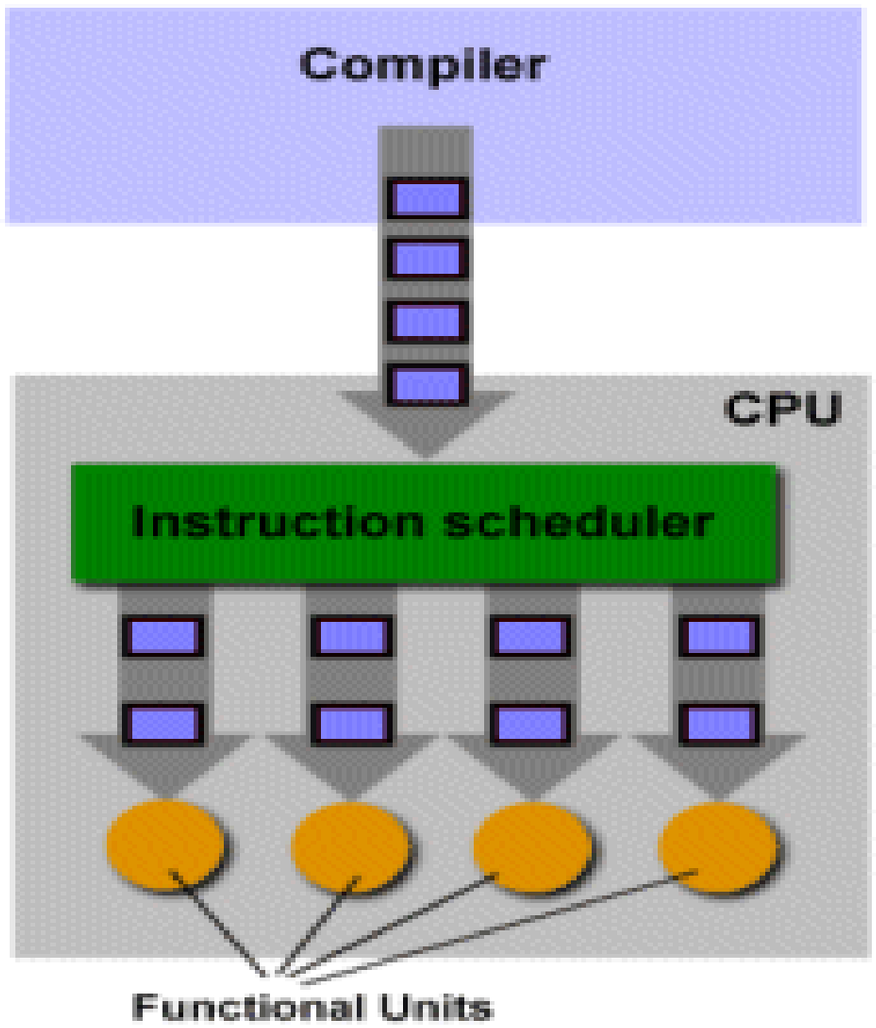
- It can extract parallelism for "well written" codes
- Can schedule instructions in a global window
- Can make a register allocation more intelligent, less aggressive, favoring the extraction of parallelism
- Can generate code with speculative execution
- Can place data, restructure loops to improve spatial and temporal locations of programs (improve caches usage)
- Programmers needs to know the limitations and strengths of optimizing compilers to be able to write effective programs
- Compilers + architecture innovations

VLIW Philosophy

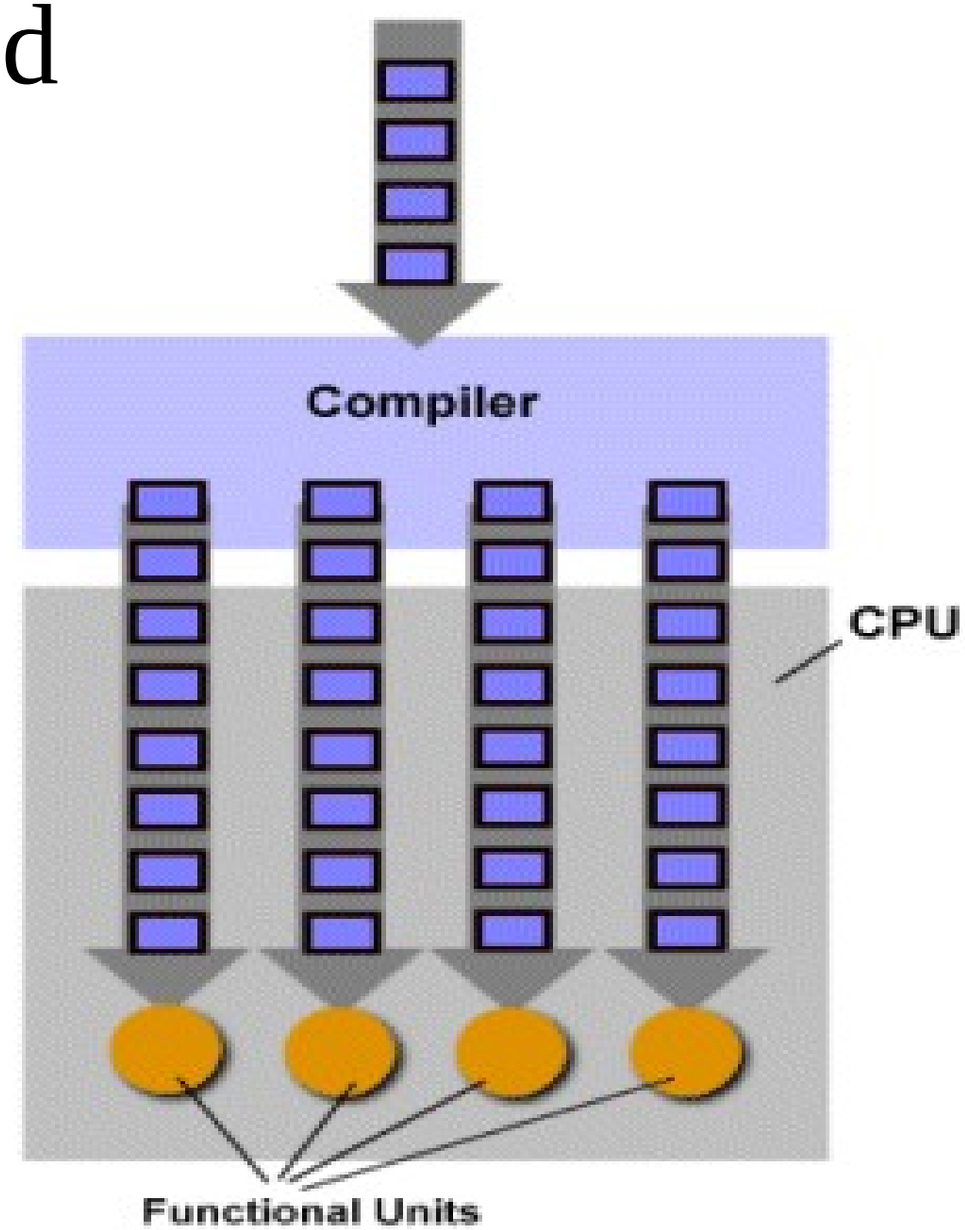
- Let the compiler do its best
  - Optimize as much as it can
- Design a processor that exposes the parallelism at the architectural level
  - Compiled assembler code encodes the parallel packets
- Packets get put in a *very long instruction word* (VLIW)
  - Instruction becomes a concatenation of *n* parallel operations
  - Fixed size
  - Fixed latencies of explicit and non-unitary operations
  - Placement of operations is restricted by the functional units
- Intel Itanium processor family based on VLIW concepts

Superscalar versus VLIW

- VLIW hardware has more flexibility in how instructions can be arranged



Dynamic Superscalar Instruction Scheduling



VLIW Instruction Scheduling

Sequential Code versus VLIW

```
ldw  r1, 0(r2)
sub  r4,r1,r6
and  r6,r1,r7
or   r8,r1,r9
```

```
Cycle 0: ldw    r1, 0(r2)
Cycle 1: …
Cycle 2: …
Cycle 3: sub r4,r1,r6 || and r6,r1,r7
Cycle 4: or r8,r1,r9
```

Sequential code

Superscalar execution

Inst 0:	**ldw r1, 0(r2)**	**; nop**	**; nop**	**; nop;;**
Inst 1:	**nop**	**; nop**	**; nop**	**; nop;;**
Inst 2:	**nop**	**; nop**	**; nop**	**; nop;;**
Inst 3:	**nop**	**; sub r4,r1,r6**	**; and r6,r1,r7**	**; nop;;**
Inst 4:	**nop**	**; or r8,r1,r9**	**; nop**	**; nop;;**

Code VLIW

VLIW format

Scheduling Models

- "Equals" Model (EQ)
  - Each operation lasts exactly its latency
  - For example, the destination register will only be updated when the latency of the operation expires
  - Decreases pressure on registers (better reuse of architectural registers)
  - The exception model becomes complex

- "Less-Than-or-Equals" Model (LEQ)
  - An operation may last less than its latency
  - For example, the destination register can be updated at any time between the launch date and the latency of the operation
  - Simplifies precise management of exceptions
  -

Source : Josh Fisher

Semantics of EQ and LEQ

R1 is considered free here

```
Inst 0:  lw r1, 0(r2); nop              ; nop;              nop;
Inst 1:  nop           ; nop              ; nop;              nop;
Inst 2:  nop           ; nop              ; nop;              nop;
Inst 3:  nop           ; sub r4,r1,r6     ; and r6,r1,r7;     nop;
Inst 4:  nop           ; or r8,r1,r9      ; nop; nop;
```

VLIW code with EQ semantics

```
Inst 0:  lw r1, 0(r2); nop              ; nop;              nop;
Inst 1:  nop           ; sub r4,r1,r6     ; and r6,r1,r7;     nop;
Inst 2:  nop           ; or r8,r1,r9      ; nop;              nop;
```

VLIW code with LEQ semantics

Advantages of VLIW

- Clean, simple and more scalable architecture
- High degree of parallelism of instructions
- Low cost for micro-architecture
- Dynamic executions are visible architecturally
  - More predictable codes

Disadvantages of VLIW

- Works well only for programs with statically extractable parallelism
  - Requires a very powerful optimizing compiler
- Very sensitive to cache layout
- Code size generated can have a significant number of NOPs
- No binary compatibility between generations of the same family

Compacted VLIW Instructions

Opc Dst Src1 Src2	Opc Dst Src1 Src2	Opc Dst Src1 Src2	Opc Dst Src1 Src2
Operation 1	**Operation 2**	**Operation 3**	**Operation 4**

- If not compacted, the VLIW instructions are wide ...
- Compilers can not always fill all holes
  - Functional units and latencies
- Code space is not free
  - Embedded, it is even very expensive
- How to reduce code size?
  - Remove NOPs
  - Use a compact format
- With variable VLIW instruction encoding (stop-bit)
- Compact code

Source : Josh Fisher

Differences Between Superscalar and VLIW

	Superscalar	VLIW
Instruction flows	The instructions are loaded with a sequential stream of scalar operations	The instructions are loaded with a sequential stream of multiple operations
Loading and scheduling instructions	The loaded instructions are dynamically scheduled by the hardware	The loaded statements are statically ordered by the compiler
Width of issue	The number of instructions loaded in parallel is fixed dynamically by the hardware	The number of instructions loaded in parallel is fixed statically by the compiler and the architecture
Order execution instructions	Dynamic loading allows execution in order and out of order	Static scheduling only allows one execution in order
Architectural Implications	The superscalar is a micro-architectural technique	VLIW is an architectural technique since hardware details are visible to the compiler / programmer.

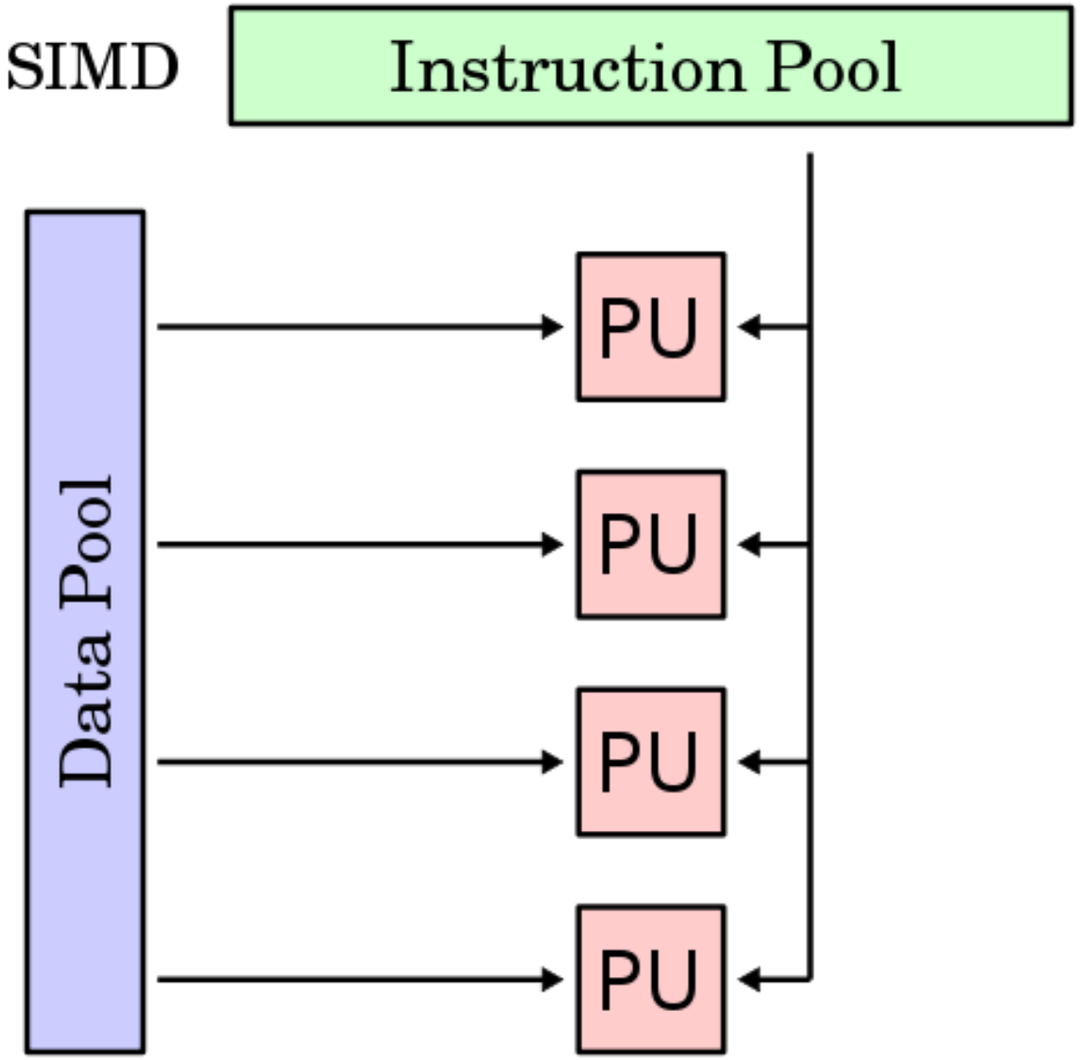
Source : Josh Fisher

Vector Extension

- Optimization possibilities
  - Duplication of functional units
    - exploited by superscalar and out-of-order
  - Expansion of functional units
    - towards SIMD programming
- Flynn's taxonomy
  - *SISD*: Single Instruction Single Data
  - *MISD*: Multiple Instruction Single Data
  - *SIMD*: Single Instruction Multiple Data
  - *MIMD*: Multiple Instruction Multiple Data

SIMD Instructions

- Idea is to use a single instruction (from the ISA) that will operate on multiple data at the same time
- Where is the data?
- In addition to scalar operands, the registers are extended to contain multiple scalar data (call a *vector*)
  - Vector registers



Advantages of SIMD

- Expands calculation performance at lower cost
  - More transistors to build these units
  - Depends on the operation
    - vector addition is like a simple addition without propagation
  - Chaining between vector operations is possible
- Minimal changes to the rest of the processor
  - No expansion of the pipeline as for a superscalar
  - No need to discover the parallelism of data
    - done by the compiler

Disadvantages of SIMD

- Exhibits optimization at the level of the architecture
  - Instruction set must provide multiple operations on data
  - Vector instructions support certain vector size
    - SSE (128 bits), AVX (256 bits), MIC/AVX512 (512 bits)
  - Need to generate specific code
- Imposes a certain degree of precision and parallelism
  - Example: 16 single-precision floats on MIC
- Applies to particular code classes
  - Problems with control flow
  - Regular scientific codes
  - Cryptography
  - Any code where simultaneous operations on multiple values of the same type are present and not difficult to discover

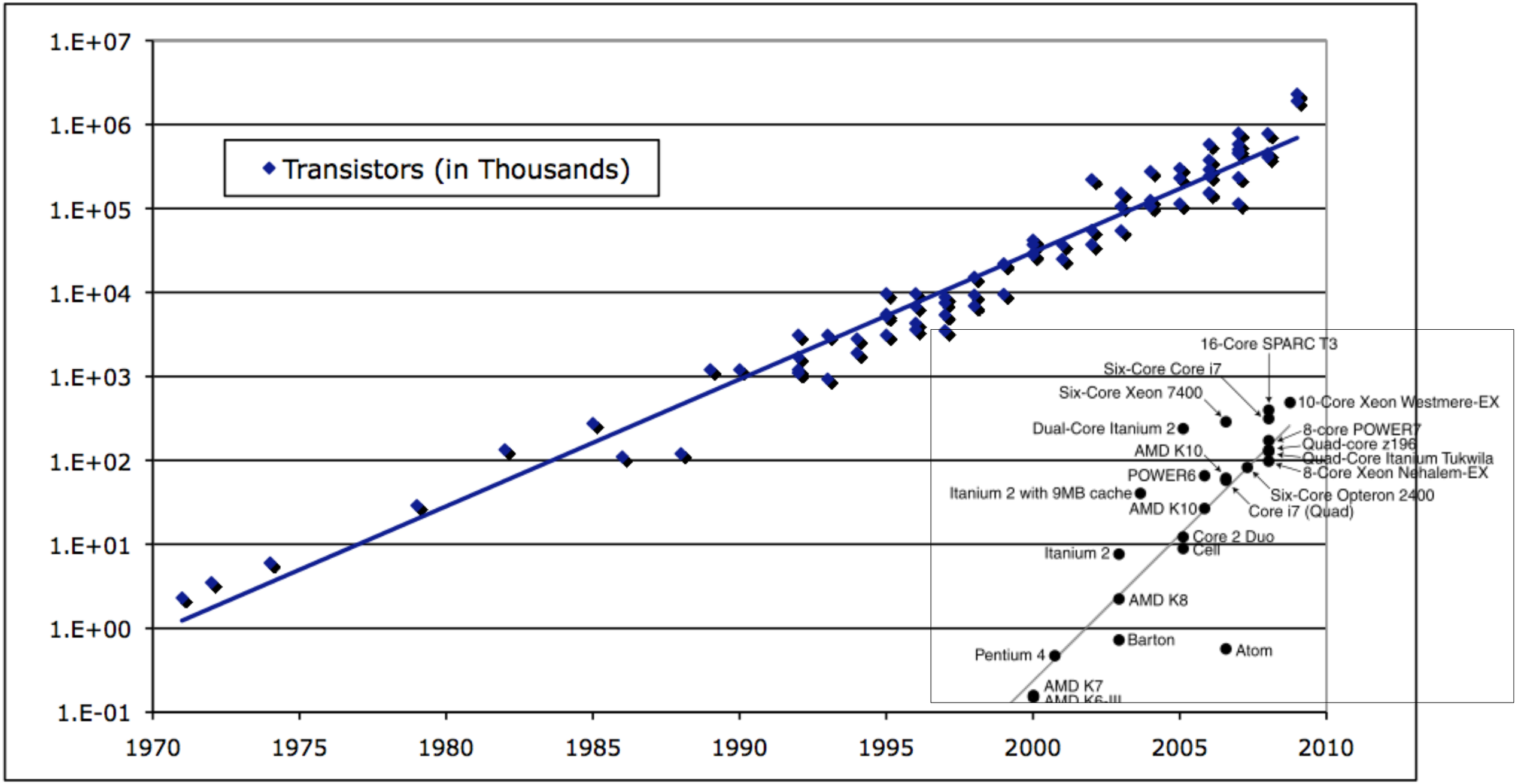
Conclusion from CM-1

- Steps to optimize a calculation core
  - Concept of pipeline
  - Mechanisms to avoid branches
    - out-of-order execution
    - branch prediction
  - Execute multiple instructions at a time
    - superscalar
    - VLIW
  - Multiple calculations per instruction
    - SIMD vector extension

Leveraging Moore's Law

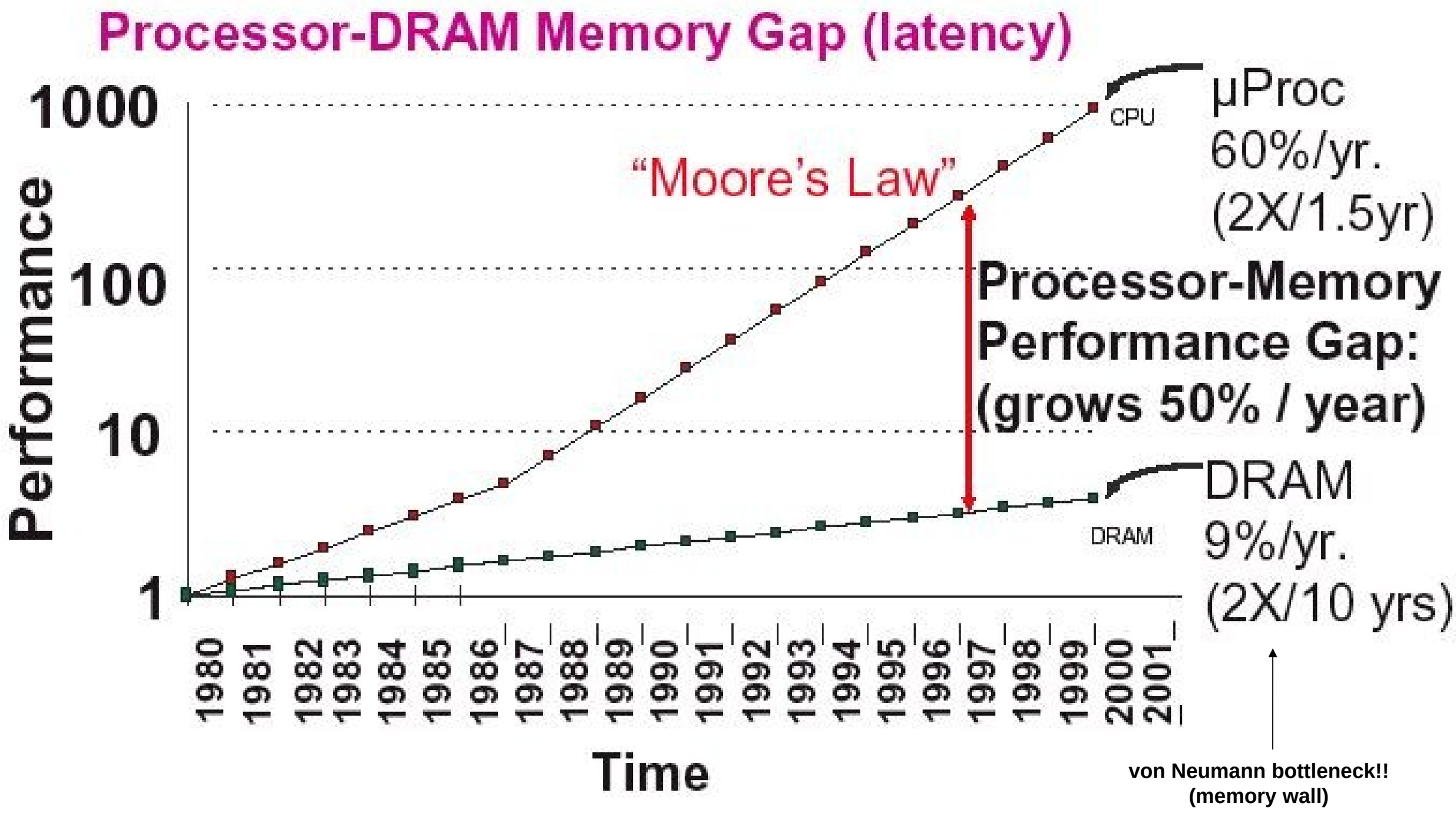
- Moore's Law (Gordon E. Moore, Intel co-founder)
  - # transistors in an integrated circuit doubles every 2 years
  - Observation or conjecture, not a physical or natural law
  - End of Moore's Law?
    - expected to continue to ~2020
- More transistors = more parallelism opportunities
- Microprocessors
  - Implicit parallelism
    - pipelining
    - multiple functional units
    - superscalar
  - Explicit parallelism
    - SIMD instructions
    - long instruction works
- Dennard scaling
  - Power requirements are proportional to area
  - Performance per watt grows with transistor density
  - Dennard scaling has broken down since 2007!

Microprocessor Transistor Counts (1971-2011)



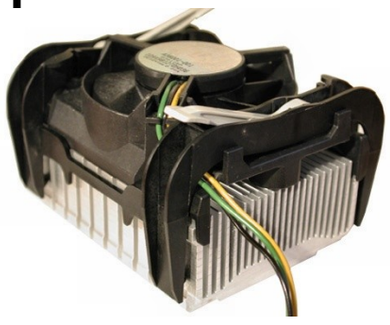
Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç
Slide from Kathy Yelick

What's Driving Computing Architecture?



What has happened in the last 15 years?

- Processing chip manufacturers increased processor performance by increasing CPU clock frequency
  - Riding Moore's law
- Until the chips got too hot!
  - Greater clock frequency greater electrical power
  - Breakdown of Dennard scaling
  - Pentium 4 heat sink
  - Frying an egg on a Pentium 4
- Add multiple cores to add performance
  - Keep clock frequency same or reduced
  - Keep lid on power requirements



Power Density Growth

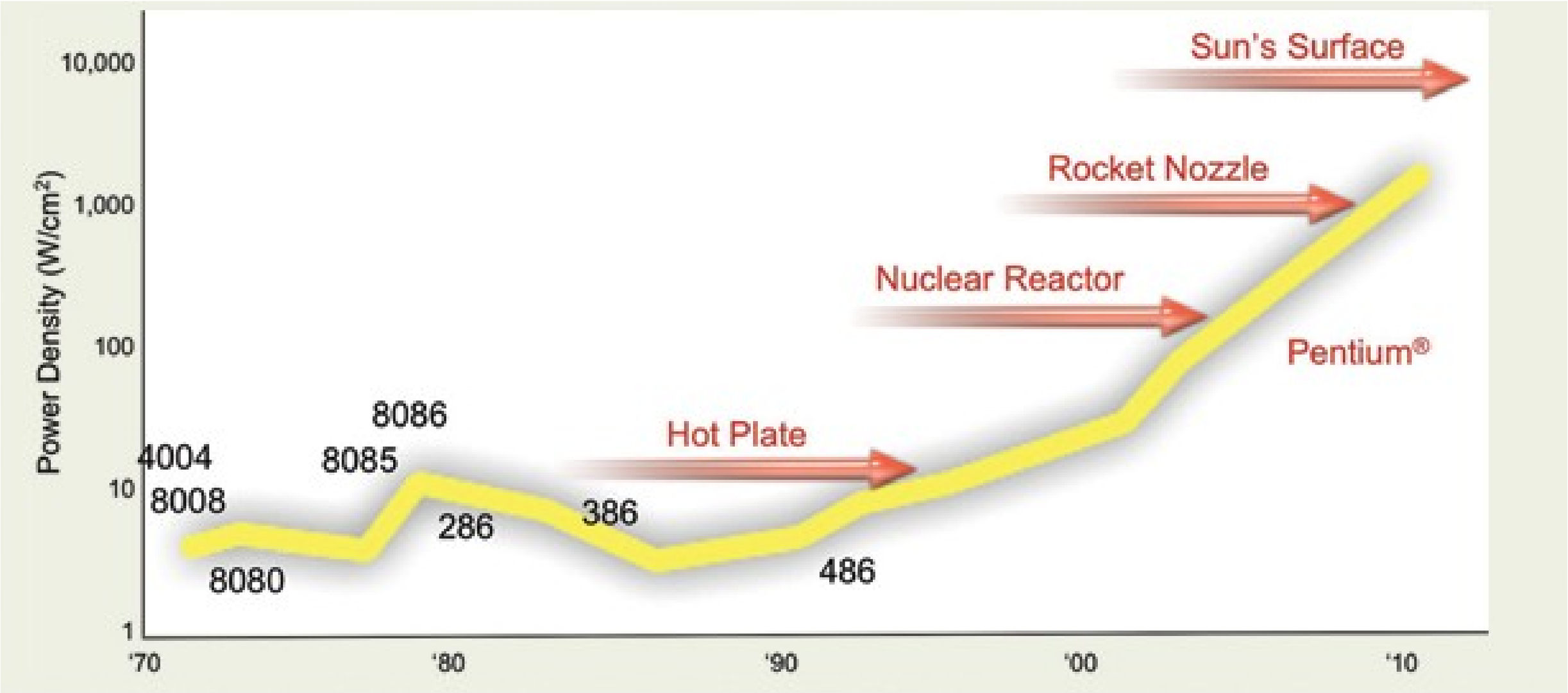
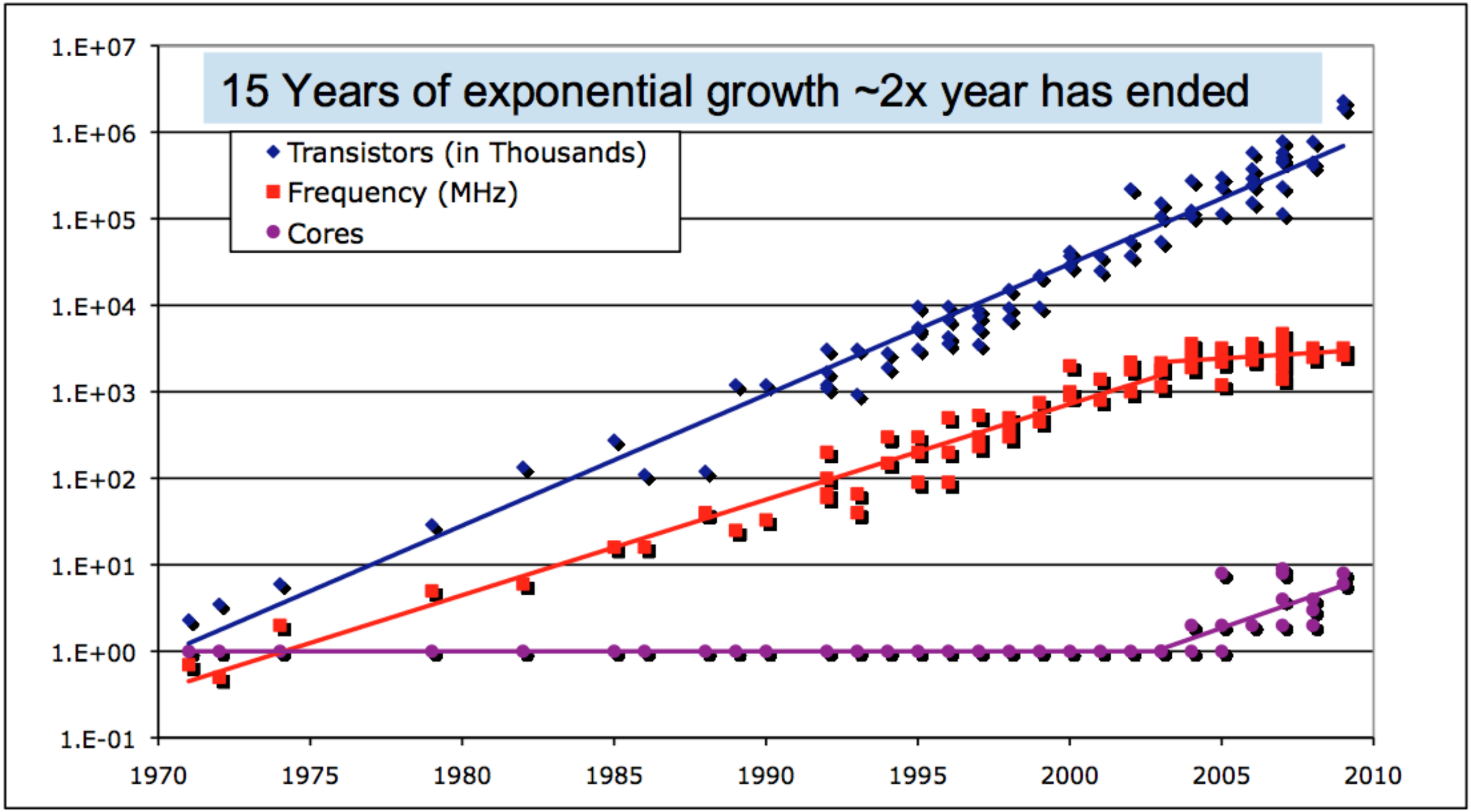


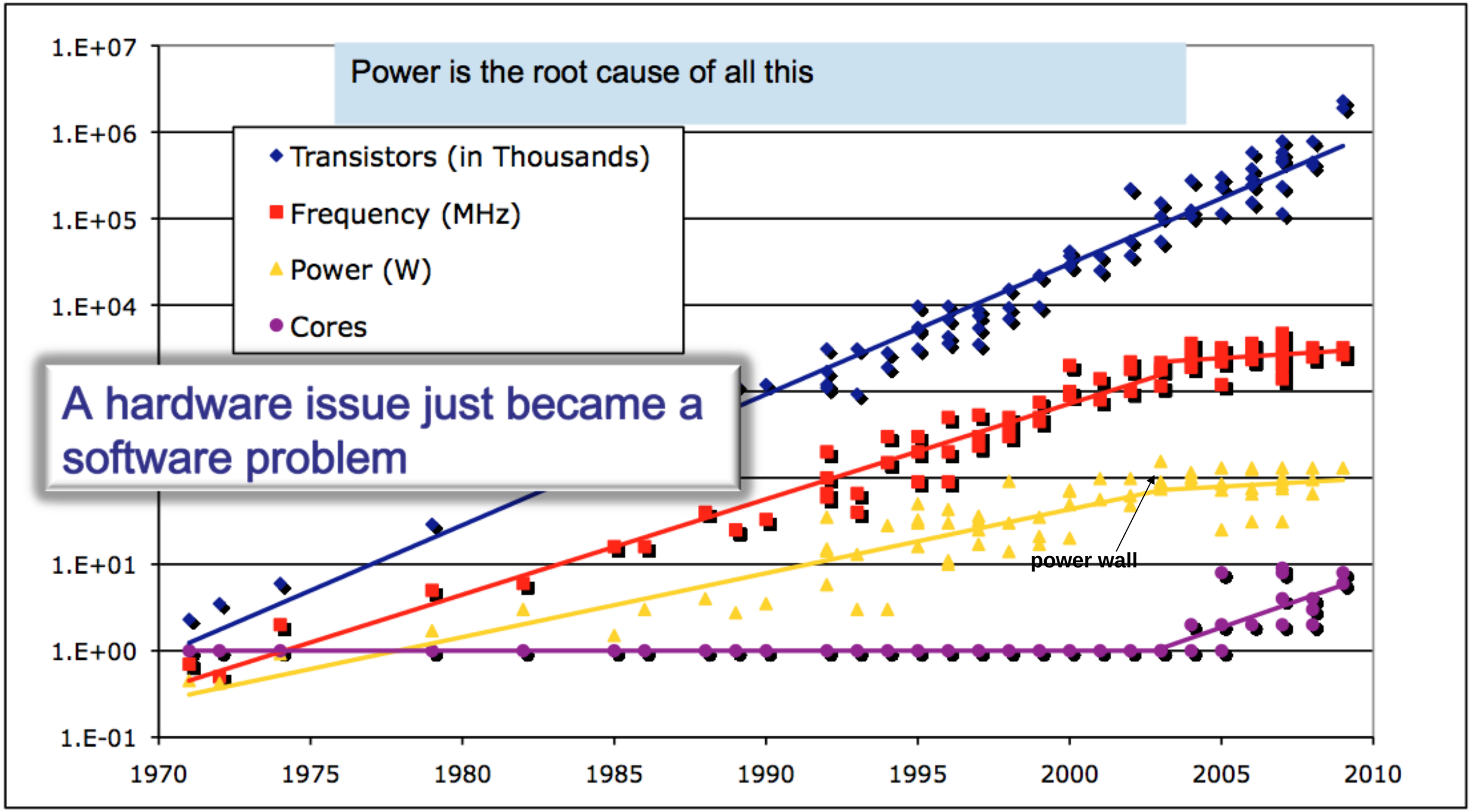
Figure courtesy of Pat Gelsinger, Intel Developer Forum, Spring 2004

What's Driving Parallel Computing Architecture?



Data from K. Olukotun, L. Hammand, H. Sutter, B. Smith, C. Batten, K. Asanovic; Slide from Kathy Yelick

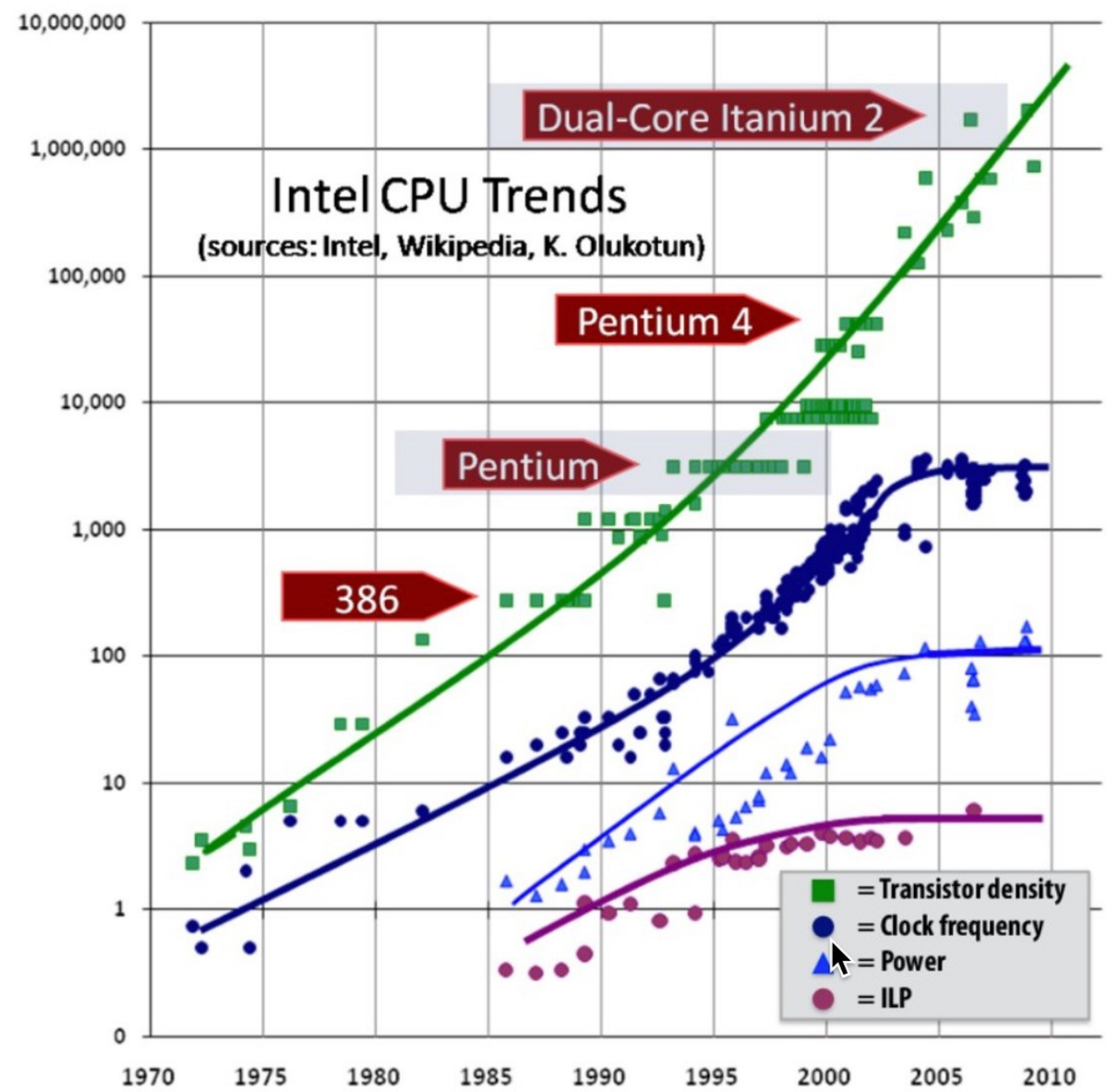
What's Driving Parallel Computing Architecture?



Data from K. Olukotun, L. Hammand, H. Sutter, B. Smith, C. Batten, K. Asanovic; Slide from Kathy Yelick

Single-core Performance Scaling

- Rate of single-instruction stream performance scaling has decreased to almost 0
- Frequency scaling is limited by power
- ILP scaling is tapped out
- CPU architects are now building faster processors by adding more execution units to run in parallel
- Parallel software must be written to see performance gains

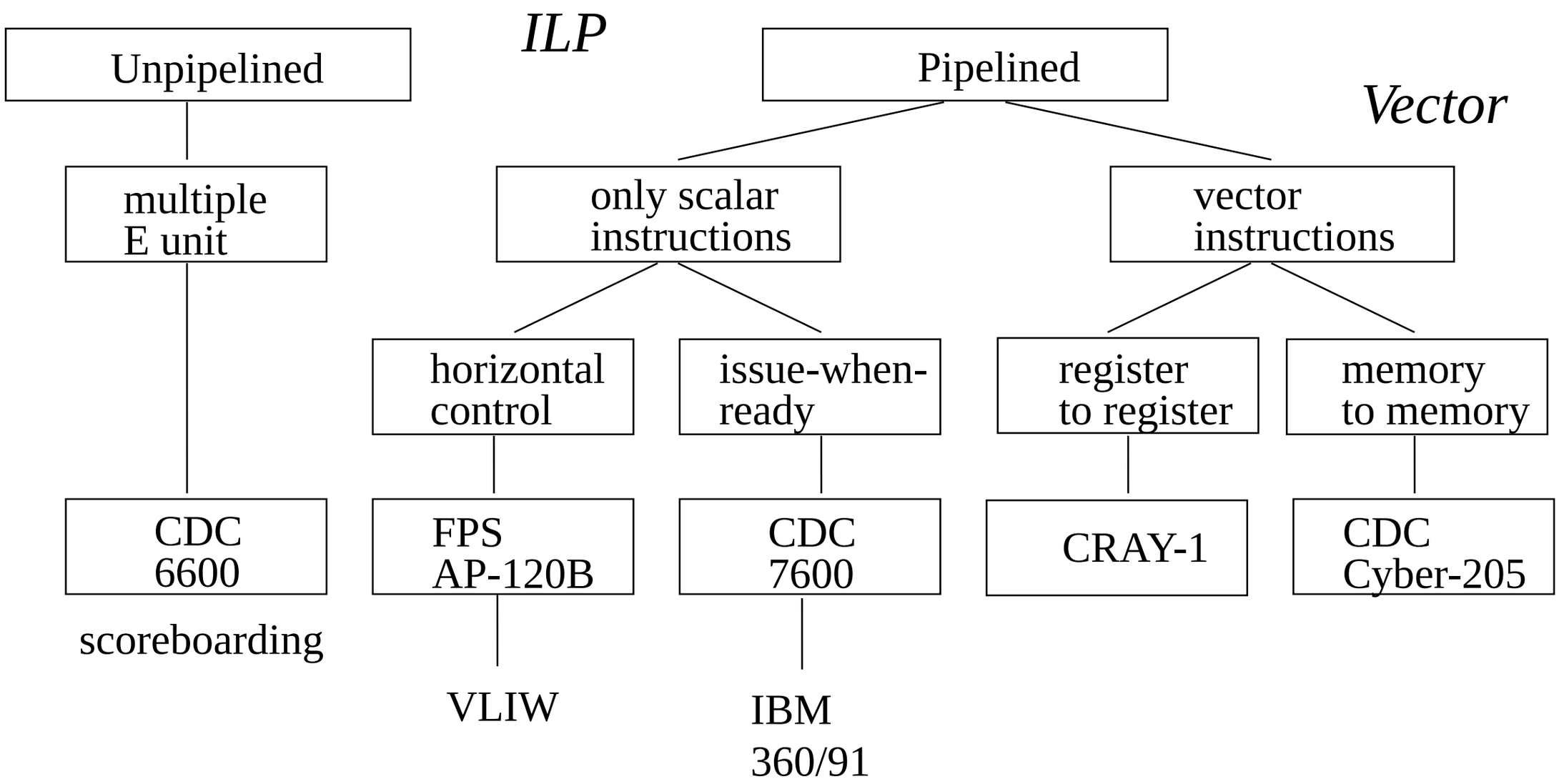


Phases of Supercomputing (Parallel) Architecture

- Phase 1 (1950s): sequential instruction execution
- Phase 2 (1960s): sequential instruction issue
  - Pipeline execution, reservations stations
  - Instruction Level Parallelism (ILP)
- Phase 3 (1970s): vector processors
  - Pipelined arithmetic units
  - Registers, multi-bank (parallel) memory systems
- Phase 4 (1980s): SIMD and SMPs
- Phase 5 (1990s): MPPs and clusters
  - Communicating sequential processors
- Phase 6 (2000s): many cores, accelerators, scale, …
- Phase 7 (2010s): many more cores, heterogeneity, …

Parallelism in Single Processor Computers

- History of processor architecture innovation



Vector Processing

- Scalar processing
  - Processor instructions operate on scalar values
  - Integer registers and floating point registers
- Vectors
  - Set of scalar data
  - Vector registers
    - integer, floating point (typically)
  - Vector instructions operate on vector registers (SIMD)
- Vector unit pipelining
- Multiple vector units
- Vector chaining

Liquid-cooled with inert fluorocarbon. (That's a fluorocarbon fountain!!!)

Cray 2